Quake 3 Network Protocol

Jacek Fedoryński
jfedor@jfedor.org

January 2020

Quake III Arena is a networked multiplayer first-person shooter developed by id Software and released in December 1999. The game’ s source code was released under the terms of GNU General Public License in August 2005.

We go over the network component of the game’ s engine, from the general principles of operation down to the actual bits of the on-the-wire protocol. Along the way we describe some elegant features of the engine, like the event system and the separation of the engine code from the game logic using virtual machines.

Companion software

Even though we try to provide enough information to implement the network protocol, sometimes code is easier to understand than a natural language description of what it does. A from-scratch implementation of the protocol is released as a companion to this text: a proxy that parses all the messages exchanged between the server and the client, optionally rewriting some of the packets to provide a proof-of-concept aimbot functionality. In addition to helping with understanding the protocol, the proxy might also be useful for experimentation, as it provides the ability to simulate network latency and packet loss. Its source code is available at https://github.com/jfedor2/quake3-proxy-aimbot.

Overview

Quake 3 works in a client-server model: all the players participating in a match are connected to the same server. The client, running on the player’ s machine, is responsible for sampling input (keyboard, mouse) from the player and sending it to the server. The server, located on the local network or somewhere on the Internet, runs the simulation and sends the state of the world to the client, where it is rendered for the player to see. Even though the server is the authority on what happens in the game, the client is not a “dumb” one. In order to mitigate the effects of network latency, parts of the simulation pertaining to the local player’ s movements are also performed client-side. The mechanism, called *client prediction* and first introduced in QuakeWorld in 1996, is one of the reasons a fast-paced multiplayer game like Quake III Arena could comfortably be played over high-latency dial-up modem connections that were popular at the time of the game’ s release.

As a special case, when the game is played in single-player mode (against bots), the client and the server are running on the same machine, in the same process, but the principles of operation remain the same.

Engine vs. game code

Quake 3’ s code is divided into two main parts: the *engine* part and the *game* part. The game part contains the rules of the game. It defines how different objects in the game interact with one another: things like what happens when a rocket hits a player or what happens when a player’s health drops below zero. The engine part contains the operating system specific code (graphics, sound, input, network), the renderer, the bot library, and some other things like the specifics of the on-the-wire network protocol. Both the engine and the code parts are run on the server as well as on the client.

This divide is very interesting for more than one reason. The entire source code for Quake III Arena was released under the terms of GNU GPL in 2005. Before that only companies that licensed the engine from id Software could get their hands on both the engine and the game part. Games like *American McGee’s Alice* and *Return to Castle Wolfenstein* were developed under such an agreement. But a long time before the full source code was made available, almost immediately after the game’s release, the source code for just the *game* part was released publicly, no paid license necessary. This allowed anyone to create and distribute so called *mods*, modifications to the game’s rules. Combined with the ability to create own game assets (maps, models, textures), it made it possible to make almost entirely new games, with a distinct look and gameplay. This friendliness towards developers of mods and *total conversions* was common in the Doom and Quake series of games. It didn’t hurt id Software financially, because people running the mods still had to own the original game.

Not all of the modifications had to be entirely new games. Some of them only introduced new rules for scoring or modified the game’s physics. What’s most interesting from this text’s point of view is that large parts of the network code, responsible for the core game behavior, were located on the game side. It was therefore possible for enthusiasts to make mods that changed how the network latency affected the players. One such mod, named *Unlagged*, aimed to compensate for the network lag by calculating where the players *saw* other players when trying to shoot them, instead of where the players actually were from the server’s authoritative point of view at that time.

Virtual machines

The separation between engine and game code raises some interesting technical questions. Because of the existence of mods, the game code should be treated as untrusted. Quake 3 includes the ability to automatically download the game code to the client when joining a server that’s running a mod that the client has not seen before. Allowing the server operator to make the client run arbitrary native code is an instant red flag from a security point of view.

To deal with this, the game part of the code in Quake 3 is compiled to bytecode for a virtual machine created for this purpose. This limits what the game code can do and prevents it from accessing the rest of the user’s system. But interpreting bytecode during gameplay wouldn’t be optimal from a performance point of view. To get the sandboxing benefits of a virtual machine and the performance of native code, Quake 3 compiles the bytecode to native code when loading the game. Or at least that’s what it does by default if running on one of the supported architectures (x86, PowerPC). It still keeps the ability to run the bytecode in interpreted mode. Such virtual machine setup has one big downside, even when performance is no longer a problem: it makes it harder to debug game code during development. Because of this, the Quake 3 game code can also be compiled to native code directly, along with the rest of the engine, and loaded as a dynamic library, making it more friendly to the debugger.

There are three virtual machines in Quake 3, one simply named *game*, which is the server side of the game logic, one named *cgame*, short for *client game*, which is the client side of the game logic, and one named *ui*, responsible for the client user interface, which means the game menus can also be customized in mods.

To communicate between the engine code and the game code, Quake 3 has *VM calls* and *syscalls*. For example, when the engine wants to ask the game to run calculations for a single server frame, it makes a `GAME_RUN_FRAME` VM call, which runs the `G_RunFrame()` function inside the VM. And when the client game code wants to ask the engine to render a single frame, it makes a `CG_R_RENDERSCENE` syscall, which ends up calling the `RE_RenderScene()` function in the renderer library. All this happens on the same single thread, there is no multithreading in Quake 3, except optionally in the renderer.

Event system

Everything in Quake 3, both client and server, happens in response to events. Player input like mouse movement and keypresses, and also packets received from the network, all go through a unified event system. Even the passing of time is communicated to the engine using a separate type of event. In addition to decoupling the engine from operating system specific code, this opens up an interesting possibility. During normal gameplay it’s possible to record all the events going through the queue in a journal file. Then it’s possible to start the game in a special mode where it only processes events from the journal file saved earlier. In this mode the engine ignores all normal input like mouse, keyboard, network and even time. The gameplay session that results from replaying such a journal is deterministically identical to the original session, even if some external conditions change. Because the passing of time is also journalled, the game will render the same frames, running at the same rate of frames per (virtual) second. This ability to replay a session is a very valuable tool when debugging hard to reproduce errors.

Demos

While event journalling is a feature designed to help the developers debug their code, there’s another mechanism, this time created for the players to use, that also makes it possible to record and then replay gaming sessions. Before screen recording and streaming live over the Internet was practical, players would share demo files to show their accomplishments to others. Demo files are similar to event journals in that they store the progress of the game in a file, but they’re different in that instead of storing all the events, they store the network packets that the client received from the server during the recorded session. Those packets contain enough information to recreate the session.

Playing back a demo is different than playing back a journal file. While the gameplay does look the same as during the original session, the objective this time is not to retrace the exact code paths that were executed when the recording was done. Because the time passes naturally when playing back a demo (instead of being read from the journal), if the computer that is doing the playback is faster or slower than the original machine, it might render a different number of frames. Even on the same computer, though it will render a similar number of frames, they will most likely not happen at the exact same timestamps as when the demo was recorded.

A special mode of demo playback called timedemo exists to allow measuring how fast a particular machine can run the game (benchmarking). In this mode, the frames are processed as fast as possible, but the virtual in-game time is stepped a fixed amount of 50 milliseconds each frame. So the game runs at a virtual rate of 20 frames per second and the actual frame rate is measured using real clock. At the end of demo playback, an average of that frame rate is printed out and the user can compare it to the results of replaying the same demo in timedemo mode on another machine.

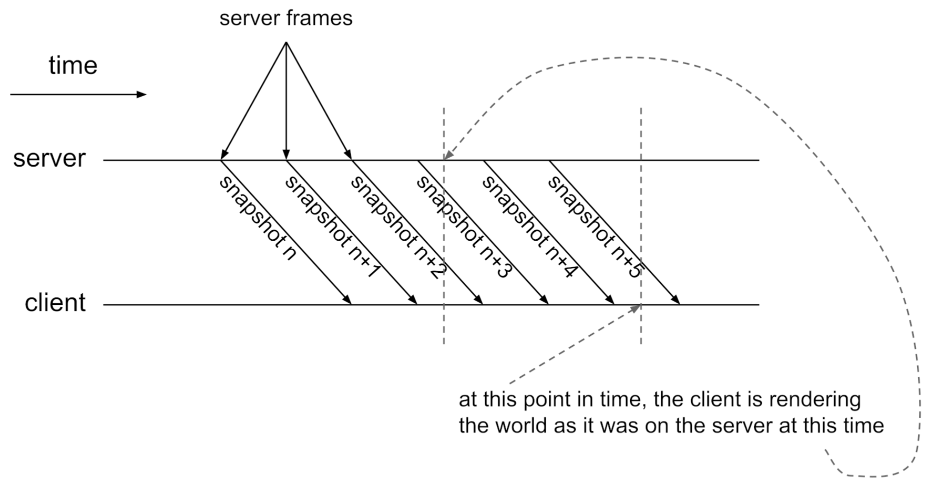
Principle of operation

Let’ s take a closer look at what happens on the client and on the server and at the information flow between them.

The server runs its frames at a fixed rate of 20 per second. In each frame it goes through all of the entities (objects present in the game world) and lets them “think”. This is when rockets move according to their velocities, damage is inflicted and other game logic is run, including checking conditions like has the fraglimit been hit. Everything except the players’ movement happens in this server frame calculation. After executing the game logic and determining the current state of the world, the server communicates that state to all clients in what is called *snapshots*.

When do the players move on the server? They are moved *between* server frames, immediately after a network packet containing the information about the movement intended by the user (called *user commands*) is received by the server. We’ ll get back to this when we discuss what happens on the client side.

The client runs its frames as fast as it can (up to 125 Hz or the display refresh rate if V-sync is enabled). The frame rate of the client may vary, but it will usually be faster than the 20 Hz used by the server. What’ s the point of processing more frames on the client than on the server? After all, if the server is the authority on the state of the world and the client is only getting this state 20 times per second, won’ t it be rendering the same image a few times until it gets another update? Not quite. To make the visuals smoother, the client is interpolating between two states given to it by the server. If it knows the state of the world at time *t* and at time *t+50 ms* and it needs to render additional frames between those points in time (because it has a nice graphics card that can run the game at 60 FPS), it interpolates the positions of all visible objects between their known two states. That means that when the client is rendering the frame at *t+16 ms*, it already needs to have received the information about the server frame from *t+50 ms*! The only way that is possible is if the client intentionally delays its view of the world in relation to what it’ s receiving from the server. That is in addition to the delay caused by the network: when the client is receiving a network packet from the server, the information it contains is already out of date by a number of milliseconds the packet needed to travel from the server to the client. That number is practically zero on a local network, but definitely non-negligible on 1999 Internet accessed via a dial-up modem.



What happens when the network packet containing the next snapshot is delayed or lost and the client runs out of states to interpolate between? Then it’ s forced to do what it normally tries to avoid: extrapolate or guess where the objects will be if they keep moving the same way they’ re currently moving. This might not be a big problem for objects that move in a straight line like rockets or that are only affected by gravity. But other players move in an unpredictable manner and because of that the game normally wants to interpolate and not extrapolate, even if it means artificially delaying everything by up to a single server frame in addition to whatever network lag is present.

The above description applies to all game objects except the most important one: the player. This case is special, because the player’ s eyes are the first person camera through which we are seeing the game world. Any latency or lack of smoothness here would be much more noticeable than with other game objects. Which is why sending user input to the server, waiting for the server to process it and respond with the current player position and only then rendering the world using that delayed information isn’ t a good option. Instead the Quake 3 client works as follows. In each frame it samples user input (how much the mouse was moved in which direction, what keys are pressed), constructs a user command, which is that user input translated into a form independent of the actual input method used. It then either sends that user command to the server immediately or stores it to be sent in one batch with future user commands from subsequent frames. The server will use those commands to perform the player movement and calculate the authoritative world state. But instead of waiting for the server to do that and send back a snapshot containing that information, the client also immediately performs the same player movement locally. In this case there’ s no interpolation - the move commands are applied onto the latest snapshot received from the server and the results can be seen on the screen immediately. In a way this means that each player lives in the future on their own machine, when compared to the rest of the world, which is behind because of the network latency and the delay needed for interpolation. Applying user inputs on the client without waiting for a response from the server is called prediction.

Usually the client’ s predictions about player movements will match the server’ s calculations exactly. But it may happen that they arrive at different outcomes. For this to occur, something must happen on the server that the client didn’ t know about at the time, like a collision with another player or knockback after being hit with a rocket. When that happens, the client will of course have to correct its idea of the player’ s position and orientation, but it will try to avoid just snapping to the coordinates provided by the server. If the error is small, it will smoothly transition between the erroneous position and the correct one over the next few frames.

Network protocol

All network communication in Quake 3 happens over UDP. The UDP protocol is an unreliable one, in the sense that the individual datagrams (packets) may arrive out of order, duplicated, or not arrive at all. The network layer in the operating system doesn’ t have any confirmation or retransmission mechanism and it’ s up to the application to implement those where necessary. (Just like latency, packet loss is almost never a problem on a local network, but it happens often enough on the Internet.)

This unreliable model suits a fast-paced real-time game like Quake 3 well - it makes little sense to trust the network layer of the operating system with acknowledging and retransmitting packets. By the time the OS knows that a packet was lost on the network (because a certain amount of time passed without the packet being acknowledged by the other side), the information it contained is very much out of date anyway. Even if some of the information does indeed need to be retransmitted, it’ s better to handle this at the application level, as the application understands the format and meaning of the message and can decide to retransmit some parts and replace others with more up-to-date information.

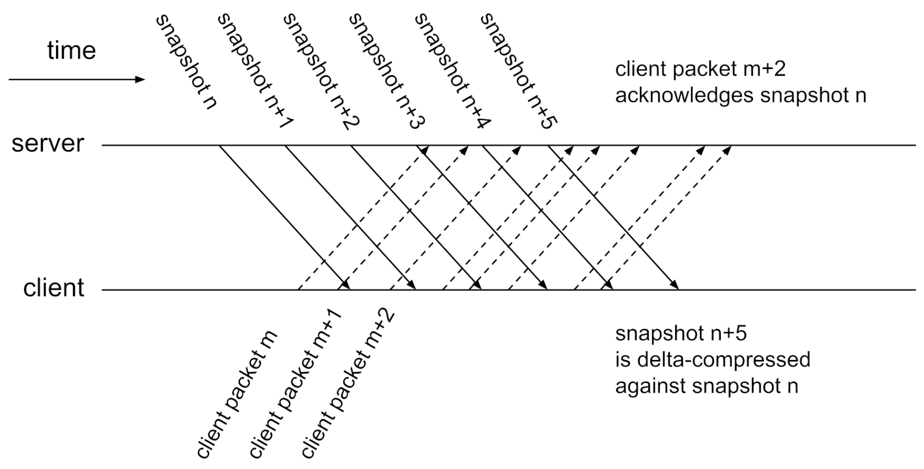
In Quake 3, different types of messages are sent from the server to the client and from the client to the server. If we only consider the messages exchanged during normal gameplay (ignoring the initial handshake and level restarts), the packets sent by the client contain user commands (player’s moves) and *client commands*, while the packets sent by the server contain snapshots (state of the world) and *server commands*.

Client commands and server commands are text commands that are used for things like player chats and sending scores. They are reliable messages so they must be confirmed and retransmitted if necessary. Every client and server command has a sequence number and every packet sent by the server or the client contains the sequence number of the last command received from the other side. How does the client and the server decide when to retransmit a command? They just retransmit all commands until they are acknowledged. This means that if the network latency is high enough, commands will be repeated even if there is no packet loss, if the interval between packets is shorter than the time a packet needs to reach the other side and the confirmation to return.

User commands containing player movement are not reliable in the sense that they absolutely need to be received by the server (after some time they’re useless anyway), so the client won’t retransmit them indefinitely. But we would still prefer if all of them reached the server, even in the face of packet loss. So while there is no direct confirmation mechanism, user commands are also retransmitted by default. But instead of retransmitting them until confirmed, the client just repeats them a fixed (configurable) number of times. By default each user command is sent twice. That way if one packet is lost, the server still gets the player’s movement in the next packet.

User commands contain a timestamp, but that value does not come into consideration with regard to when the move happens on the server. It happens immediately after the command is received and the timestamp from the user command is only used to compare against the previous command to see how far the player should be moved (absolute time is used, not just a time delta, because otherwise a cheating client could try and move faster by sending a bigger delta).

Snapshots sent by the server to all clients contain the state of the world after applying all the player movement that happened since the previous snapshot and executing one frame of the game logic. If a packet containing one snapshot is lost on the network, it doesn’t really make sense to send it again, instead the server will just send the next, more current snapshot, and the client will have to deal with the information deficit by extrapolating the positions of game objects, instead of interpolating. Does that mean that the client doesn’t need to tell the server which snapshot it received last because the server simply doesn’t care? On the contrary, because of a clever delta compression scheme that the server uses when sending the snapshots, it does in fact care a lot. Even though the server won’t be retransmitting any snapshot, it uses this information to only send the difference (delta) between the last snapshot confirmed by the client, and the current one. That way, even with possible packet loss, each individual packet is smaller in size, but is still guaranteed to contain enough information for the client to reconstruct the current snapshot.



Snapshots contain information about the player that the receiving client is responsible for (playerstate) and about all the other players and objects that the client could need (entities). The server doesn’ t necessarily send all the entities, only the ones that the player could see or interact with, based on where on the map they are currently located. The playerstate and the entitystate are structures with a predefined set of fields of certain types (floating point or integer) and sizes in bits.

Here’ s how the delta compression of this data works. The current snapshot is compressed against an older snapshot, not necessarily the previous one. If the network latency between the server and the client is non-negligible, it will usually be a snapshot from a few packets back, because that will be the newest one that the server knows for sure that the client has received (because it has confirmed it in one of its packets). This means that both the server and the client need to store a certain number or old snapshots.

The server goes through all the fields in the playerstate, and checks which fields have changed between the old snapshot and the current one. Then for each field that hasn’ t changed, it only needs to send a single bit, communicating that fact to the client. In fact, it doesn’ t even need to send one bit for each of the fields, it first sends the index of the last field that has changed and for all the fields that haven’ t changed after that last changed one, it doesn’ t need to send anything.

Similarly after the server decides what entities the client must know about, it compares this list with the list from the old snapshot. Some of the entities will remain unchanged, some will disappear and some will be present in the new snapshot when they weren’ t in the old one. Only in the case where the entity exists in the new snapshot, and is not identical to the one in the old, do the actual fields of the entity need to be sent. Even then, only the fields that have changed are sent, similarly to how it was done for the playerstate.

The server will at most send as many packets per second as the number of frames it processes (normally 20). The client has two ways of limiting that number: it can specify it wants a smaller number of snapshots per second, or it can specify the maximum number of bytes per second it wants to receive. The server will keep track of how much data it’ s sending to each client and will skip a snapshot if sending it would mean going over the limit.

The client will send a packet for every frame it renders, unless it would mean going over a configurable limit that can be set between 15 and 125 packets per second. When it needs to skip sending a packet, the user commands it would put in it will go in the next packet that is sent.

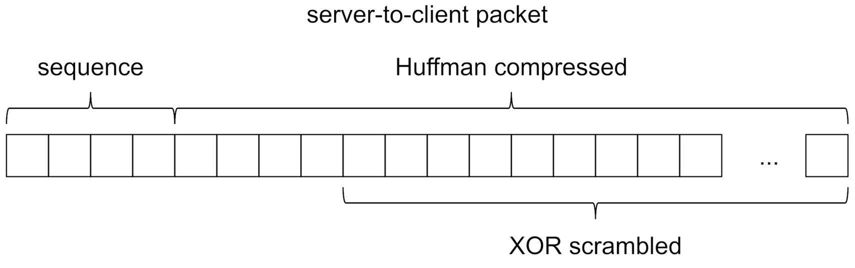
On-the-wire protocol

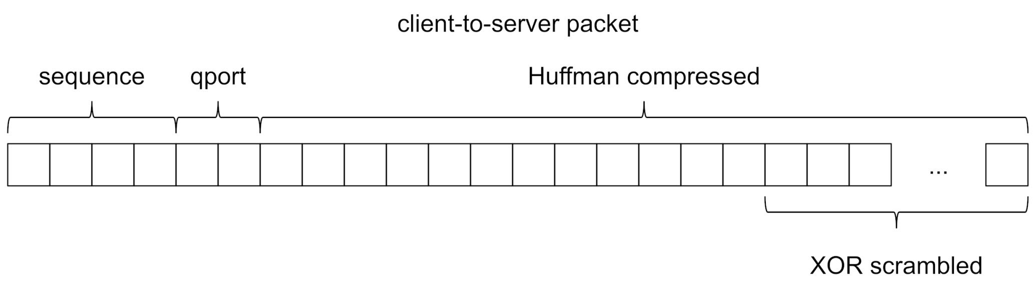
Now that we know what types of messages the clients are exchanging with the server and what information the messages contain, let’ s take a look at the actual bits that travel over the network.

Before the client is connected to the server, they exchange *connectionless* or *out-of-band* packets. These are used by the client to query the state of the server (to find out what map and game type is being played and what other players are connected) and as part of the handshake if the player decides to connect. Connectionless packets start with four 0xFF bytes. After that an ASCII command follows. The client sends commands like `getinfo`, `getstatus` or `getchallenge` and the server responds with `statusResponse`, `infoResponse`, `challengeResponse`, etc. One of these commands, `connect`, is special because it’ s followed by a key-value map called *userinfo*, that’ s not sent in plain ASCII, but instead it’ s Huffman-compressed.

After the client connects to the server, the rest of the exchanged messages follow a format that’ s a little more complex. The first four bytes of the message are the message sequence (in little-endian order). Then the client-to-server messages also have a two-byte (again, little-endian) value called *qport*. This value is used by the server in place of the UDP source port to work around the behavior of certain NAT routers that sometimes change the port mid-connection. After that everything in the packets is Huffman-compressed.

Additionally, to make life harder for cheaters, most of the packet is obfuscated by performing a XOR operation on it. This can’ t really be considered encryption, as the information that the XOR key is derived from is transmitted in plaintext beforehand, but it certainly makes it harder to understand the protocol without access to the engine’ s source code. For client-to-server packets, the XOR key is derived from the first three 32-bit values transmitted in the packet under consideration, the challenge that was part of the initial handshake, and also the text of the last confirmed server command. For server-to-client packets, the XOR key is derived from the message sequence, that same challenge and also the text of the last confirmed client command. Since the information sent at the beginning of the packets is needed to decode the rest of the packet, the first raw 12 bytes in client-to-server packets and the first raw 4 bytes in server-to-client packets (counting from the beginning of the Huffman-compressed region) are not XOR-scrambled. This is not sufficient in the general case, as the Huffman-compressed representation of a 12-byte long message can be longer than 12 bytes. So it’ s possible that the values needed to derive the XOR key will be XOR-scrambled themselves. It doesn’ t happen in practice. We can also see that to be able to decode the messages, one has to observe the entire connection from the beginning, to have the challenge and the client and server commands.





Huffman compression

Quake 3’ s network protocol uses an adaptive Huffman compression algorithm, as described in Khalid Sayood’ s classic book, *Introduction to Data Compression* (and many other places). But the only part where the algorithm actually operates in its adaptive mode is the `connect` command sent by the client during the initial handshake. After the “connect ” string, the rest of the packet contains the userinfo structure, which is transmitted using adaptive Huffman compression, starting with an empty tree. When a symbol that hasn’ t been transmitted before needs to be sent, it is sent (after the code for the *not-yet-transmitted* node) least-significant bit first.

For all the other packets sent as part of an established connection, everything after the first four bytes (for server-to-client packets) or first six bytes (for client-to-server packets) is transmitted using the same Huffman algorithm, but the tree remains fixed and is not updated after the transmission of each symbol. The tree used is created using a hardcoded frequency table, which presumably corresponds to a sample of network traffic captured during the development of the game.

The Huffman algorithm is used to transmit symbols with a length of 8 bits. Whenever the game needs to send a value of a size that is not an even multiple of 8 bits, it first sends the additional uneven bits as-is (uncompressed), starting with the least-significant ones. Then it sends the rest of the value, Huffman-compressed in 8-bit chunks.

Server-to-client packet

This section and the next describe the actual on-the-wire protocol using a hopefully self-explanatory pseudocode. Together with an understanding of the Huffman compression and XOR obfuscation used, and also the definitions of fields in entity and playerstate structures, it should be enough to implement the protocol.

```
sequence (32 bits)
if (sequence=0xFFFFFFFF) {
  command (ASCII, till the end of the packet)
} else {
  // rest of the packet is Huffman-compressed
  // it is also XOR-scrambled, except for the first 4 raw bytes
  // of the Huffman-compressed region
  reliable_acknowledge (32 bits)
  repeat {
    svc_op (8 bits)
    if (svc_op = 2 (svc_gamestate)) {
      last_client_command (32 bits) // that the server has received
      repeat {
        gamestate_op (8 bits)
        if (gamestate_op = 3 (svc_configstring)) {
          configstring_index (16 bits)
          configstring (null-terminated ASCII)
        }
        if (gamestate_op = 4 (svc_baseline)) {
          entity_number (10 bits)
          update_or_delete (1 bit) (always 0)
          entity_changed (1 bit)
          if (entity_changed = 1) {
            field_count (8 bits)
            for i in (0 .. field_count-1) {
              field_changed (1 bit)
              if (field_changed = 1) {
                // we need to know the entity field definitions
                if (field i is of type float in the definition) {
                  float_is_not_zero (1 bit)
                  if (float_is_not_zero = 1) {
                    int_or_float (1 bit)
                    if (int_or_float = 0) {
                      float_as_int (13 bits)
                    }
                    if (int_or_float = 1) {
                      float_as_float (32 bits)
                    }
                  } else {
                    // field value is 0
                  }
                } else {
                  int_is_not_zero (1 bit)
                  if (int_is_not_zero = 1) {
                    int_value (number of bits as in field i definition)
                  } else {
                    // field value is 0
                  }
                }
              }
            }
          }
        }
      } until (gamestate_op = 8 (svc_EOF))
      client_number (32 bits)
      checksum_feed (32 bits)
    }
    if (svc_op = 5 (svc_serverCommand)) {
      command_sequence (32 bits)
      command (null-terminated ASCII)
    }
    if (svc_op = 6 (svc_download)) {
      block (16 bits)
      if (block = 0) {
        download_size (32 bits)
      }
      size (16 bits)
      if (size > 0) {
        data (size*8 bits)
      }
    }
    if (svc_op = 7 (svc_snapshot)) {
      server_time (32 bits)
      delta_num (8 bits) // this snapshot is delta-compressed
                         // from delta_num snapshots ago
      snap_flags (8 bits)
      areamask_length (8 bits)
      areamask (areamask_length*8 bits)
      // playerstate:
      field_count (8 bits)
      for i in (0 .. field_count) {
        field_changed (1 bit)
        if (field_changed = 1) {
          // we need to know the playerstate field definitions
          if (field i is of type float in the definition) {
            int_or_float (1 bit)
            if (int_or_float = 0) {
              float_as_int (13 bits)
            }
            if (int_or_float = 1) {
              float_as_float (32 bits)
            }
          } else {
            int_value (number of bits as in field i definition)
          }
        }
      }
      // arrays:
      arrays_changed (1 bit)
      if (arrays_changed = 1) {
        stats_changed (1 bit)
        if (stats_changed = 1) {
          stats_bits (16 bits)
          for i in (0..15) {
            if (bit i is set in stats_bits) {
              stats_bit_i (16 bits)
            }
          }
        }
        persistant_changed (1 bit)
        if (persistant_changed = 1) {
          persistant_bits (16 bits)
          for i in (0..15) {
            if (bit i is set in persistant_bits) {
              persistant_bit_i (16 bits)
            }
          }
        }
        ammo_changed (1 bit)
        if (ammo_changed = 1) {
          ammo_bits (16 bits)
          for i in (0..15) {
            if (bit i is set in ammo_bits) {
              ammo_bit_i (16 bits)
            }
          }
        }
        powerups_changed (1 bit)
        if (powerups_changed = 1) {
          powerups_bits (16 bits)
          for i in (0..15) {
            if (bit i is set in powerups_bits) {
              powerups_bit_i (32 bits)
            }
          }
        }
      }
      // entities:
      repeat {
        entity_number (10 bits)
        if (entity_number != 1023) {
          update_or_delete (1 bit)
          if (update_or_delete = 0) {
            entity_changed (1 bit)
            if (entity_changed = 1) {
              field_count (8 bits)
              for i in (0 .. field_count-1) {
                field_changed (1 bit)
                if (field_changed = 1) {
                  // we need to know the entity field definitions
                  if (field i is of type float in the definition) {
                    float_is_not_zero (1 bit)
                    if (float_is_not_zero = 1) {
                      int_or_float (1 bit)
                      if (int_or_float = 0) {
                        float_as_int (13 bits)
                      }
                      if (int_or_float = 1) {
                        float_as_float (32 bits)
                      }
                    } else {
                      // field value is 0
                    }
                  } else {
                    int_is_not_zero (1 bit)
                    if (int_is_not_zero = 1) {
                      int_value (number of bits as in field i definition)
                    } else {
                      // field value is 0
```

```
                  }
                }
              }
            }
          }
        }
        if (update_or_delete = 1) {
          // the entity is not present in the new snapshot
        }
      }
    } until (entity_number = 1023)
  }
  } until (svc_op = 8 (svc_EOF))
}
```

Client-to-server packet

```
sequence (32 bits)
if (sequence=0xFFFFFFFF) {
  command (ASCII, till the end of the packet)
  // as a special case, if the command starts with “connect ”, what follows
  // is adaptive Huffman-compressed
} else {
  qport (16 bits)
  // from this point on, the packet is Huffman-compressed
  // it is also XOR-scrambled, except for the first 12 raw bytes
  // of the Huffman-compressed region
  server_id (32 bits)
  server_message_sequence (32 bits) // last message the client has received
  server_command_sequence (32 bits) // last command the client has received
  repeat {
    clc_op (8 bits)
    if (clc_op = 4 (clc_clientCommand)) {
      command_sequence (32 bits)
      command (null-terminated ASCII)
    }
    if (clc_op = 2 (clc_move) or clc_op = 3 (clc_moveNoDelta)) {
      command_count (8 bits)
      repeat (command_count times) {
        server_time_relative (1 bit)
        if (server_time_relative = 0) {
          server_time (32 bits)
        }
        if (server_time_relative = 1) {
          server_time_delta (8 bits)
          // server_time = previous_server_time + server_time_delta
        }
        command_changed (1 bit)
        if (command_changed = 1) {
          // the values of the following fields are XOR-scrambled
          angles0_changed (1 bit)
          if (angles0_changed = 1) {
            angles0 (16 bits)
          }
          angles1_changed (1 bit)
          if (angles1_changed = 1) {
            angles1 (16 bits)
          }
          angles2_changed (1 bit)
          if (angles2_changed = 1) {
            angles2 (16 bits)
          }
          forwardmove_changed (1 bit)
          if (forwardmove_changed = 1) {
            forwardmove (8 bits)
          }
          rightmove_changed (1 bit)
          if (rightmove_changed = 1) {
            rightmove (8 bits)
          }
          upmove_changed (1 bit)
          if (upmove_changed = 1) {
            upmove (8 bits)
          }
          buttons_changed (1 bit)
          if (buttons_changed = 1) {
            buttons (16 bits)
          }
          weapon_changed (1 bit)
          if (weapon_changed = 1) {
            weapon (8 bits)
          }
        }
      }
    }
  } until (clc_op = 5 (clc_EOF))
}
```

Fragments

When Quake 3 needs to send a packet that's 1300 bytes long or longer, it divides it into parts called fragments that are sent separately and then reassembled on the other side before further processing. In practice this only ever happens when sending the *gamestate* (starting positions of all objects and some configuration strings) after the client connects to the server or when a new map is loaded. The format of a fragmented packet is as follows:

```
sequence (32 bits) // sequence number of the original packet with the
                   // most-significant bit set to indicate fragmentation
qport (16 bits) // only if client-to-server packet
fragment_offset (16 bits) // where in the original packet this fragment fits
fragment_length (16 bits)
data (fragment_length*8 bits)
```

Fragment length of 1300 means that more fragments are coming so if the original packet's length was an even multiple of 1300, then at the end a zero-length fragment is sent so that the other side knows that this is the end of the packet.

Initial handshake

When the client first connects to the server, they exchange certain messages before the connection is considered to be established. There's also a third party, the authorization server, maintained by the game's creators. Its role is to verify that the player is using a genuine copy of the game by checking the CD key provided by the client. So at the same time that the client sends a `getchallenge` message to the server, it also sends a `getKeyAuthorize` message to the central authorization server, with the unique CD key. When the (game) server receives the `getchallenge` message from the client, it sends a `getIpAuthorize` message to the authorization server, asking if the client should be allowed to play. If it receives a positive response from the authorization server or it doesn't receive any answer within a certain time limit, it sends a `challengeResponse` message to the client, containing a challenge number. The client puts that number, along with certain other information like the protocol version, in the next message it sends, `connect`. Server then responds with a `connectResponse` message and the connection is established. Up until this point, all the packets exchanged were connectionless, meaning they had 0xFF in their first four bytes and weren't XOR-scrambled nor Huffman compressed (except for the userinfo part of the `connect` packet). The following packets start with their sequence numbers and follow the regular packet structure.

XOR keys

For server-to-client packets that are part of an ongoing connection, the XOR keys used for the obfuscation are derived from the challenge value, exchanged at the beginning of the connection, the sequence number of this particular packet, and the last acknowledged client command text (its sequence number is the first thing sent in each packet in the Huffman-compressed part).

The exact value for the *i*-th byte (counting from the beginning of the XOR-scrambled region and starting at zero) is calculated as follows:

```
key = challenge xor sequence xor (last_command[i mod command_length] * (1+(i mod 2)))
```

Except if the n-th character in the command text is is “%” or is out of 7-bit ASCII range, the value of “.” is used instead.

For client-to-server packets, the XOR keys are derived from the challenge, the `server_id` value that is sent as the first thing in each message, last confirmed server packet sequence number, sent as the next thing, and the text of the last acknowledged server command, the exact value for the *i*-th byte calculated as follows:

```
key = challenge xor server_id xor server_message_sequence xor (last_command[i mod
command_length] * (1+(i mod 2)))
```

The same exceptions for the message text apply.

In addition to this treatment, the values sent by the client as part of user commands are obfuscated again, using the same XOR operation, but a different key (this time the operation is done before Huffman compression when written by the client and after decompression when read by the server). The key used is derived from the `checksum_feed` value that is part of gamestate sent by the server, last confirmed server packet sequence number, a hash of the last confirmed server command text, and the `server_time` value that is part of each user command.

```
key = checksum_feed xor server_message_sequence xor last_command_hash xor server_time
```

With the hash of the last command defined as follows. First a 32-bit sum is calculated:

```
partial_hash = sum for i=0..n-1 (last_command[i] * (119+i))
```

where `n = min(last_command_length, 32)`

And then the final hash:

```
hash = partial_hash xor (partial_hash shr 10) xor (partial_hash shr 20)
```

Console commands and variables

One of the many novel aspects of the Quake series of games was the introduction of the console. The console, activated by pressing the tilde (~) key, is a command-line interface to the game’s internals, similar to the command prompt present in most modern desktop operating systems. In addition to settings and functions accessible from the normal game UI, it allows extensive customization of the game and exposes some debug functions, many of which apply to the network component.

Like many Unix shells, Quake 3’s console has tab-completion, meaning that if we type the beginning of a command and press the Tab key, it will either complete the command or show all the commands that begin with this prefix, if there’s more than one.

In addition to executing commands, one can also set console variables (or *cvars* for short). The cmdlist command shows a list of all commands, while the cvarlist command shows a list of all variables. Below is a selection of console variables that allow us to modify some of the behavior discussed before.

`cl_packetdup`
how many times a user command is repeated in subsequent packets (meaningful values: 0-5, default: 1)

`cl_maxpackets`
how many packets per second the client will send at most (meaningful values: 15-125, default: 30)

`com_maxfps`
how many frames per second the client will render at most (default: 85)

`sv_fps`
how many frames per second the server will process (default: 20)

`vm_game`, `vm_cgame`, `vm_ui`
what type of code the VMs will run (0=native, 1=interpreted bytecode, 2=translated bytecode, default: 2)

`snaps`
how many snapshots per second the client wants to receive (default: 20)

`rate`
how many bytes per second the client wants to receive at most (default: 3000)

`sv_maxRate`
server-side limit for the client rate (meaningful values: >=1000 or 0, default: 0, meaning no limit)

`cl_timeNudge`
offsets the client’ s view of time when interpolating/extrapolating between snapshots, positive values mean bigger delay, but lower chance of needing to extrapolate, negative values mean the opposite (meaningful values: -30 to 30, default: 0)

`showpackets`
if nonzero, prints information about sent and received packets - their sizes and sequence numbers (default: 0)

`cg_lagometer`
if nonzero, enables the lagometer (default: 1)

`cg_nopredict`
disables client prediction (default: 0)

`cg_predictItems`
enables client prediction of item pickups (default: 1)

`cl_shownet`
if nonzero, prints debug information about delta-compressed entities and playerstate received in a snapshot (meaningful values: -2, -1, 0, 1, 2, 3, 4, default: 0)

`journal`
0: normal operation, 1: write all events to journal.dat file, 2: read events from journal.dat file, disregarding normal input (default: 0; needs to be set from the command line, e.g. `+set journal 1`)

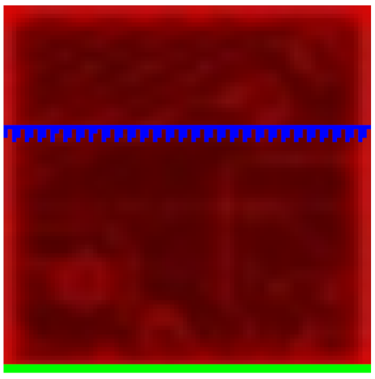
Lagometer

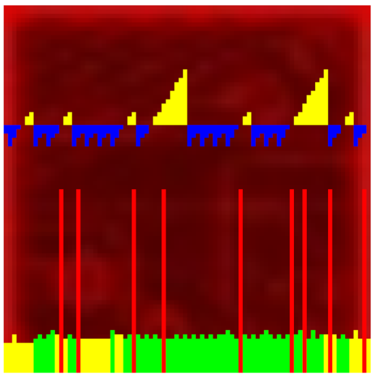
The network connection is so important for gameplay in Quake 3 that the game includes a diagnostic interface, the *lagometer*, that is enabled by default. Shown in the lower right corner of the screen, it consists of two graphs. They’ re located one above the other, but their X axes are separate and usually they will move at different speeds.

The top graph moves one pixel for every frame rendered by the client. Blue color corresponds to frames interpolated between two snapshots, while yellow means that the frame had to be extrapolated. On the Y axis is the difference between the client frame time and the time of the snapshot that the game is extrapolating from or interpolating towards.

The bottom graphs moves one pixel for every snapshot received (or not received) from the server. Green and yellow correspond to a properly received snapshot, yellow additionally meaning that a previous snapshot was intentionally not sent by the server because of the bandwidth limit requested by the client (the server communicates this by setting a bit in snap_flags). The Y axis is the ping, or the round-trip time between the client and the server. A vertical red bar means that a snapshot was lost on the network (the client knows this because the packets have sequence numbers).

Shown below are two example lagometers. The one on the left would be displayed on a good connection with 20 ms ping, no packet loss and the client rate set to 25000. The one one the right corresponds to a poorer connection with 150 ms ping, 10% packet loss and client rate set to 3000.





Closing thoughts

Hopefully this gives at least a basic understanding of Quake 3’ s network architecture and encourages some experimentation and code analysis. There are many interesting aspects of the engine that we haven’ t discussed here, such as the renderer or the bot system. Its source code is definitely worth a look.