JUNE 30, 2012

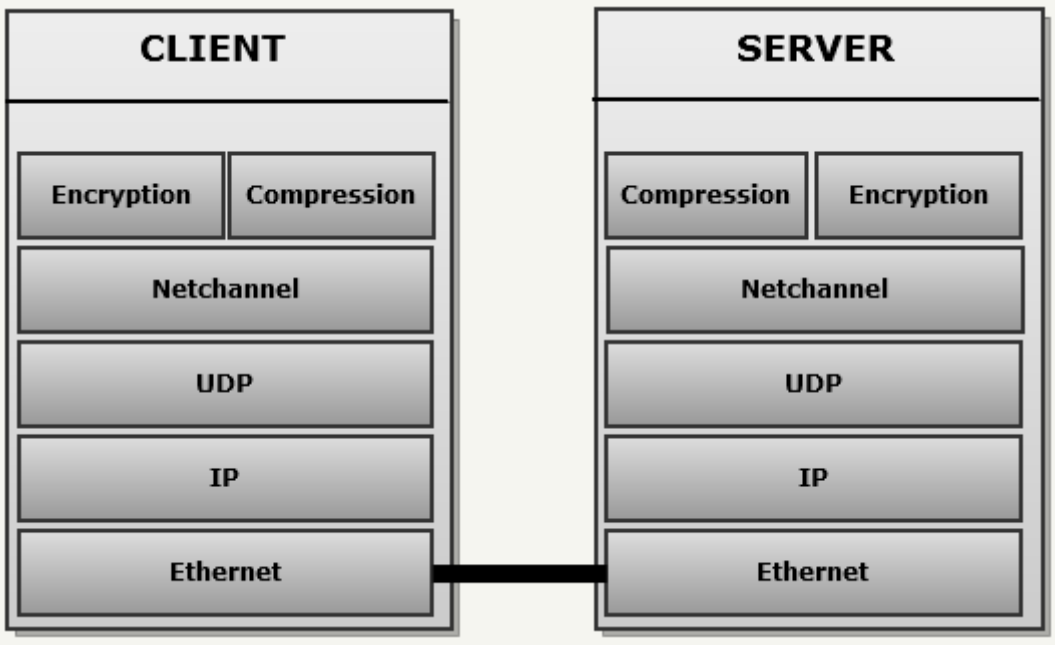
QUAKE 3 SOURCE CODE REVIEW: NETWORK MODEL (PART 3 OF 5) >>

The network model of Quake3 is with no doubt the most elegant part of the engine. At the lower level Quake III still abstract communications with the NetChannel module that first appeared in Quake World. The most important thing to understand is:

In a fast paced environment any information that is not received on first transmission is not worth sending again because it will be too old anyway.

As a result the engine relies essentially on UDP/IP: There is no trace of TCP/IP anywhere since the "Reliable transmission" aspect introduced intolerable latency. The network stack has been augmented with two mutually exclusive layers:

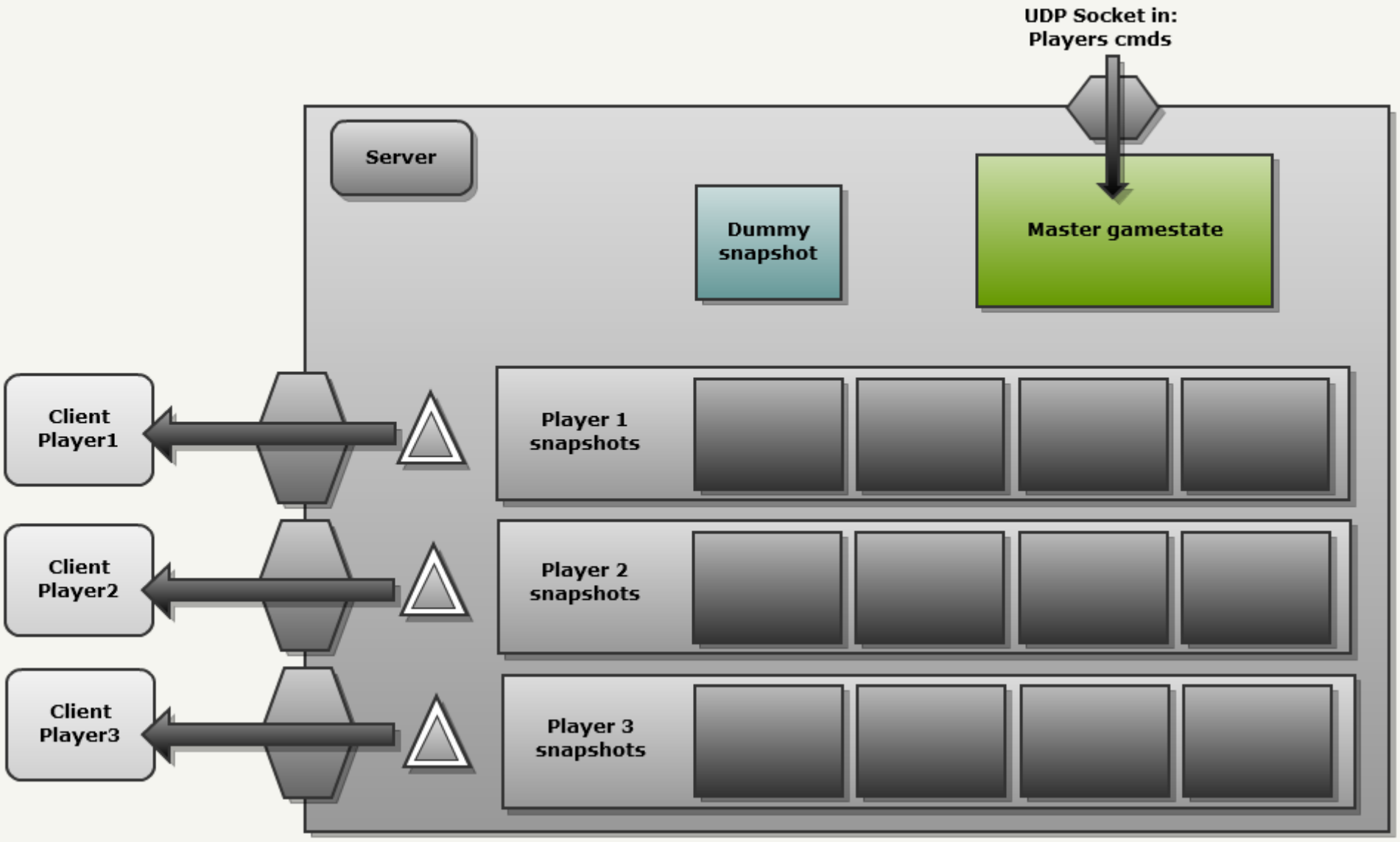
- Encryption using preshared key.
- Compression with pre-computed huffman key.



But where the design really shine is on the server side where an elegant system minimize the size of each UDP datagram while compensating for the unreliablity of UDP: An history of snapshots generate deltas packets via memory introspection.

Architecture

The Client side of the network model is fairly simple: Client sends commands to the Server each frame and receive update for the gamestate. The Server side is as bit more complex since it has to propagate the Master gamestate to each Client while accounting for lost UDP packets. This mechanism features three key elements:



- A **Master Gamestate** that is the universal true state of things. Clients send theirs commands on the Netchannel. They are transformed in event_t which will modifiy the state of the game when they arrive on the Server.
- For each Client the server keeps the **32 last gamestate** sent over the network in a cycling array: They are called snapshots. The array cycle with the famous binary mask trick I mentioned in Quake World Network (Some elegant things).
- The server also features **a "dummy" gamestate** with every single field set to zero. This is used to delta snapshots when there is no "previous state" available.

When the server decides to send an update to a client it uses each three elements in order to generate a message that is then carried over the NetChannel.

Trivia : To keep so many gamestate for each players consumes a lot of memory: 8 MB for 4 players according to my measurements.

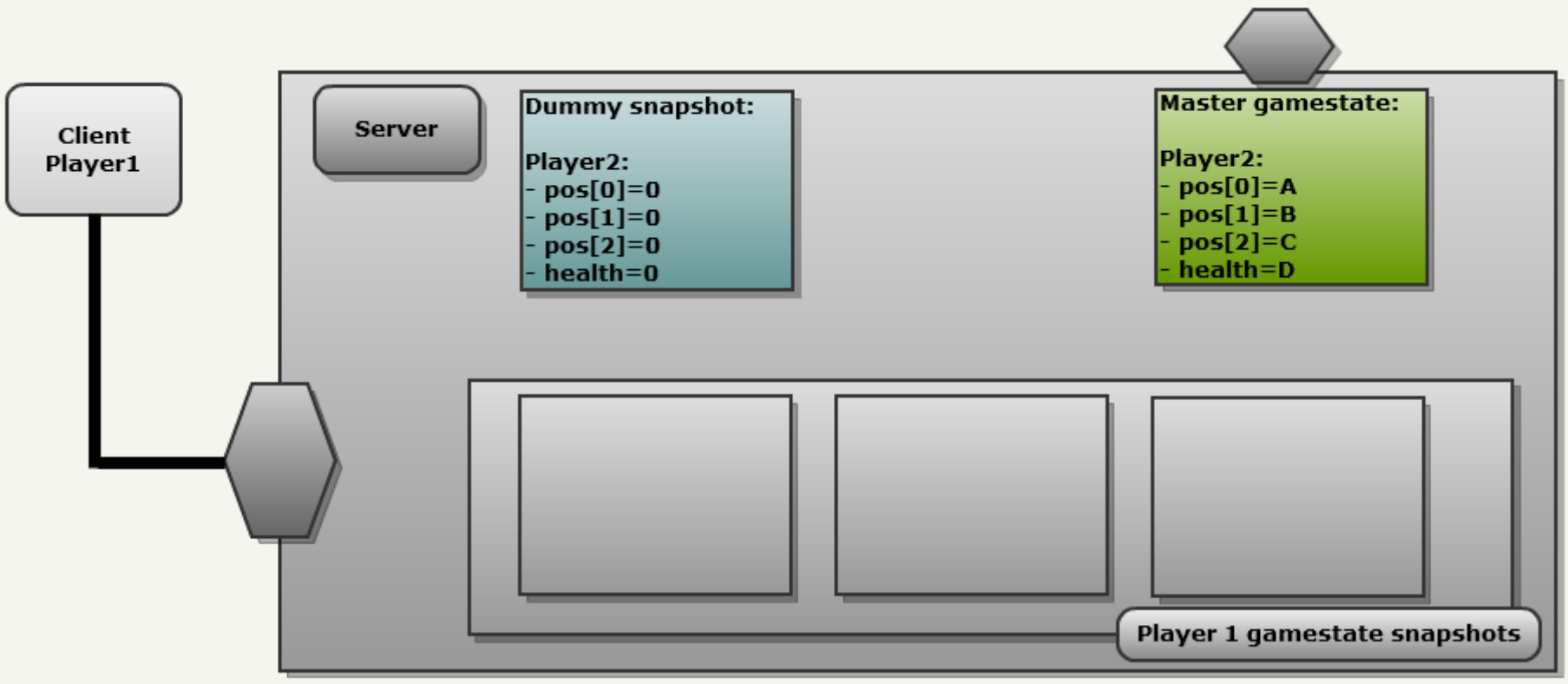
Snapshot systems

In order to understand the snapshop system, here is an example with the following conditions:

- The server is sending update to a Client1.
- The server is attempting to propagate the state of Client2 which has four 4 fields (3 ints position[X], position[Y], position[Z] and one int health).
- Communication are done over UDP/IP: Those messages gets lost quite often on the internet.

Server Frame 1:

The Server has received a few updates from every client. They have impacted the Master gamestate (in green). It is now time to propagate the state to Client1:

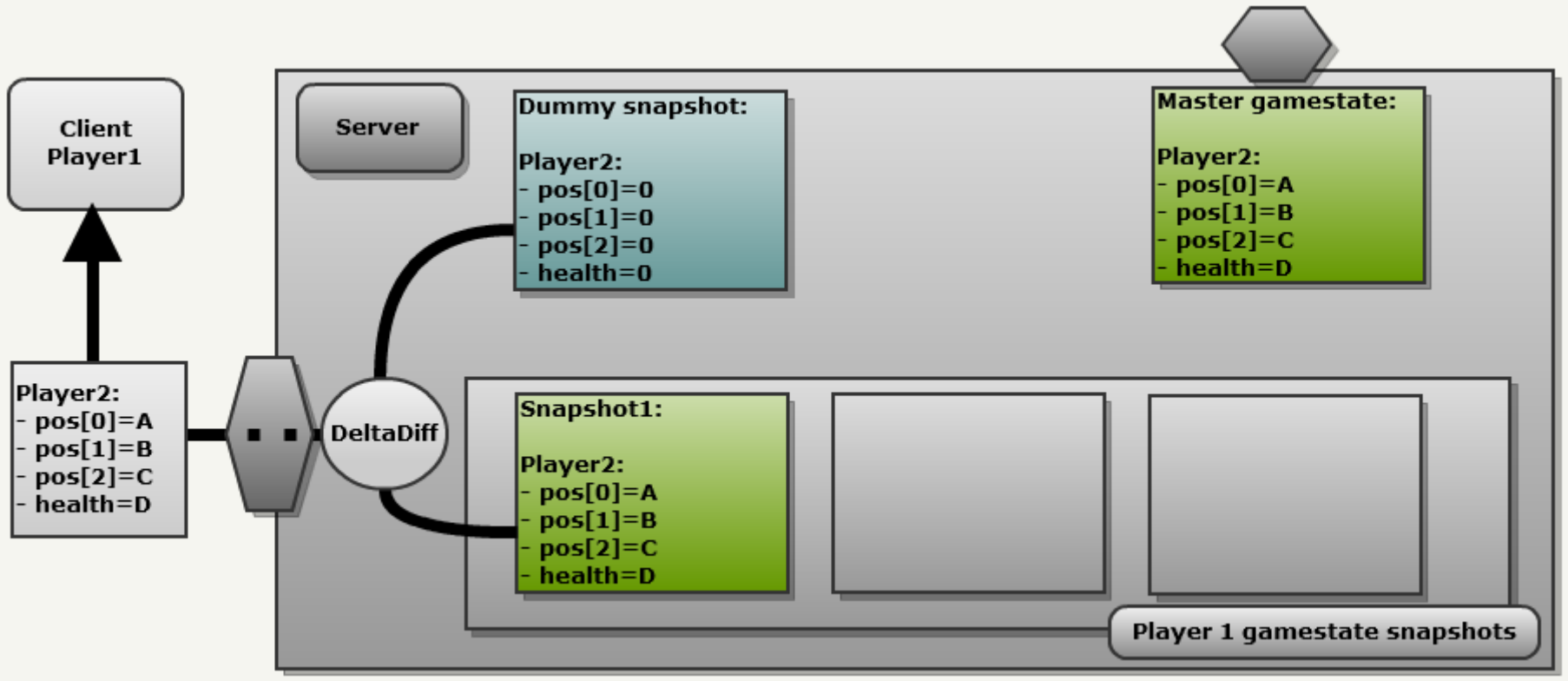


In order to generate a message the network module will ALWAYS do the following :

1. Copy the Master gamestate in the next Client history slot.
2. Compare it with an other snapshot.

This is what we can see in the next drawing:

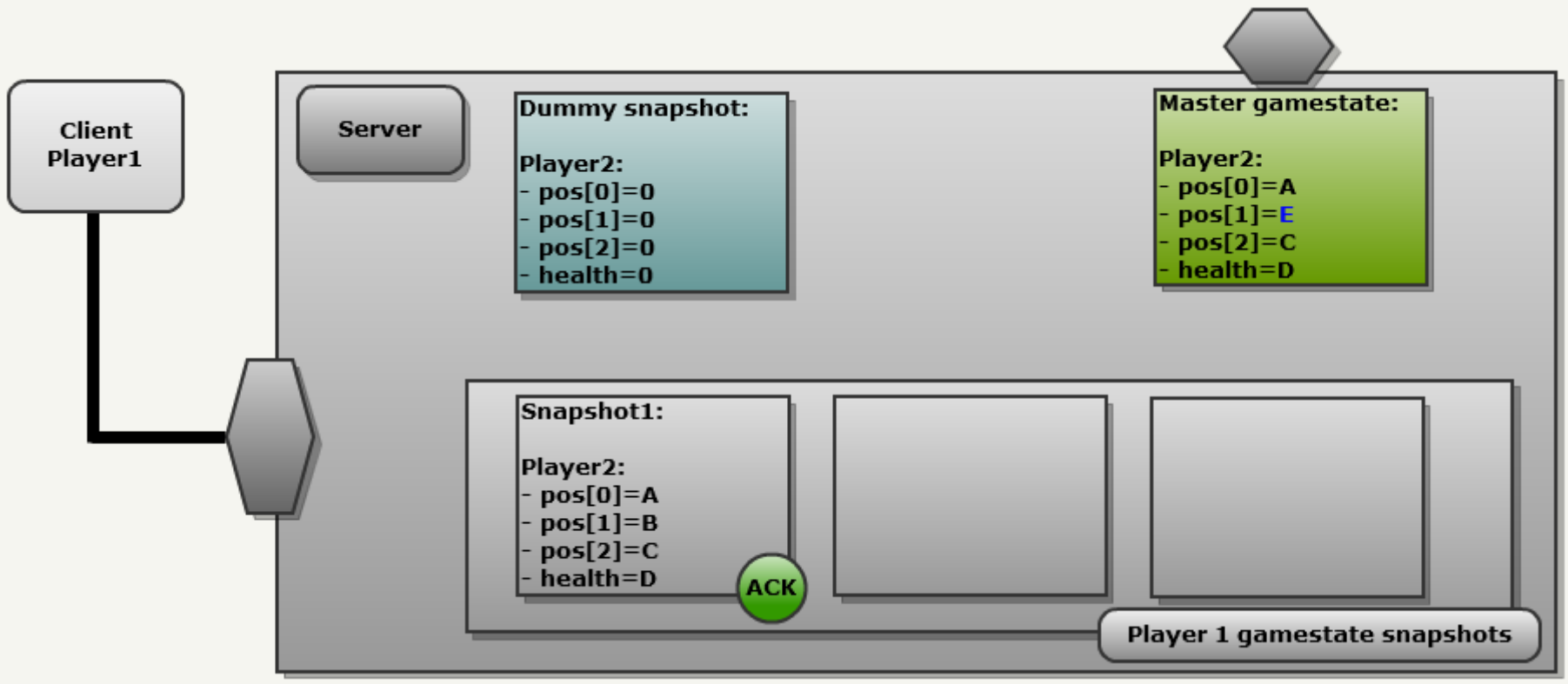
1. Master gamestate is copied at index 0 in Client1 history: It is now called"Snapshot1".
2. Since this is the first udpate, there are no valid snapshot in Client1 history so the engine is going to use the "Dummy snapshot" where all fields are always ZEROed. This results in a FULL update since every single field is sent to the NetChannel.



The key point to understand here is that if no valid snapshots are available in the client history the engine will pick "dummy snapshot" to generate a delta message. This will result in a full udpate sent to the Client using 132 bits (each field is preceeded by a bit marker): `[1 A_on32bits 1 B_on32bits 1 B_on32bits 1 C_on32bits]`.

Server Frame 2:

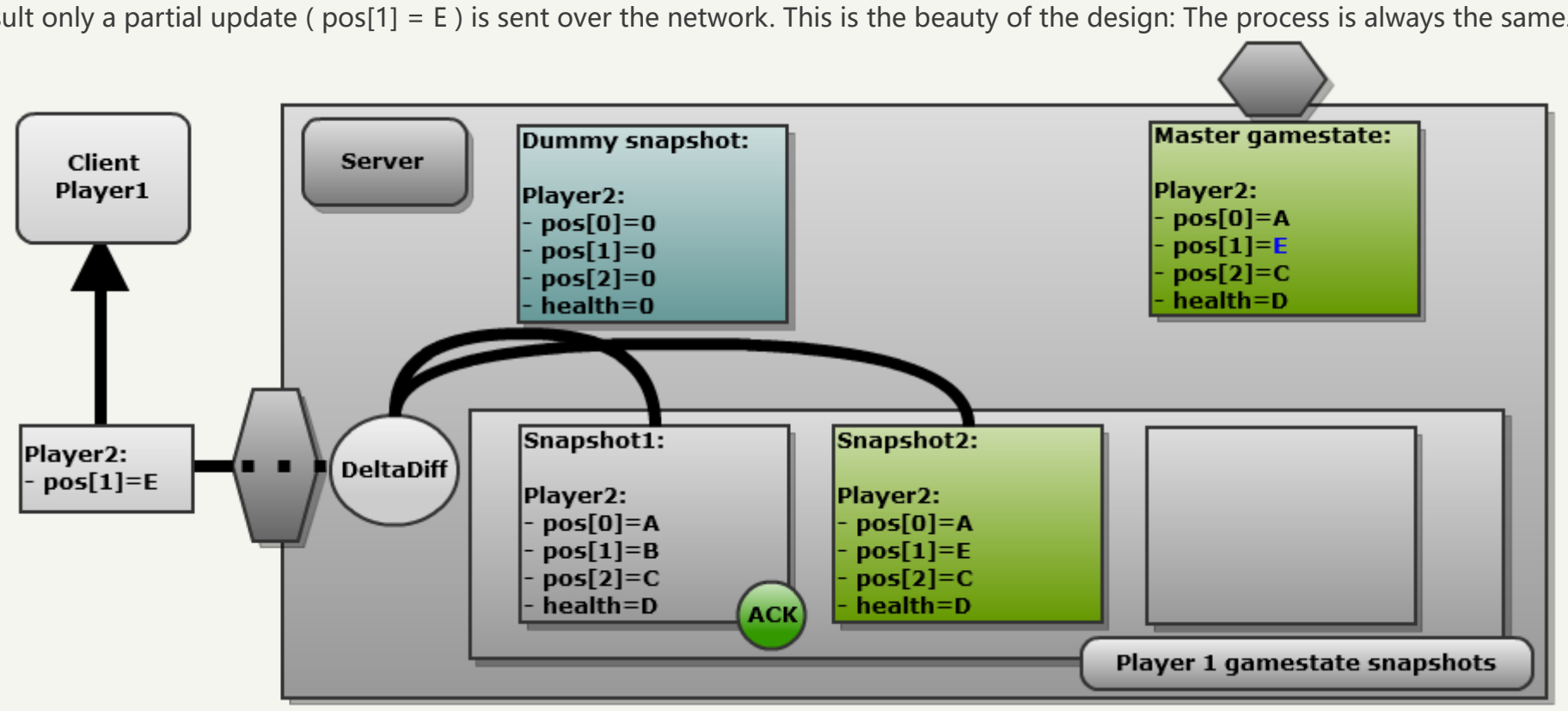
Now let's move forward in time: this is the Server second frame. As we can see each client have sent commands and they have impacted the Master gamestate: Client2 has moved on the Y axis so pos[1] is now equal to E (in blue). Client1 has also sent commands but more important it has also acknowledged receiving the previous udpate so Snapshot1 has been marked as "ACK":



The process is the same:

1. Copy the Master gamestate in the next Client history slot: (index 1): This is Snapshot2
2. This time we have a valid snapshot in the client history (snapshot1). Compare those two snapshots

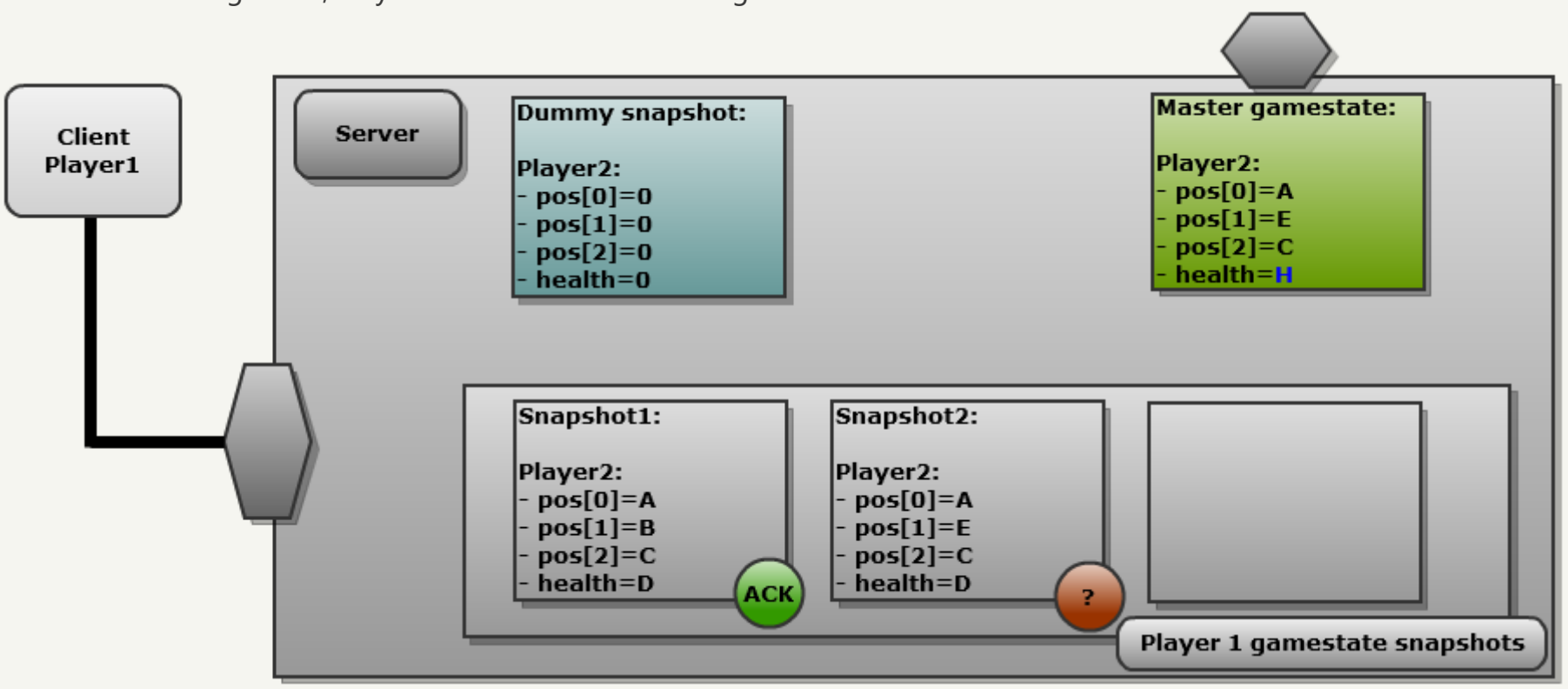
As result only a partial update (pos[1] = E) is sent over the network. This is the beauty of the design: The process is always the same.



Note : Since each field is preceeded by a bit marker (1=changed, 0=not changed) the partial update above would uses 36 bits: [0 1 32bitsNewValue 0 0].

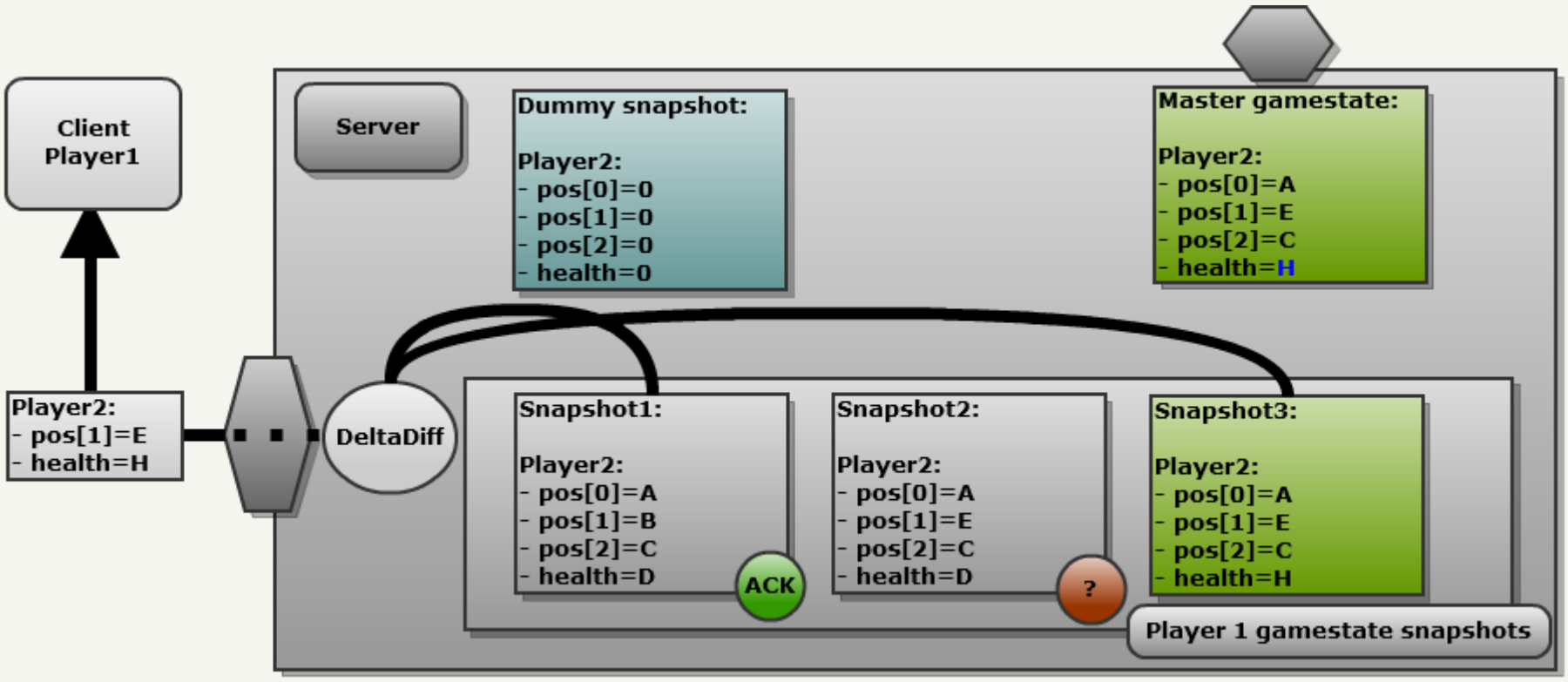
Server Frame 3:

Let's move forward one more step in order to see how the system deals with lost packets. This is now Frame 3. Clients have kept on sending commands to the server. Client2 has lost life and health is now equal to H. But Client1 has not acknowledged the last update. Maybe the Server's UDP got lost, maybe the ACK from the Client got lost but bottom line is that it cannot be used.



Regardless the process remains the same:

1. Copy the Master gamestate in the next Client history slot: (index 2): This is Snapshot3
2. Compare with the last valid acknowledged snapshot (snapshot1).



As a result the message sent it partial and contains a combination of old changes and new changes: (pos[1]=E and health=H). Note that snapshot1 could have been too old to be used. In this case the engine would have used the "dummy snapshot" again, resulting in a full update.

The beauty and elegance of the system resides in its simplicity. The same algorithm automatically:

- Generate partial or full update.
- Resend OLD information that were not received and NEW information in a single message.

Memory introspection with C

You may wonder how Quake3 is comparing snapshots with introspection...since C does not have introspection.
The answer is that each field locations for a `netField_t` is preconstructed via an array and some clever preprocessing directives:

```
typedef struct {
    char    *name;
    int     offset;
    int     bits;
} netField_t;

// using the stringizing operator to save typing...
#define     NETF(x) #x,(int)&((entityState_t*)0)->x

netField_t  entityStateFields[] =
{
{ NETF(pos.trTime), 32 },
{ NETF(pos.trBase[0]), 0 },
{ NETF(pos.trBase[1]), 0 },
...
}
```

The full code of this part can be found in snapshot.c's `MSG_WriteDeltaEntity`. Quake3 does not even know what it is comparing: It just blindly follow `entityStateFields`'s index,offset and size...and sends the difference over the network.

Pre-fragmentation

Digging into the code we see that the NetChannel module slices messages in chunks of 1400 bytes (`_Netchan_Transmit`), even though the maximum size of an UDP datagram is 65507 bytes. Doing so the engine avoid having its packets fragmented by routers while traveling over the internet since most network MTU are 1500 bytes. Avoiding router fragmentation is very important since:

- Upon entering the network the router must block the packet while it is fragmenting it.
- Upon leaving the network problems are even worse since every part of the datagram have to be waited on and then time-costly re-assembled.

Reliable and Unreliable messages

If the snapshot system compensate for UDP datagrams lost over the network, some messages and commands must be GUARANTEED to be delivered (when a player quits or when the Server needs the Client to load a new level).

This guarantee is abstracted by the NetChannel: I wrote about it a few years ago (wow my drawings have come a long way !!).

Recommended readings

Brian Hooks a member of the developing team wrote a little bit about the Network Model.

The author of Unlagged: Neil "haste" Toronto also described it.

Next part

The Virtual Machine system

@