Networking - Structuring Data

Introduction

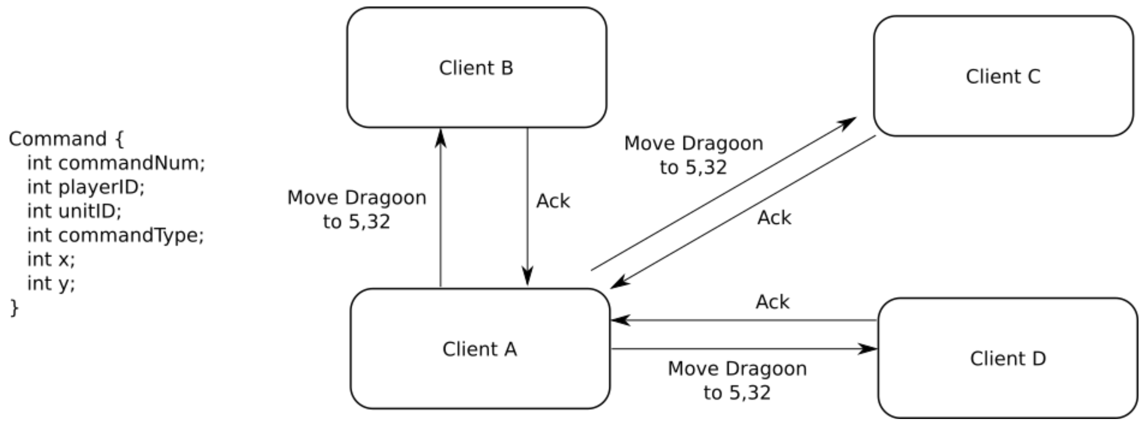
In the previous tutorial, we saw how to create connections between machines using the *Internet Protocol*, and how to send data using the *TCP* and *UDP* networking protocol. The actual contents of the data being sent along that network connection depends heavily on the type of game being played. Different genres tend to require different data to be transmitted, and their networking model may have an impact on the overall structure of a codebase. In this tutorial, we'll take a look at how RTS and FPS games generally structure their networking model, including what data they send, and perhaps more importantly, what data they *don't* send. We'll do this by taking a look at two games in particular: *Blizzard's* popular RTS game *StarCraft*, and the classic (and now open source) FPS competitive game *Quake 3: Arena* by *id Software*.

Case Study 1: StarCraft

For our first case study into game networking, we're going to look at a classic RTS game, Blizzard's *StarCraft*. In StarCraft, up to 8 players control 100s of units from a top down perspective, ordering their tanks and soldiers to outmanoeuvre and attack their opponent's units. This raises some interesting challenges for networking models, although the answer to the question of how a game from 1998 can support the movement of 100s of units across a 56k modem connection may surprise you.

Peer to Peer

StarCraft is primarily a *peer to peer* networked game; all of the player's on each team are all connected to each other, and they all send game data to each other, via the TCP networking protocol. This means the data packets are ordered and reliable as far as the clients are concerned, but may lead to laggy gameplay - we'll come on to why shortly. As StarCraft is closed-source, we can't be exactly sure of the exact structure of what is being sent, but what we *can* be sure of is that each unit in the game is not sent by the player that 'owns' the unit. There's really no need - player's don't have direct control over a unit, rather they click and issue orders to them, which the unit AI then tries to carry out according to its state machine. This is an important consideration in terms of reducing bandwidth, as each player's copy of the unit *should* do the same thing if told to to the same action, so sending positional information for every unit along the way should then be unnecessary. We can model the process of moving units in StarCraft as a series of commands sent from one player to the others:



```
Command {
  int commandNum;
  int playerID;
  int unitID;
  int commandType;
  int x;
  int y;
}
```

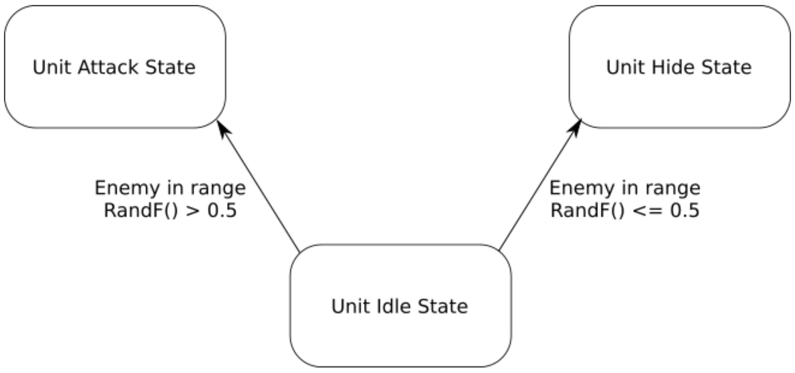
Each player sends commands relating to a particular unit, to do a particular thing (fire, move, use ability, and so on), at a particular point on the map. There could also be a 'targetID' type int as well - whatever is needed to fully encapsulate what a particular command is supposed to do. As packets containing commands are received, clients send an acknowledgement packet back, confirming that the packet has been received.

Lock-Step

What makes StarCraft's networking model is that it is a *lockstep* model. That is, the next true game frame (where units run their AI and play their animations etc) cannot begin until every client has acknowledged the other player's commands. In a way, we can think of StarCraft as being a turn-based game, with very short turns, where every player submits their command, and the game cannot continue until every player has done so. The side effect of this, is that the game only really runs as fast as the player with the slowest connection - if a player takes 300ms to receive the commands of frame n, then the other players in the game can't continue until they receive the acknowledgements another 300ms later, giving a very 'slowed down' feeling to the game.

Unlike with some other real-time games (including our next case study, Quake 3), as unit position information and so on is never sent, only commands, a player can never miss a command - or rather, if they do, they must be dropped from the game immediately! If a player never receives the command to move the Dragoon from one side of the map to the other, as far as that player is concerned the Dragoon is standing where it used to be, and can be shot at and blown up. What might really be happening, though, is that that Dragoon is in their base, shooting all of their builder units - they'd never know, as they never received the command moving the unit, and therefore never will, as the commands aren't resent, and they are the only way in which the game state is ever manipulated. The only option for StarCraft when a player falls behind or stops receiving messages is to disconnect them, as they have 'desynchronised' from the true game state:

By keeping the players synchronised in lockstep, the same things should happen on every player's copy of the game. However, to make this happen, the AI of each unit in the game must be completely *deterministic* - for a given set of inputs, it must *always* do the same thing. For example, the following very simple state machine could cause a lot of problems in a lockstep game like StarCraft:



Each player's copy of the game will generate its own random number whenever a unit comes within range of that AI, resulting in the unit doing different things on every machine. The solution to this is to have every frame of the game seed the random number generator with a unique identifier (even just an integer of how many frames have elapsed) for every successive 'round' of network transmission. If every machine then receives the same information, and runs the exact same code, and so acts on the information in the same way, then the same thing will happen. This might mean that the random number generator of a lockstep game isn't in fact very random, but this is actually what we want - a completely controllable 'pseudo-random' element to the game. Some games just have a preset array of random numbers generated when the game is compiled, with each successive call to get a random number just cycling to the next random number in the array - this removes any chance of divergence from machine to machine, if even they use different C runtimes, or operating systems.

Case Study 2: Quake 3

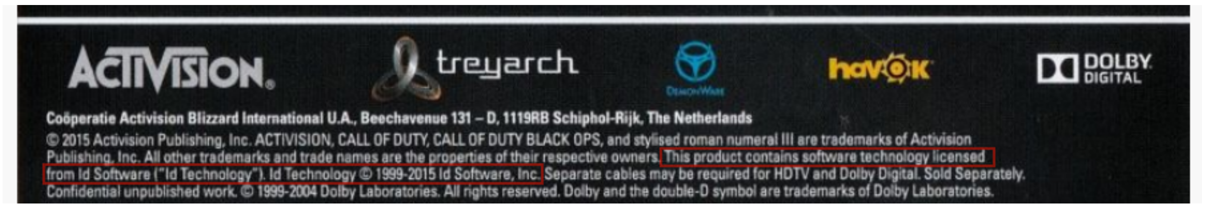
Fast paced games such as first person shooters generally do not follow the same lock-step execution pattern as an RTS game like the *StarCraft* example above. There's a few reasons for this:

1. There's a different number of game entities in most FPS games. Most FPS games have quite low player counts - Blizzard's *Overwatch* game is usually played as a 6v6. While games like *Player Unknown's Battlegrounds* and *Fortnite* have raised this to up to 100 players, that's still low compared to the 200+ units that might be spread across the map in a 2v2 game of *StarCraft*. This results in different bandwidth requirements, and a different overall way of communicating information to each player.

2. Latency matters. Unlike the distant overhead view of an RTS, we can see what is happening directly in front of us, making the jerky, periodic movements of a lockstep model more obvious. Even more so when the fire buttion is pressed - having to wait 100ms before the gun fires is very annoying, and will feel 'laggy'. In an RTS game, the delay time can be effectively hidden behind something that 'feels' like gameplay - maybe the unit responds with a voice cue, or plays an animation, before doing what what its told. This makes sense for a big, slow moving tank or formation of cavalry, but if your character doesn't respond instantly in an FPS, you instantly feel disconnected from the experience, something which has shown itself as even more of a problem in Virtual Reality applications.

3. Unpredictable nature. Once an RTS unit has been told what to do (either directly by a mouse click, or in response to some other event), the AI takes over, and will follow its instructions exactly - if the lock step mechanism has been done correctly, every copy of the game will do the same thing given the same input. In an FPS however, there's a player directly controlling every action of the character on screen, so game clients cannot really predict what the other players will do in a given time period, making it even harder to hide the latency of network gaming.

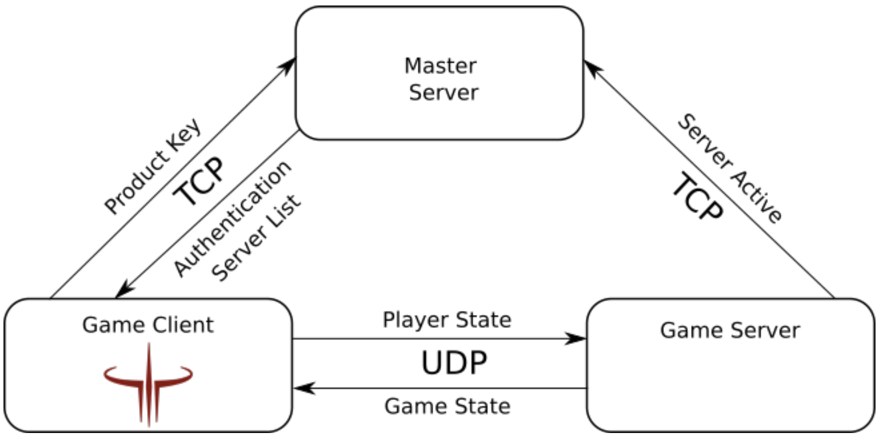
To see an example of how a fast-paced FPS game works over a network, we're going to take a look at the networking model of a very influential multiplayer game, *id Software's Quake 3:Arena*. Another old game! Surely a game from 1999 can't tell us anything about the technology in modern FPS games? Well, let's take a look at the back of the box of 2015's *Call of Duty: Black Ops 3*, by *Activision*:


Coöperatie Activision Blizzard International U.A., Beechavenue 131 – D, 1119RB Schiphol-Rijk, The Netherlands
© 2015 Activision Publishing, Inc. ACTIVISION, CALL OF DUTY, CALL OF DUTY BLACK OPS, and stylised roman numeral III are trademarks of Activision Publishing, Inc. All other trademarks and trade names are the properties of their respective owners. This product contains software technology licensed from Id Software ("Id Technology"). Id Technology © 1999-2015 Id Software, Inc. Separate cables may be required for HDTV and Dolby Digital. Sold Separately. Confidential unpublished work. © 1999-2004 Dolby Laboratories. All rights reserved. Dolby and the double-D symbol are trademarks of Dolby Laboratories.

Not so outdated after all! Nearly every Call of Duty game is built on a game engine derived from *id Software's idTech3* engine, so that code written back in 1999, including its networking model, has found its way onto many platforms, and powered many popular games. Even more useful is that the full source code to id's Quake 3:Arena game was made open source in 2005, meaning its freely available to examine, making it ideal for a case study.

Client-Server Model

Unlike StarCraft's *peer to peer* model, Quake 3 and most other FPS games use a *client-server* model, whereby all players in a particular matchup all connect to the same single server, which fully arbitrates all communication between players - player's never know each other's IP address, only the server's. They all also connect to a single master server to authenticate their game keys, so that pirated copies of the game won't allow player's to play online (somewhat of a drawback to an essentially online-only multiplayer FPS game!).



The server runs a copy of the game just as the clients do, meaning that the server has the same game level data, and the same code to execute on in-game events, making communication between clients and servers relatively simple - each client can load the game level to know where various game entities are (health packs, power ups, spawn points, and so on), and both the client and server will know properties on each entity (how fast it should move per frame, and so on), as they can load the same data from the same files.

The server is the ultimate authority on where entities (whether they are players, rockets, or powerups) are in the world, so while the server can tell clients where things are in the game, there's no need for clients to send the position of their player character in the world - instead, they send the keys pressed and mouse movement, which the server can then used to run a 'frame' of the game just as clients do, running the calculations required to determine the 'true' position of any player in the world.

Client transmission

The main data sent by the player to the server every game tick is the **usercmd_t struct**. Here's its definition (within the *q_shared.h* file, line 1248):

```
typedef struct usercmd_s {
   int          serverTime;
   int          angles[3];
   int          buttons;          //use item etc
   byte         weapon;           // weapon active
   signed char forwardmove, rightmove, upmove;
} usercmd_t;
```

idTech3 usercmd_s struct

There's not much to it! But its enough to represent everything the player can do in an FPS game. There's 3 **chars**, to represent forward/backward movement, sidestepping, and jumping/ducking. There's also a buttons variable, which is used as a *bitmask* - that is, every bit is used like an individual boolean value, allowing for 8 states to be contained within a single byte. In *Quake 3:Arena* this is used to store whether the player is currently firing their weapon, or using a currently held powerup, but could just as easily represent the player trying to use their character's abilities in a game like *Overwatch*, or throw their grenade in *Call of Duty*.

The *serverTime* variable is pretty important, as it will let the server know what game tick the client *thinks* they're currently in, and is used to order received packets relative to each other.

The *angles* variables are used to store the result of pitch and yaw from mouse movement (or joypad, or keyboard movement - the server doesn't care *how* the player rotated, only that they *did*), and any 'roll' from visual effects related to knockback from being hit by a BFG 10K.

For the most part, this is all a client ever sends to the main game server - the entire state of how a player moves and affects the world can be summed up by the keys / buttons that have been pressed at any particular point in time, so sending this data to the server is all we need to replicate the game state, making for a very efficient client to server transmission; this is handy as most players will be on *asymmetric* connections with much more download than upload, so minimising uploaded data is very important for overall network game quality.

Server transmission

There's two different types of data sent to each client, either *commands*, or *snapshots*. Commands are used to tell each client that something important has happened - a player has disconnected, connected, or typed something in chat, for example. Commands are transmitted with a call to *SV_GameSendServerCommand*:

```
void SV_GameSendServerCommand( int clientNum, const char *text ) {
   if ( clientNum == -1 ) {
      SV_SendServerCommand( NULL, "%s", text );
   } else {
      if ( clientNum < 0 || clientNum >= sv_maxclients->integer ) {
         return;
      }
      SV_SendServerCommand( svs.clients + clientNum, "%s", text );
   }
}
```

SV_GameSendServerCommand

At first glance, it seems somewhat odd that a command is always text-based; surely a single **struct** with an **int** player ID and an **int** to cast to an **enum** of actions to enact on that player would do? While this is true, the idTech way works as it ties directly into the internal command console of the engine - you can press the tilde key at any time to bring down the command console, and type in commands like a DOS prompt - typing in 'disconnect 1' would try to kick player 1 off the server, for instance. Commands received by a client are sent through to the same code that typing into the console is, allowing any new command to be sent without needing it to also be represented by another identifier, and, as commands are only occasionally sent, they are unlikely to saturate the bandwidth of a client's connection, even as strings.

When commands are sent, they are done so by also sending a command sequence number - each client gets their own sequence integer as part of the server structure used to represent them, **client_s**:

```
typedef struct client_s {
...Stuff up here
   char  reliableCommands[MAX_RELIABLE_COMMANDS][MAX_STRING_CHARS];
   int   reliableSequence;
   int   reliableAcknowledge;
   int   reliableSent;
...more stuff down here
```

idTech3 client_s struct

Upon receiving a command, the client sends an acknowledgement that they have received it, allowing the server to increment their own internal *reliableAcknowledgement* variable for that player.

If the server has to send new commands before having received an acknowledgement for a command, the server will retransmit the 'missing' commands along with the new one, as commands must be processed in a *strict ordering* - the game state should not be allowed to diverge from what the server thinks is going on, which could happen if the commands 'player 0 won the round' and 'player 0 disconnected' were handled in an incorrect order. This is a demonstration that, although UDP is an 'unreliable' message protocol, some degree of network reliability can be encoded within the UDP packets being sent - always work on the assumption that a packet *might* go missing, and that until you know otherwise, you should be able to re-send a network packet until it is received by everybody. Client code should also check that they aren't receiving commands that, due to network latency, they've received copies of due to not being able to send an acknowledgement in time.

Server Snapshots

The Quake 3 networking model uses the idea of snapshots of the game state at any particular period in time. As different packets may be received or lost at different points in time, each client connected to the game has its own list of snapshots stored on the server, with each representing what the server knows about the world at a particular point in time. This is represented in Quake 3 as a series of **snapshot_t structs** for each connected player:

```
typedef struct {
   int              snapFlags;
   int              ping;
//from this / snapshot rate, we know last state
   int              serverTime;

   byte         areamask[MAX_MAP_AREA_BYTES];
   playerState_t  ps;

   int              numEntities;
   entityState_t  entities[MAX_ENTITIES_IN_SNAPSHOT];

   int              numServerCommands;
   int              serverCommandSequence;
//MORE STUFF!
}
```

idTech3 snapshot_t struct

For every network synchronised entity in the world, an **entityState_s** struct is stored, containing the following data (amongst others):

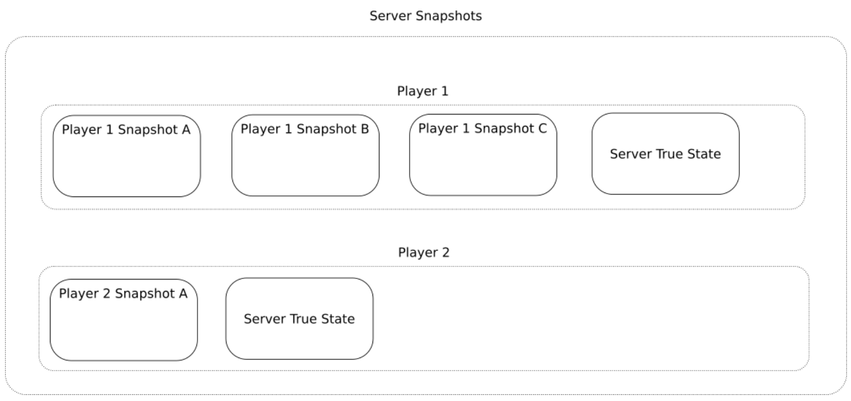
```
typedef struct entityState_s {
   int      number;        // entity index
   vec3_t   origin;
   vec3_t   origin2;

   vec3_t   angles;
   vec3_t   angles2;
   int      clientNum;
   int      frame;
//lots more stuff!
} entityState_t;
```

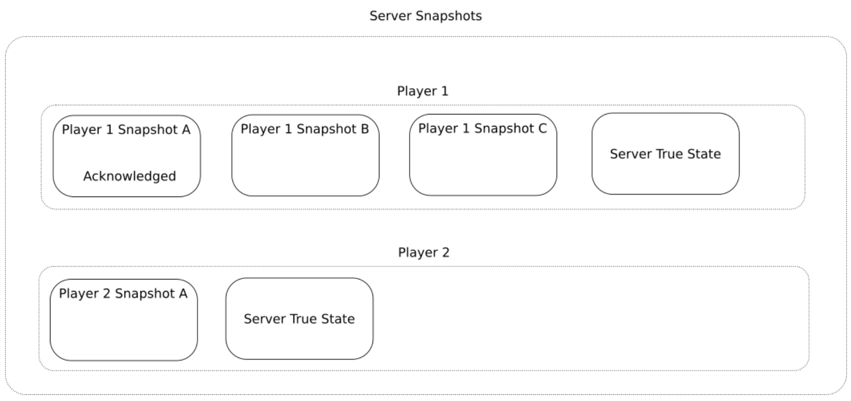
idTech3 entityState_s struct

Most important is that the position and orientation is sent, along with the animation frame and a unique identity number. Each object is stored in the list of **entityState_ts** within a **snapshot_t**, and the server stores a set of **snapshot_t** structs for each connected client. To see why, lets look at an example of some data transmission in the game.

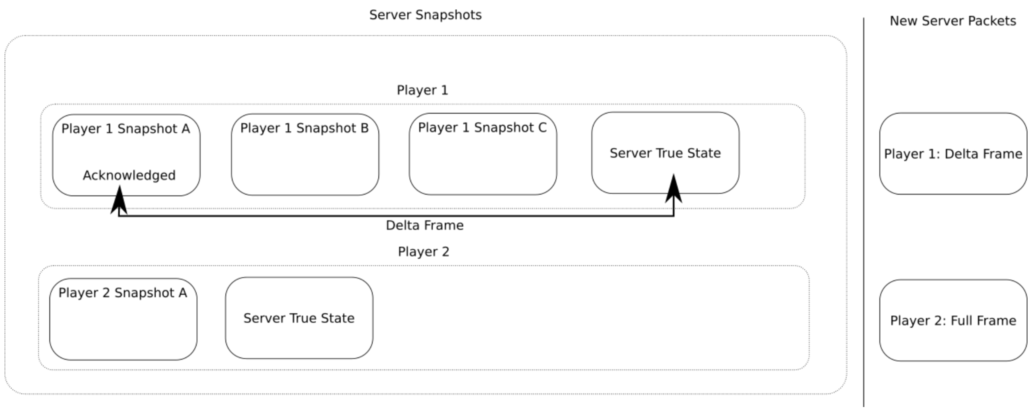
The server runs at a particular update rate - by default it was 20hz, but could be up to 120hz in later versions of the game. When its time for a new update, the game makes a copy of the true state of the game, and places it into the next available 'slot' for each client:



From there, the server moves on to sending new information to each client so that they can 'see' the latest game state. But which slot's snapshot to send to each player? As clients receive data, they later send packets containing the ID of the last snapshot they received, as an acknowledgement to the server that it can move on to sending the next game state. For now, we'll assume that client 2 has not yet acknowledged any snapshots (perhaps they've just connected), while client 1 has successfully acknowledged at least one prior game snapshot:



So far, so good! Now, each of those **snapshot_t** structures might be quite big - there's up to 256 network synchronised entities stored in a single **snapshot_t**. What the game server will do is try to only send the *delta* of the current **snapshot_t** against the last *acknowledged* snapshot - that is, it will try to only send the data that's actually changed since the last time a client said they received the game state. In this case, player 1 has acknowledged a game state snapshot, and so only needs to be sent the delta of the latest state and this previous state, while player 2 has yet to acknowledge a snapshot, and can therefore only be sent the full game snapshot:



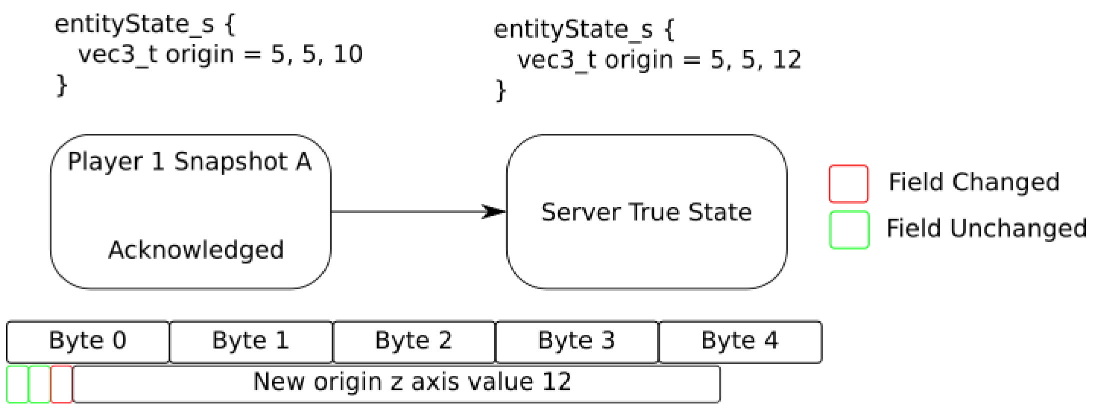
Client 1 *might* have received snapshots B and C, but the server doesn't know this yet, either because of network latency or packet loss. By not making any assumptions over what might have been received, this network model is robust against missing data - if snapshot C is *never* acknowledged, it doesn't matter *that* much, the server will just keep calculating the difference between the current game snapshot and the last acknowledged one, and send that.

Overall, this means that *reliable* connections will get more compressed data (not much changes between snapshots, so not much data needs to be sent), but *unreliable* connections can still operate, albeit at greater bandwidth cost, and greater memory on the server for storing all of the game states for each client. As clients acknowledge snapshots, the server can recycle the memory of 'older' snapshots, as they won't be required any more - the server always works out the difference between the *latest* state, and the *last acknowledged* state.

Snapshot Compression

You might be wondering exactly what form this snapshot compression takes. Rather than sending the snapshot as-is, or compressing it with a lossless compression algorithm like a zip file, snapshots are compressed as a form of delta compression - the client has some state (the game world as of the last snapshot received and acknowledged), and the incoming data is based off the *difference* to that state. To do this, the server works out whether every variable within the snapshot is actually different than in the previously acknowledged state - if it's the same, the server just writes a single 0 bit to an array of bytes that will be sent as a packet. If it has changed, instead a 1 is written, and the server then writes the value into the array. This means that a particular variable in the snapshot might start half way through a byte, requiring more work to 'uncompress' back into the full data structure, but it does save some bandwidth. While an **int** is always 32 bits, there might be some additional meaning to an **int** that means that actually, less bits are required - Quake3 limits the number of active entities in a level to 1024, so while the *numEntities* variable inside **snapshot_t** might be an **int**, the server only ever needs to transmit 10 bits to represent its valid range.

As a quick example of this in action, here's how a simple 3 component vector might be sent in a game state, assuming that snapshot A is the latest state the server knows the player has received. As the x and y axes haven't changed, they can be represented as a single bit, but as the z axis has changed, the snapshot writes a 1 to its data buffer, and then the 32 bits of the value. Even with the extra bits being written, the position has gone from 12 bytes in total, to less than 4 and a half, and the data can be compressed beyond *that* by using the same compression techniques as found in programs like **7zip**.



Tutorial Code

Different games require different network data, based on many factors - how often the game itself updates, whether there is a real time requirement, and whether it can be guaranteed that every player will have the same deterministic result to a particular input. Therefore there's no real single recipe for success with our networked games - even the packets being sent could be compressed in different ways using a bit buffer or a delta state computed from some previous state in some way.

While there's not one single way of structuring data, we can at least look at the basic skeleton of a data transmission model that could be used to get some of the objects defined within the codebase moving around according to network traffic. To do so, we're going to take a look at some new classes - one to represent the network view of an object, and some to represent the packets needed to send that network view to another machine.

NetworkObject class

Inside the **GameObject** class, every instance can also have a **NetworkObject** - this is designed to represent something that should be 'network synchronised' - the server should send data relating to the world state of these objects to any connected clients, and clients should apply incoming data to these objects. To do so, the state of an object for sending across the network can be expressed in terms of a small **NetworkState** struct, which looks as follows:

```cpp
namespace NCL {
   using namespace Maths;
   namespace CSC8503 {
      class GameObject;
      class NetworkState   {
      public:
         NetworkState();
         virtual ~NetworkState();

         Vector3     position;
         Quaternion  orientation;
         int         stateID;
      };
   }
}
```

NetworkState Class header

Different games will have different requirements for objects (and as we saw in the StarCraft case study, may not even *need* to synchronise objects), but this seems like a good baseline to work from in other real-time games. Along with a position and orientation, this also stores a *stateID* integer - this is used to order incoming packets correctly when the could be received out of order due to packet switching. Every time the server sends a **NetworkState**, it will increment that value - if a client receives a **NetworkState** with a *stateID* less than its currently accepted value, it was 'old', and should be ignored.

Back to the **NetworkObject** itself. Its main purpose is to send and receive packets to transform into **NetworkState** objects, and apply them to a passed in **GameObject**. To make sure that the correct state packets are sent to the correct objects, each object will be given a specific integer ID - if the game code being ran on the client and the server are identical, they should be able to generate identical integer IDs for **GameObjects** constructed as part of the 'level', while dynamic player objects can also be handled in game code (player one could always have the ID '1000', player 2 '2000' etc, to separate them out from the world objects).

The only **public** methods are to either read and parse a packet received from the server, or to build up the correct packet for sending to a client. To get an idea of how a more compact 'snapshot' type system might work, the packets being send might either be 'full' packets with a complete **Vector3** for position and **Quaternion** for orientation, or a ’delta’ packet , which is the difference between the current server gamestate and the latest acknowledged packet from the client.

```cpp
class NetworkObject     {
public:
   NetworkObject(GameObject& o, int id);
   virtual ~NetworkObject();

   //Called by clients
   virtual bool ReadPacket(GamePacket& p);
   //Called by servers
   virtual bool WritePacket(GamePacket** p,
      bool deltaFrame, int stateID);
   void UpdateStateHistory(int minID);

protected:

   NetworkState& GetLatestNetworkState();

   bool GetNetworkState(int frameID, NetworkState& state);

   virtual bool ReadDeltaPacket(DeltaPacket &p);
   virtual bool ReadFullPacket(FullPacket &p);

   virtual bool WriteDeltaPacket(GamePacket**p, int stateID);
   virtual bool WriteFullPacket(GamePacket**p);

   GameObject& object;

   NetworkState lastFullState; //latest full NetworkState

   std::vector<NetworkState> stateHistory;

   int networkID; //id of this object
};
```

NetworkObject Class header

New GamePackets

To represent our new data transfer, we have 3 new **GamePackets**, defined in *NetworkObject.h*:

```cpp
struct FullPacket : public GamePacket {
   int      objectID;
   NetworkState fullState;

   FullPacket() {
      type = Full_State;
      size = sizeof(FullPacket) - sizeof(GamePacket);
   }
};
struct DeltaPacket : public GamePacket {
   int      fullID;
   int      objectID;
   char  pos[3];
   char  orientation[4];

```

```
         DeltaPacket() {
            type = Delta_State;
            size = sizeof(DeltaPacket) - sizeof(GamePacket);
         }
      };
      struct ClientPacket : public GamePacket {
         int      lastID;
         char  buttonstates[8];
         ClientPacket() {
            size = sizeof(ClientPacket) - sizeof(GamePacket);
         }
      };
```

New NetworkObject packets

A **FullPacket** is the most obvious to understand - it wraps up a **NetworkState** struct, along with the integer ID of the object to apply the **NetworkState** to. The **DeltaPacket** might need a little more explaining, though! The *pos* and *orientation* chars are to store the difference from the last 'full' position and orientation, in the hope that they haven't changed too much. The *objectID* is used as in **FullPacket**, while *fullID* is used to determine which **NetworkState** the position and orientation should be a delta *of* - if we use the wrong base data when trying to reconstruct a full position and orientation, things will definitely go wrong!

The last packet, **ClientPacket**, represents what data the client sends to the server. As in the Quake3 example earlier in the tutorial, this doesn't have to be a position or orientation, but can instead be the buttons and mouse movements of the player, which can be send to the server and processed to produce the same result as part of a game frame. Your game may well have different requirements, but the ability to press 8 different things seems like a good start (forward, backwards, left, right, jump, duck and 2 attack buttons could be stored within this, for instance). The only other variable, *lastID*, is used to send to the server an *acknowledgement* of which **NetworkState** was last successfully received from the client, letting it know that it can try and form delta packets from a later point in time.

Writing Packets

The server can tell a **NetworkObject** to write a new **GamePacket** containing its state via the *WritePacket* method. This takes in a pointer to a pointer - depending on whether it should fill a 'full' packet or a 'delta' packet, this may get filled with either a **FullPacket** or a **DeltaPacket**; once the packet has been made, the server doesn't really care what it is, just that it needs to be sent out to clients. When a delta packet is made, it is done so against a specific state that the server thinks the client should already have, but for whatever reason, if this fails (line 4), the server can always try to send a full packet to that player instead (line 5).

```
1 bool NetworkObject::WritePacket(GamePacket** p, bool deltaFrame,
2          int stateID) {
3    if (deltaFrame) {
4       if (!WriteDeltaPacket(p, stateID)) {
5          return WriteFullPacket(p);
6       }
7    }
8    return WriteFullPacket(p);
9 }
```

NetworkObject::WritePacket method

When a **FullPacket** is written by the server, the latest state of the object will always be encoded in it. The packet will also get an increases *stateID* integer - individual clients will be behind this value by some amount, but the server will be able to work this out based on a map, which we'll see shortly; as far as this function is concerned, we're creating a new 'snapshot' of the whole world, which individual clients can then try and catch up to.

```cpp
bool NetworkObject::WriteFullPacket(GamePacket**p) {
   FullPacket* fp = new FullPacket();

   fp->objectID             = networkID;
   fp->fullState.position   = object.GetTransform().GetWorldPosition();
   fp->fullState.orientation =
      object.GetTransform().GetWorldOrientation();
   fp->fullState.stateID      = lastFullState.stateID++;
   *p = fp;
   return true;
}
```

NetworkObject::WriteFullPacket method

In the case where the server tries to write a delta packet, it will try to do so against an existing **NetworkState**, selected by the *stateID* variable. If for some reason this **NetworkObject** doesn't have that particular existing state, we need to return **false** (line 5). If we can continue, we fill out our **DeltaPacket**, letting it know what state it is a delta of (line 8) and which object it is a delta of (line 9). On line 15 and 16 we work out the difference between the current game state's position and orientation, and subtract the selected states variables, giving us the difference since that previously written state. In order to efficiently represent the change in position and orientation, we store just a single byte per field. For position, we're going to just hope the object has moved less than 127 units on any particular axis - we'll skip the fractional part entirely, and let 'full' packets fill this gap. For orientations, as we know that each variable inside the quaternion will be within the range -1 to 1, we can expand that out to the full range of a byte by multiplying it by 127, so that it's now -127 to 127 - we can divide this back down later on the client side, but for now we can just finish off the method by writing the new **GamePacket** to the input parameter, and return true.

```cpp
 1 bool NetworkObject::WriteDeltaPacket(GamePacket**p, int stateID) {
 2    DeltaPacket* dp = new DeltaPacket();
 3    NetworkState state;
 4    if (!GetNetworkState(stateID, state)) {
 5       return false; //can't delta!
 6    }
 7
 8    dp->fullID      = stateID;
 9    dp->objectID    = networkID;
10
11    Vector3     currentPos   = object.GetTransform().GetWorldPosition();
12    Quaternion  currentOrientation   =
13             object.GetTransform().GetWorldOrientation();
14
15    currentPos            -= state.position;
16    currentOrientation    -= state.orientation;
17
18    dp->pos[0] = (char)currentPos.x;
19    dp->pos[1] = (char)currentPos.y;
20    dp->pos[2] = (char)currentPos.z;
21
22    dp->orientation[0] = (char)(currentOrientation.x * 127.0f);
23    dp->orientation[1] = (char)(currentOrientation.y * 127.0f);
24    dp->orientation[2] = (char)(currentOrientation.z * 127.0f);
25    dp->orientation[3] = (char)(currentOrientation.w * 127.0f);
26    *p = dp;
27    return true;
28 }
```

NetworkObject::WriteDeltaPacket method

Reading Packets

Our NetworkObjects can receive two types of packets - either a **DeltaPacket** or a **FullPacket**, as outlined earlier. The *ReadPacket* method receives a **GamePacket**, and determines whether its one of these two types, and if so, casts it, and passes the result on to the two functions that actually perform the packet data extraction, *ReadDeltaPacket* and *ReadFullPacket*.

```cpp
bool NetworkObject::ReadPacket(GamePacket& p) {
   if (p.type == Delta_State) {
      return ReadDeltaPacket((DeltaPacket&)p);
   }
   if (p.type == Full_State) {
      return ReadFullPacket((FullPacket&)p);
   }
   return false; //this isn't a packet we care about!
}
```

NetworkObject::ReadPacket method

If the received packet is a 'delta' packet, we need to first work out whether we can actually perform the delta calculation. To do so, we examine the *fullID* of the incoming **DeltaPacket**, and see whether its the same as the last accepted 'full' state; if it's not, we can't do anything with it (it doesn't matter whether the delta is newer or older, this client just simply doesn't have the data to correctly process it), and so we need to just return false. If we *do* accept the delta packet, we can now dump any old **NetworkState** objects that the **NetworkObject** is holding - we'll never need to use them again, so we can just clean up the memory! Once we've done that, we can reconstitute the change in position and orientation by doing the inverse as we did when forming the delta - we now divide the quaternion bytes by 127 and turn them back into floats, and add on our position data (we might lose some of the fractional part of the 'true' position, but this will later be fixed next time the server sends a 'full' packet).

```cpp
bool NetworkObject::ReadDeltaPacket(DeltaPacket &p) {
   if (p.fullID != lastFullState.stateID) {
      return false;//can't delta this frame
   }
   UpdateStateHistory(p.fullID);

   Vector3     fullPos          = lastFullState.position;
   Quaternion  fullOrientation  = lastFullState.orientation;

   fullPos.x  += p.pos[0];
   fullPos.y  += p.pos[1];
   fullPos.z  += p.pos[2];

   fullOrientation.x  += ((float)p.orientation[0]) / 127.0f;
   fullOrientation.y  += ((float)p.orientation[1]) / 127.0f;
   fullOrientation.z  += ((float)p.orientation[2]) / 127.0f;
   fullOrientation.w  += ((float)p.orientation[3]) / 127.0f;

   object.GetTransform().SetWorldPosition(fullPos);
   object.GetTransform().SetLocalOrientation(fullOrientation);
   return true;
}
```

NetworkObject::ReadDeltaPacket method

If the received packet is a 'full' data packet, we can just grab the position and orientation straight out of it, and set it on our object's **Transform**. We do need to make sure that it wasn't an old, or otherwise delayed packet, that contains an out of date **NetworkState** - we can determine this by checking to see whether the newly received *stateID* is less than the last accepted *stateID* - if its less, it must be 'older', and should therefore just be discarded.

```
bool NetworkObject::ReadFullPacket(FullPacket &p) {
   if (p.fullState.stateID < lastFullState.stateID) {
      return false; // received an 'old' packet, ignore!
   }
   lastFullState = p.fullState;

   object.GetTransform().SetWorldPosition(lastFullState.position);
   object.GetTransform().SetLocalOrientation(lastFullState.orientation);

   stateHistory.emplace_back(lastFullState);

   return true;
}
```

NetworkObject::ReadFullPacket method

Maintaining State History

As the client receives **NetworkStates**, it stores them in the *stateHistroy* vector temporarily, as it may need them later, to calculate a 'delta' from. When a client accepts a delta packet, or the server receives acknowledgement from a client that a **NetworkState** has been received, we can get rid of old **NetworkState** objects, based on their *stateID* variable. We can do so by iterating over the *stateHistory* vector, and removing any entries that have a *stateID* less than some parameter value.

```
void NetworkObject::UpdateStateHistory(int minID) {
   for (auto i = stateHistory.begin(); i < stateHistory.end(); ) {
      if ((*i).stateID < minID) {
         i = stateHistory.erase(i);
      }
      else {
         ++i;
      }
   }
}
//get the latest state received from a server
NetworkState& NetworkObject::GetLatestNetworkState() {
   return lastFullState;
}
//get a particular saved state on either the client or server side
bool NetworkObject::GetNetworkState(int stateID, NetworkState& state){
   for (auto i = stateHistory.begin(); i < stateHistory.end(); ++i) {
      if ((*i).stateID == stateID) {
         state = (*i);
         return true;
      }
   }
   return false;
}
```

NetworkObject::ReadFullPacket method

Integrating networking into your game

The tutorial codebase you have been provided with comes with a subclass of **TutorialGame**, suitably enough called **NetworkGame**. This can be used as the basis for either a client or a server version of your game. It contains some example methods that you can modify to send data to the server version of the game, which can then determine correct delta packets to try and send back. For example, here's the *UpdateAsClient* method:

```
 1 void NetworkedGame::UpdateAsClient(float dt) {
 2    ClientPacket newPacket;
 3
 4    if (Window::GetKeyboard()->KeyPressed(KEYBOARD_SPACE)) {
 5       //fire button pressed!
 6       newPacket.buttonstates[0] = 1;
 7       newPacket.lastID = 0; //You'll need to work this out somehow...
 8    }
 9    thisClient->SendPacket(newPacket);
10 }
```

NetworkObject::UpdateAsClient method

This will work out whether particular keys have been pressed, and collate them into a **ClientPacket** ready for sending to a server. However, it so far has no way of working out the last acknowledged **FullPacket** received from a server - you should create the correct **PacketReceiver** to handle getting messages of type **Full_State** on the client, to then update some integer that can then be used in place of the 0 on line 7 in the **UpdateAsClient** method.

Similarly, here's the *BroadcastSnapshot* method of the **NetworkedGame** class:

```
11 void NetworkedGame::BroadcastSnapshot(bool deltaFrame) {
12    std::vector<GameObject*>::const_iterator first;
13    std::vector<GameObject*>::const_iterator last;
14
15    world->GetObjectIterators(first, last);
16
17    for (auto i = first; i != last; ++i) {
18       NetworkObject* o = (*i)->GetNetworkObject();
19       if (!o) {
20          continue;
21       }
22       int playerState = 0; //You'll need to do this bit!
23       GamePacket* newPacket = nullptr;
24       if (o->WritePacket(&newPacket, deltaFrame, playerState)) {
25          thisServer->SendGlobalPacket(*newPacket); //change...
26          delete newPacket;
27       }
28    }
29 }
```

NetworkObject::UpdateAsClient method

It will go through every object of the world, and if it has a **NetworkObject** component (and thus, should be network synchronised), will use the *WritePacket* method outlined earlier to try and create the correct packet. However, there's currently two problems with this method. For one, There's currently no way for the server to know which packets have been successfully received by the client and acknowledged - you could perhaps store this inside a map between a player number and an int, or directly inside some sort of 'Player' class. This will then be used in place of the *playerState* variable on line 22. The other problem is that the server shouldn't send the same data to every client (line 25) - you will need a method that can send a specific packet to a specific player, and then put the for loop starting on line 17 inside another for loop iterating over each player, so that unique packets can be sent to each player based on what last state they received and acknowledged. Remember that when the eNet library receives an event (as in the **GameServer::UpdateServer** method), it can determine a particular peer object - it you store this peer object when an event of type **ENET_EVENT_TYPE_CONNECT** is received, you can then gain a mapping of player to eNet peer, which is pretty handy for then using in a server call to **enet_peer_send** to send a packet to a particular player.

Conclusion

Maintaining consistency of state across multiple players is a tricky problem, and different game types handle this in different ways. If a game is designed to run in *lockstep*, consistency is fairly easy to maintain, at the expense of the game slowing down while everyone waits for someone to acknowledge the current state of the world. Part of this consistency will also be down to the deterministic nature of the game being networked - the same events must lead to the same result on each machine, or the consistency of state will be lost, and each player will see a slightly different version of the game.

In games with more real-time restrictions such as Quake 3, inconsistency of state must be accommodated for in a different way. Lost packets and delayed acknowledgements from players cannot slow the game down, but can be rectified via the use of game snapshots; with each being a copy of the full state of the game level. By acknowledging received snapshots, the server can know which player has received which snapshots, which can then be used to try and send less data, based on calculating delta packets, and by compacting the data being sent by trying to cut up data fields into as little data as possible - perhaps only a single bit for a single **bool**.

While the mechanism of sending snapshots will help fix network delays and network data loss, it can't by itself solve the problem of latency in FPS games. If a client is far away from a server, their version of the game will be delayed to the 'true' state of the server, and thus their version of the game will also be out of date - not very good for trying to aim at opponents! The mechanisms we've seen here can be used for the basis of something that can help though - if a server is storing multiple versions of the game world as a historical set of snapshots, it will know what any particular client should have been able to see at some earlier point in time, and from there determine whether shots fired by a player at that time would look like they would hit, and try and reconcile the two states. There's a lot more to networking than just efficient data transfer, as client and server-side prediction of state is also very important.