Quake II Network Protocol Specs

Tim Ferguson. timf@dgs.monash.edu.au v0.03, 12/3/98

Contents

1 Introduction

The Quake II network protocol is very similar to the QuakeWorld protocol. As in the past, the server to client network packets have the same format as the demo files (Quake II's dm2 files). This information can be taken from 'The unofficial DM2 format description' by Uwe Girlich: `http://www.physik.uni-leipzig.de/\~{}girlich/games/`

I have attempted to decipher the client to server packets. This document is in a similar format to the QuakeWorld network protocol specs of Olivier Montanuy: `http://www.ens.fr/\~{}cridlig/bot/`

In future versions of this document, I will attempt do decipher more of the protocol and include a connection set up sequence with pseudo code (again, this is very similar to (and HEAPS simpler than) the QW set up sequence). I may also release source code for establishing a connection.

One thing to note is that client to server player update packets contain a checksum. The purpose for this checksum is to discourage people creating proxy bots as stated in one of John Carmack's plan updates:

> '* checksum client move messages to prevent proxy bots'

This would make all forms of bot programming impossible until someone deciphers the checksum routine (which would defeat the purpose of putting the checksum in the first place). The checksum routine for v3.10 and v3.13 has been hacked and is included in this spec. Currently, it appears that for each major release, Id Software will need to change the checksum routine to combat the current hacked version. Also, the checksum not only prevents the creation of proxy bots, but also the creation of autonomous bots.

People who have directly and indirectly assisted in the decoding of this protocol are:

- Uwe Girlich: Uwe.Girlich@itp.uni-leipzig.de
- Jorgen Karlsson: zappa@mbox305.swipnet.se
- Kekoa Proudfoot: kekoa@graphics.stanford.edu

This revision of the document has been a long time coming. I must apologise, but I have been too busy working on my PhD to get anything done related to Quake.

Legal note:

Quake II, QuakeWorld are trademarks of id software inc. All technical information is copyright (c) 1997 id software. The document is not a publication of id software, and they probably wont answer questions related to it.

The document is copyright (c) 1998 Tim Ferguson.

Permission to use, copy and distribute unedited copies of this whole document is hereby granted.... etc... If you have anything to add to this document, please contact me.

2 Special Packets

These are the same as the QuakeWorld specs. They are text strings containing commands for the client and server:

Header	Content
0xFFFFFFFF	text string \0

These packets are connection less in that they do not require a connection to be established with the server before being sent. When a client sends a special packet, typically the server returns a response to the sending client. These packets are used to interrogate and maintain the state of a server (status, rcon), and initiate a connection (getchallenge, connect).

3 Game Packets

3.1 Client to Server Packets

These game packets are sent from the client to the server:

Sequence number 1	Sequence number 2	Client ID	Message contents
4 bytes	4 bytes	2 bytes	byte stream
Id of this packet	Last packet recvd	ID number	sequence of commands

The sequence numbers are similar to the QuakeWorld specs. When transmitting data using the UDP/IP network protocol, packets may be lost (routers overflow) or arrive in a different order (packets take different routes) from which they were sent. Therefore, each game packet contains two sequence numbers:

- Sequence number 1 is the identification of the message itself, as set by the transmitter. The sequence numbers start at one and increment by one with each packet sent. This allows the receiver to detect loss and reorder packers.
- Sequence number 2 is the identification of the last message received and can be used to determine which packets were lost.

As Quake II is a real time playing game, any form of error correction has to operate on that pretext. From what can be ascertained from playing with the network protocol, Quake II uses a form of update error correction using frame numbers (similar to sequence numbers), and which is independent of the above sequence numbers.

As with the QuakeWorld protocol, bit 31 (0x80000000) of the sequence number (the 'hidden' bit) has a special meaning. It is used to mark packets with reliable data, allowing the client to resend the information if it is lost in transit. The reliable data bit is set in sequence number 1 when sending a reliable message in a packet. When a packet is received with the reliable bit set in sequence number 1, the reliable bit in sequence number 2 of the outgoing packets must be toggled. The sender of a reliable packet knows if the packet reached its destination when it sees the reliable bit toggled in a packet with a sequence number greater than the reliable packet's sequence number. Only one reliable packet can be in transit at any time.

A client ID or port number is sent when a client connects to a server. The number appears to be chosen at random during startup, and is sent with every client to server packet (it is used to disambiguate multiple clients from the same IP address due to a socket port router mapping problem John Carmack mentioned in a .plan update).

0x00 Bad

```
{
   command = ReadByte();       // 0x02
}
```

Internal message indicating misparsed packets. Not an actual game message.

0x01 Nop

```
{
   command = ReadByte();       // 0x01
}
```

No operation and contains no data.

0x02 Player Update

The data in this message is three compressed versions of the usercmd_t data type defined in the gamex86.dll source code: 'q_shared.h'.

```
{
struct {
   float  pitch, yaw, roll;
   short  forward, side, up;
   char   msec, light_level, buttons, impulse;
} ucmd[3], *u;
long   client_frame;
char   mask;
int    chk_sum, i;
```

```
   ucmd[0] = 0;   // Delta code 1st user command against 0

   command = ReadByte();       // 0x02
   chk_sum = ReadByte();       // packet checksum.
   client_frame = ReadLong();  // Client frame number.
   for(i = 0; i < 3; i++)
   {
      if(i != 0)     // Delta code against previous user command
         ucmd[i] = ucmd[i-1];

      u = ucmd + i;
      mask = ReadByte();       // delta compression mask
      if(mask & 0x01) u->pitch = GetAngle16();  // pitch orientation (up/down)
      if(mask & 0x02) u->yaw = GetAngle16();    // yaw orientation (left/right)
      if(mask & 0x04) u->roll = GetAngle16();   // roll orientation
      if(mask & 0x08) u->forward = GetShort();  // speed in forward direction
      if(mask & 0x10) u->side = GetShort();     // speed in right direction
      if(mask & 0x20) u->up = GetShort();       // speed in upward direction
      if(mask & 0x40) u->buttons = GetByte();   // see below
      if(mask & 0x80) u->impulse = GetByte();   // impulse command code

      u->msec = ReadByte();          // the time taken to render a frame
      u->light_level = ReadByte();   // light level the player is standing at
   }
}
```

Bits for the buttons variable are also found in 'q_shared.h' as follows:

- Bit 1: BUTTON_ATTACK - Set when fire button is pressed
- Bit 2: BUTTON_USE - Not used as far as I can tell.
- Bit 8: BUTTON_ANY - Set when any key is pressed (used to end intermission).

If the packet checksum is incorrect, the packet is completely ignored. See the appendix A (Player Update Checksum) on how to calculate this checksum. The checksum routine changed in the point release and current release (v3.14).

The frame number is the last frame received from the server, sent using the 0x14 Frame message. It is used when decoding the packet entities delta compression by preventing information update loss through lost packets. This is covered in more detail later.

This message is made up of three user commands. The three commands are the last three player updates in the order of oldest to newest. They are delta coded against each other, with the first packet delta coded against a all zero command. The reason for sending the last three updates in every packet is so that if either or both of the last two packets are lost, the server has enough information to perform prediction correctly. The msec value is the time between each user command, and is used for accurately calculating the future prediction points. Light level is typically used for AI when playing against monsters. The idea being that if the light level is low, a monster cannot easily see you. Perhaps this is used in cooperative games? It could be easily worked into a client side bot, making it a more realistic player.

0x03 User Info

```
{
char    *text;
```

```
    command = ReadByte();       // 0x03
    text = ReadString();        // text string of the user info.
}
```

A user information string.

0x04 Console Command

```
{
char    *text;

    command = ReadByte();       // 0x04
    text = ReadString();        // text string of the command.
}
```

A command string decoded by the console. This can be a series of semicolon separated commands. Examples include new, configstrings, baselines, begin, help, inven, use, say and disconnect.

3.2 Server to Client Packets

These game packets are sent from the server to the client:

```
+------------------+------------------+----------------------+
| Sequence number 1 | Sequence number 2 | Message contents     |
|    4 bytes        |   4 bytes         |   byte stream        |
+------------------+------------------+----------------------+
| Id of this packet | Last packet recvd | sequence of commands |
+------------------+------------------+----------------------+
```

These packets used the same format as the client to server packets, minus the client ID number. The sequence number and reliable data bit also operate in the same way.

0x00 Bad

Internal message indicating misparsed packets. Not an actual game message.

0x01 MuzzleFlash

Not yet assessed.

0x02 MuzzlFlash2

Not yet assessed.

0x03 Temp_Entity

Has been updated with more entity types since Quake II v3.10 of the DM2 specs.

0x04 Layout

Same as DM2 specs. The server sends this when a client send a console command of 'help'. The F1 key is the default key assigned to this command ("cmd help" bound to F1).

0x05 Inventory

Same as DM2 specs. The server sends this when a client send a console command of 'inven'. The tab key is the default key assigned to this command ("cmd inven" bound to tab).

0x06 Nop

Not yet assessed.

0x07 Disconnect

Same as DM2 specs.

0x08 Reconnect

Not yet assessed.

0x09 Sound

Not yet assessed.

0x0a Print

Same as DM2 specs.

0x0b StuffText

Same as DM2 specs.

0x0c ServerData

```
{
long   version, client_id;
char   uk_b1, *game_dir;
short  player_entity;

   command = ReadByte();        // 0x0c
   version = ReadLong();        // Version number as in DM2 specs
   client_id = ReadLong();      // Client ID number used in making connection
   uk_b1 = ReadByte();
   game_dir = ReadString();     // Game dir specified with '+set game'
   player_entity = ReadShort(); // players entity entry in PacketEntities.
   map_name = ReadString();     // Games map name
}
```

The client ID number is used in the game set up and is appended to the console commands: configstrings, baselines and begin.

0x0d ConfigString

Same as DM2 specs.

0x0e SpawnBaseLine

Same as DM2 specs.

0x0f CenterPrint

Not yet assessed.

0x10 Download

Currently, download is not supported by Quake II (despite several complaints).

0x11 PlayerInfo

Same as DM2 specs.

0x12 PacketEntities

Same as DM2 specs.

0x13 DeltaPacketEntities

Not yet assessed. I have never actually come across one of these packets. Do they exist? Do PacketEntities do their job?

0x14 Frame

Same as DM2 specs.

4 Game Updates Using Frames

Every one tenth of a second, a Quake II server updates the status of all entities in the game, which includes the position, orientation, velocity, etc. of all players, ammunition, weapons, etc. This information is then transmitted to all the connected clients which use interpolation (based on entity position and velocity) to render frames at the clients maximum video frame rate. The one tenth of a second update is known as one 'frame', and the server keeps track of which frame is being generated using a frame sequence counter. This counter is set to zero at the start of a level and is incremented every time a new frame is generated.

To reduce network bandwidth, the server only transmits changes to entities within a visible range using delta compression. For each server frame update, the server would first determine what entities the client can see, followed by transmitting changes to the visible entities based on the previous frame the client received (delta compression).

The server tells the client which frame it is currently sending (current frame: seq1), and which frame to delta compress against (delta reference frame: seq2) using the 0x14 Frame message. The client tells the server the last frame it received in its 0x02 Player Update message. The entity updates are transmitted using the 0x12 Packet Entities message. Also updated using the frame delta compression is the connected clients player information, using a 0x11 Player Info message. After a client has connected to a server, all server to client packets typically contain a frame message, player info message and packet entities message in that order. For a client to produce the current frame information specified by frame number seq1, it simply applies the deltas in the current packet entities and player info messages to the frame specified by the frame number seq2. The server knows which frame to delta compress against due to the client telling it in the player update message.

As stated earlier, Quake II is a real time playing game, and its error correction has to operate on that pretext. To achieve this, both the client and the server store the last N frame updates to allow for packet loss. If any of the packet entities messages are lost, the server will still transmit future entity changes based on the last server frame it knows the client received. If the difference between the clients reference frame number and the servers current frame number becomes too large (that is, greater than N), the server goes into a special mode where it transmits entity updates based on the base line values sent using the 0x0e Spawn Base Line messages during connection. The client signifies this no delta mode by setting bit 32 (0x80000000) of the frame number being set to the server. The no delta mode can be manually entered using the 'cl_nodelta 1' command in the Quake II console.

The server tells the client to delta code using the base line values by setting the reference frame number, seq2, to 0xffffffff (or -1). The server can also use this reference frame number if it is told to delta code against something it doesn't have.

The value of N for the client and server appear to be different. For the server N equals 12 meaning the server can delta code to a current frame number seq1 if the difference between it and the reference frame number seq2 is no greater than 12. For the client, N equals 16 meaning that if the difference between seq1 and seq2 is greater than 16, the client starts complaining by sending a sequence number of 0xffffffff. However the difference will never get this high as the server will start delta coding against base lines when the difference is 12 or greater.

I would like to thank Kekoa Proudfoot for his help in producing the above information by playing with the Quake II network protocol.

5 Conclusion

This document is at very early stages. If anyone has anything to add to it, please feel welcome to e-mail me. I am currently studying a PhD and don't have a lot of time to work on this, so help would be appreciated. There will be frequent updates of this document to try keeping it as up to date as possible.

A Player Update Checksum

Jorgen Karlsson was kind enough to hack the checksum routine and gives the following account of his effort:

"Before I found the md4.c file I was disassembling the Q2 executable one day in order to reverse engineer the checksum routines. I did a few C routines and copied some assembler routines directly from the disassembly and incorporated them into my own bot. I did not know what was going on but it worked, and I was in heaven :) Then later that day I was scanning through the Quake II game sources and found the md4.c file. It was used to calculate the checksum for the pak/bsp files. I looked at the source and found that some of the routines looked like some of mine reversed engineered C functions, well maybe... I compiled the md4.c with VC5.0 (optimised for speed) and checked with the disassembly, and a found, a perfect match. eureka!!"

To calculate the checksum for a player update message, you need the checksum source code, 'utils3/common/md4.c', provided up id software, and the dir_data[] table from the DM2 specs. The following two routines can then be used:

```
//
// MakeChecksum for v3.10
//
unsigned MakeChecksum_310(unsigned char *buf, int nbytes, int cl_seqnum)
{
unsigned char tbuf[64];

   if (nbytes > 62) nbytes = 62;
   memcpy(tbuf, buf, nbytes);
   tbuf[nbytes] = cl_seqnum;
   tbuf[nbytes+1] = cl_seqnum >> 8;
   return Com_BlockChecksum(tbuf, nbytes+2) & 0xFF;
}

extern float dir_data[];

//
// MakeChecksum for v3.13/v3.14
//
unsigned MakeChecksum_313(unsigned char *buf, int nbytes, int cl_seqnum)
```

```
{
int offset;
unsigned char *p, tbuf[64];

   offset = (cl_seqnum % 0x798);
   p = (unsigned char *)dir_data + offset;
   if(num_bytes > 60) num_bytes = 60;

   memcpy(tbuf, buf, num_bytes);
   tbuf[num_bytes] = p[0] ^ cl_seqnum;
   tbuf[num_bytes+1] = p[1];
   tbuf[num_bytes+2] = p[2] ^ (cl_seqnum >> 8);
   tbuf[num_bytes+3] = p[3];
   return Com_BlockChecksum(tbuf, num_bytes+4) & 0xFF;
}
```

Call the routine as follows:

```
checksum = MakeChecksum_31x(buf, nbytes, cl_seqnum);
```

where:

- buf - Points to first byte after after the checksum byte itself.
- nbytes - Is the number of bytes in the player update message minus 2 (not including the command byte 0x02 and the checksum itself).
- cl_seqnum - Is the sequence number received from the client (the first long int in the packet).

B Versions

0.01: 9th Jan 1998

- Initial draft of the specs.
- Released to get feedback from others working in the area.

0.02: 12th Jan 1998

- Spec converted to sgml.
- fixed errors and made additions given by Jorgen Karlsson.

0.03: 12th Mar 1998

- added v3.10 and v3.13 checksum routines given by Jorgen Karlsson.
- made additions given by Kekoa Proudfoot.
- extended some of the protocol description.