MARCH 9TH, 2009

QUAKE ENGINE CODE REVIEW : NETWORK (2/4)

QuakeWorld network architecture was considered a ground breaking innovation at the time. All network game successor used the same approach. Here are the details.

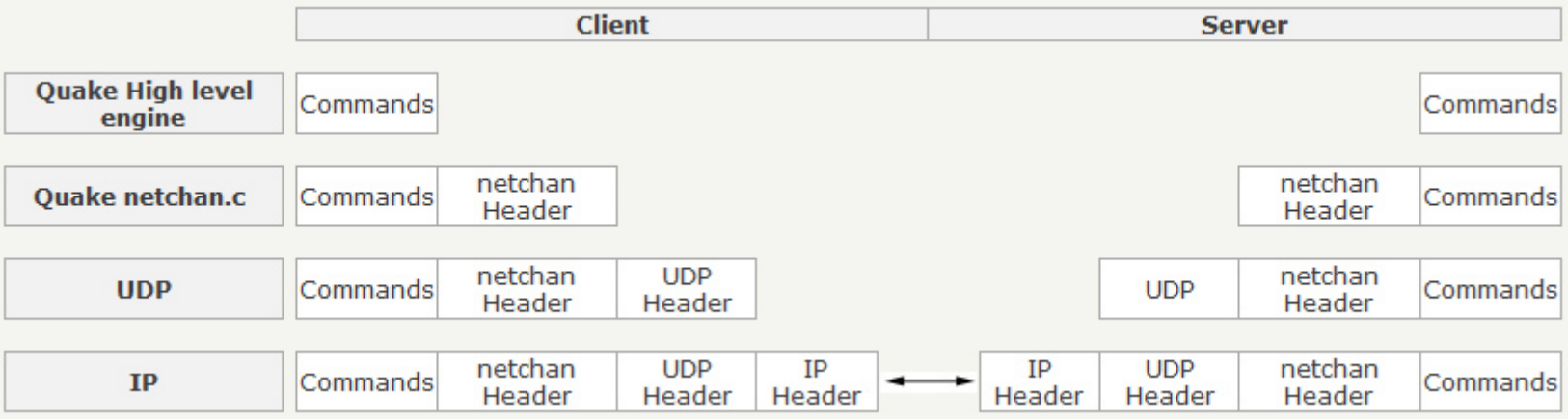
This article is in four parts :

Architecture section
Network section
Prediction section
Rendition section

Network stack

Quake's elementary unit of communication is the `command`. They are used to update a player position, orientation, health, damage, etc. TCP/IP features a lot of great functionalities that would be nice to have in a real time simulation (flow-control, reliability, packet sequencing) but could not be used for Quake World Engine (it was in the original Quake). In a FPS, information that is not received ASAP is not worth re-sending. So UDP/IP was selected and in order to implement reliability and packet sequencing, the network abstraction layer "`NetChannel`" was created.

From an OSI perspective, `NetChannel` sits very nicely on top of UDP :



So, to summarize: The engine deals mostly with `commands`. When it needs to send or receive, it delegates the task to methods `Netchan_Transmit` and `Netchan_Process` from `netchan.c` (those methods are the same on client and server).

NetChannel Header

Here is the structure of a NetChannel header:

Bit offset	Bits 0-15	16-31
0	Sequence	
32	ACK Sequence	
64	QPort	Commands
94	...	

- Sequence is an `int` initialized by the sender and incremented by one every times a packet is sent. The purpose of `sequence` is multiple but the most important is to provide the receiver with a way to recognized lost/duplicate/out of order UDP packets. The strongest bit of the int is not part of the sequence but a flag indicated that the payload (`commands`) contains reliable data (more about this later).
- ACK Sequence is also an `int`, it is equal to the last sequence number received. With this, the other end of the NetChannel can see if a packet was lost.
- QPort is here to circumvent a bug with NAT routers (Read more at the end of this page). The value is a random number set when the client starts up.
- Commands: Is the payload

Reliable messages

Unreliable commands are grouped in a UDP packet, marked with the last outgoing sequence number and sent ; it doesn't matter to the sender if they get lost.
Reliable commands are dealt with differently, the key is to understand that there can be only one reliable UDP packet unacknowledged between a sender and a receiver.

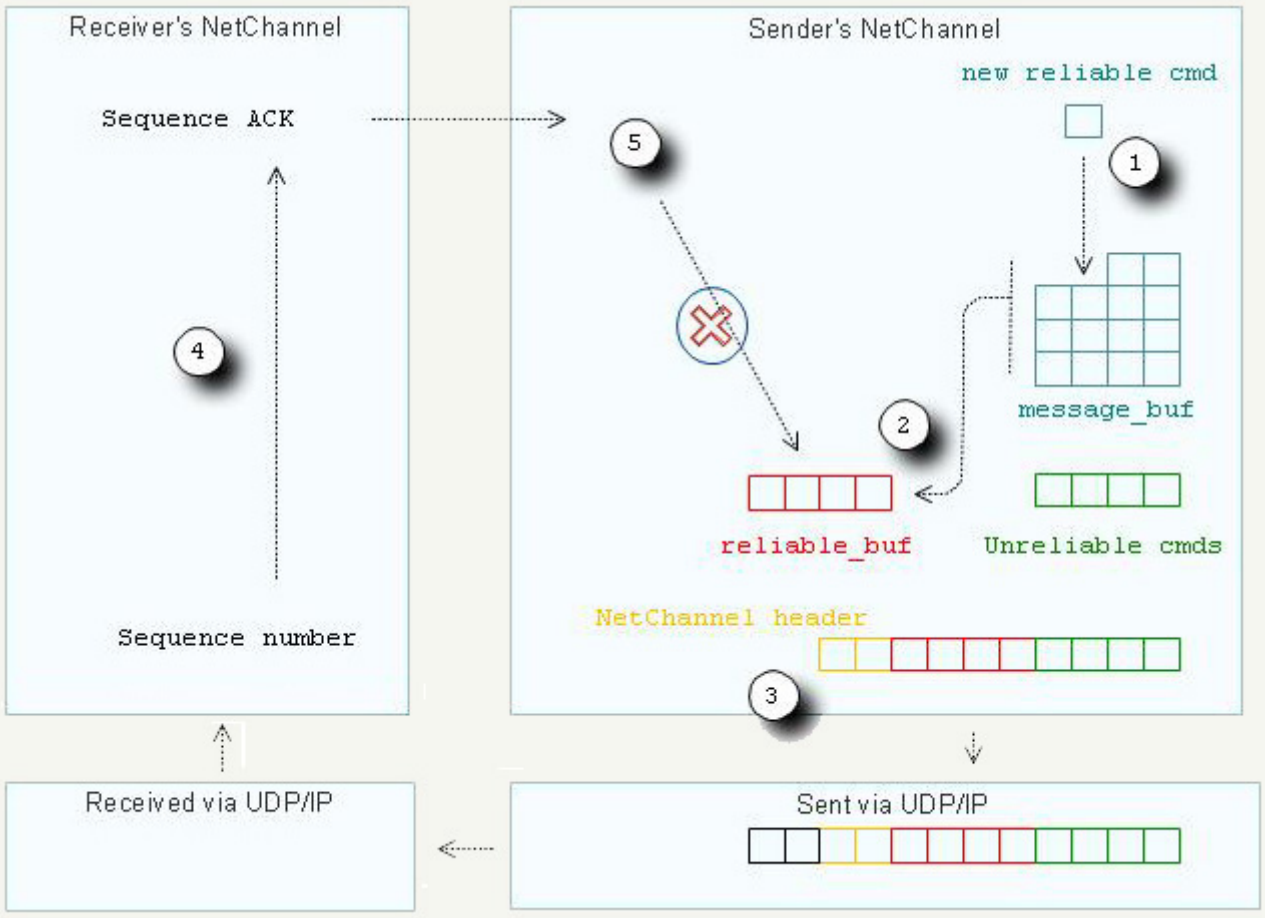
Every game loops, if a new reliable command is generated it is added to the `message_buf` array (piloted via the `message` variable) (**1**). The set of reliable commands is then moved from `message` to the `reliable_buf` array (**2**). This happens only if `reliable_buf` is empty (if it is not empty, this means that an other set of commands was sent before and has not yet been acknowledged).

The final UDP datagram is then created: NetChannel header is added (**3**) then `reliable_buf` content and unreliable commands of the moment, if there is enough space available.

On the receiving end, the UDP message is parsed, the incoming `sequence` number is transfered to the outgoing `sequence ACK` (**4**), (along with the bit flag indicating the packet contained reliable data).

On the next message received:

- If the reliable bit flag is set to true, the UDP packet made it to the receiver. NetChannel can clean `reliable_buf` (**5**) and is ready to send a new set of commands.
- If the reliable bit flag is set to false, the UDP did not made it to the receiver. NetChannel will try to send the content of `reliable_buf` again. New commands will pile up in `message_buf`, if this array overflows, the client is dropped.



Flow-Control

As far as I could read, there is flow-control on server side only ; A client sends its state updates as fast as it can.
The first flow control rule, active on the server is to send a datagram only if a datagram was received from the client. The second form of control flow is "choke", a parameter the client is able to set via the `rate` command in the console. This will make the server skip update messages, lowering the amount of data sent to the client.

Important commands

Commands have a type code stored in a `byte`, followed by the payload of the command. Probably the most important are the commands giving information about the state of the game (`frame_t`):

- `svc_packetentities` and `svc_deltapacketentities`: Update entities such as rocket trail, explosion, particles etc...
- `svc_playerinfo`: Send update about players position, last command and command duration in msec

More on the qport

Qport was added to the NetChannel header to fix a bug. Before the qport, Quake server identified a client by the combination (remote IP,remote UDP port). This worked fine most of the time but certain NAT router can change their schema of port translation (remote UDP port) sporadically. UDP port being unreliable, John Carmack explained in one of his plans that he decided to identify a client by (remote IP, Qport in NetChannel header). This fixed the confusion and also allowed the server to adjust the target UDP response port on the fly.

Latency calculation

Quake engine stores the 64 last sent commands (in a `frame_t` array: `frames`) along with the `senttime`, they are directly accessible via the sequence number used to transfer them (`outgoing_sequence`).

```
frame = &cl.frames[cls.netchan.outgoing_sequence & UPDATE_MASK];
frame->senttime = realtime;

//Send packet to server
```

Upon aknowledgment from the server, the time the command was sent is retrieved via `sequenceACK`. Latency is calculated as follows:

```
//Receive response from server

frame = &cl.frames[cls.netchan.incoming_acknowledged & UPDATE_MASK];
frame->receivedtime = realtime;
latency = frame->receivedtime - frame->senttime;
```

Some elegant things

Array index cycling

The network part of the engine stores the 64 last received UDP datagrams. The naive approach to cycle through the array would have been to use the modulo operator:

`arrayIndex = (oldArrayIndex+1) % 64 ;`

Instead the new value is calculated with an "AND" binary operation on an UPDATE_MASK, UPDATE_MASK being equal to 64-1.

`arrayIndex = (oldArrayIndex+1) & UPDATE_MASK;`

The real code is actually:

`frame_t *newpacket; newpacket = &frames[cls.netchan.incoming_sequence&UPDATE_MASK];`

Update: Here is a response I received from "Dietrich Epp", regarding modulo optimization:

```
There is a problem with the final section where using the modulo operator is called "naive".
Here is an example of the difference between the modulo and the "and" operator:

Create a file.c:
unsigned int modulo(unsigned int x) { return x % 64; }
unsigned int and(unsigned int x) { return x & 63; }

Run gcc -S file.c, and look at the output file.s.
You'll see that the functions are line for line identical -- even though optimization is turned off!
Same things go for "clever" things like doing << 5 instead of *32.
These changes make the code less readable with no benefit whatsoever,
so I've taken to considering the << 5 or & 63 versions to be "naive" and the *32 or %64 versions to be more intelligent.

--Dietrich

.globl modulo
    .type    modulo, @function
modulo:
    pushl    %ebp
    movl    %esp, %ebp
    movl    8(%ebp), %eax
    andl    $63, %eax
    popl    %ebp
    ret
    .size    modulo, .-modulo
.globl and
    .type    and, @function
and:
    pushl    %ebp
    movl    %esp, %ebp
    movl    8(%ebp), %eax
    andl    $63, %eax
    popl    %ebp
    ret
    .size    and, .-and
```

Return to main Quake Source Exploration page.