Networking - Networking Protocols

Introduction

Nearly every game released has some sort of internet connectivity. From online accounts and achievements stored in the cloud, to multiplayer games where players fight with and against each other to win, you'll be hard pressed to find a completely 'offline' game released in the modern age. In this tutorial, we'll be taking a look at the basics of how computers connect to each other, and see how to send and receive the data required to enable common online functionality in our games.

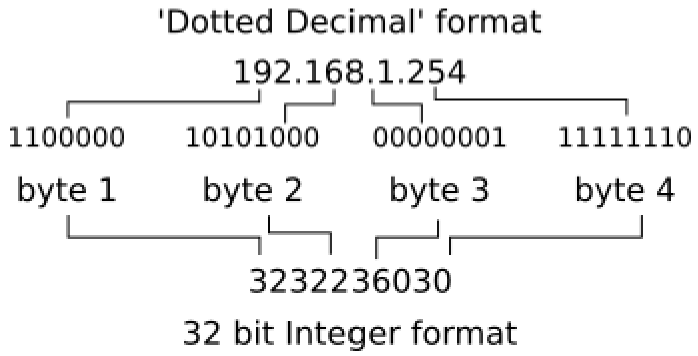
Internet Protocol Suite

For two computers to communicate, they need a well defined set of rules to follow, called a *protocol*. By following the steps of the protocol, a computer knows what data to send, when to send it, and when to wait for an answer. Knowing what data to send is important - one computer 'talking' in one protocol can't understand another that 'talks' in another protocol, just as someone speaking English may not know what to do if they receive an answer in German.

While there have been a number of protocols used for computer networking over the years, by far the most popular is the *Internet Protocol*, or simply *IP* for short. The Internet Protocol was developed in the mid 70s onwards, and it is with this protocol that you were able to navigate the University website and download this tutorial. The Internet Protocol has a set of rules that must be adhered to for successful communication, and a set of concepts that any implementation of IP must contain.

IP Address

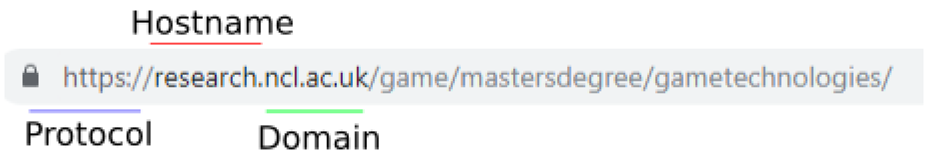
The most obvious part of the IP protocol that you will have come across is an IP *address*. This number identifies you on the internet, and allows network traffic to be sent your way, just as your house having an address lets you receive mail. The most common form of IP address is the IPv4 address, which is made up of 4 bytes, to be interpreted as numbers between 0 and 255. Here's an example of an IP address:



When typing in IP addresses, we usually do so in 'dotted decimal' format, with 4 numbers between 0 and 255, separated by a dot. This adds up to 32 bits, and so internally within a program, the IP address might be stored as a single 32 bit unsigned integer, but as the example above shows, this can quickly become a large, hard to remember number, so dotted decimal is the standard when the IP address needs to be human readable.

Host names

When we connect to another computer on the internet in our web browser, we *usually* don't type in an IP address, but instead an easy to remember URL. This URL lets us know which computer we want to communicate with, and how. As part of this URL, there is a definition of a *host name*:

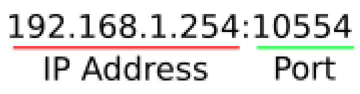

Hostname
https://research.ncl.ac.uk/game/mastersdegree/gametechnologies/
Protocol
Domain

To transform these host names into usable IP addresses, an internet provider will give us access to a DNS server - short for *Domain Name System*. This DNS server acts like a database, converting hostnames to IP addresses - as long as we know the IP address of a DNS, we can ask it to provide us with the IP address of a given hostname, which the computer can then start talking to. This is an example of a protocol *within* a protocol - there is a strict set of operations that both the client (our computer in this example) and the server (the DNS server) must adhere to in order to correctly send and receive DNS 'lookups', giving us a DNS protocol, that is communicated via the IP protocol.

There's a special host name that you might sometimes need (and we'll be using it in this part of the tutorial series) called *localhost*. This always equates to the IP address 127.0.0.1, and this IP address is itself a special address that means 'this computer'. Sometimes, we might be running a server just on our local machine, that another program on the same machine needs to talk to, via the TCP/IP protocol, so always being able to use *localhost* is useful, as we don't have to remember any IP addresses at all (either the 'magic' IP of 127.0.0.1, or the IP address of the network port). You may remember having to provide network access to the **nVidia Nsight** program - this was because the Visual Studio plugin 'talked' to the Nsight-enabled graphics driver through *localhost* - writing the information transfer in this way makes it then easy to redirect the messages to another IP, and allow one machine to receive Nsight messages from an entirely different machine, without needing any 'new' code to be done.

Ports

If computers can talk to each other via IP, how does a particular program know that a particular message is meant for it? Perhaps the server is running multiple different services, all of which have different messages. How to determine which program gets which messages? Along with an IP address, the exact routing of a message to a particular program also requires a *port* number. This is a 16 bit integer that defines a specific endpoint for communication between two programs, and allows multiple different network-enabled programs to be ran at the same time. There are a set of standard ports that certain types of network traffic will use to allow easy communication - the DNS server mentioned earlier can be contacted by sending packets to port 53, while the traffic generated by loading up websites passes through port 80. Games, too, will define ports to specify communication between players, or servers - id Software's *Doom*, for example, will use port 666 by default. The port between a server and a client doesn't necessarily have to match up - traffic could be sent to the server on one port, with the client receiving it on another. Therefore, when looking at the server browser in some games, or definining a direct IP address to connect to, you might actually see the address formatted as follows, with a colon delimiting the IP address and port:



Sending and receiving data

While IP provides us with the means to connect to another computer, and route messages to specific programs using ports, it doesn't strictly define the contents of those messages, or their format. Instead, much as with DNS, we end up with more protocols within protocols which define these message types. The two messaging types commonly used by games are the *Transmission Control Protocol* (often just called *TCP*, or sometimes *TCP/IP*), and the *User Datagram Protocol* (or *UDP* for short). The data within these protocols is sent from client to server via a series of *packets*, which as their name suggests, contain a small amount of information, plus additional information around it relating to how that data should be parsed by the protocol sending and receiving it. Packets are how the *Internet Protocol* splits up data to route it through the internet from the host device to its destination. Within the chunks of data defined within a packet, the protocols themselves have data structures - TCP has *segments*, while UDP has *datagrams*. This means that a single TCP packet could contain only part of a message from the underlying protocols; we don't *really* have to care about this bit, as it's the operating system's job to make sure that IP packets make their way to the correct application in their entirety, and present the program with completed messages.

TCP

The more common of the two protocols mentioned is TCP, as this is how your computer asks a website for information, and how that information is then sent to your computer for display. Each segment of data sent by TCP is of varying size, but will always have a specific structure at the start of it which defines properties about that packet. We can model it as a **struct** in C++ like so:

```cpp
struct TCP_Header {
    unsigned short srcPort;
    unsigned short destPort;
    unsigned int   sequenceNumber;
    unsigned int   acknowledgeNumber;
    unsigned short flags;
    unsigned short windowSize;
    unsigned short checksum;
    unsigned short urgentPointer;
    unsigned char  optionsData[128];
};

struct TCP_Packet {
    TCP_Header header;
    char data[WINDOW_SIZE]; //defined by header
};
```

Conceptual TCP packet structure

As games programmers we don't actually directly handle TCP packets in this way (so we don't have to actually *make* this struct anywhere!), we just get the data; your network driver handles the actual header and decides what to do to keep the communication working.

What we can see from this is that every time we want to send or receive some data via TCP, we need to also transmit up to 224 additional bytes! That *optionsData* segment will usually not be the full 128 bytes, with the actual size being contained within the flags property, so a receiver of a TCP packet will always know how much data is 'header', and how much is 'real data'. The first two properties of the header allow the network handler to see which port the packet should be delivered to (so the right program gets the data), and which port it came from (so the receiving program knows what type of data it is receiving). Jumping ahead in the struct a bit, we can see that every TCP packet has a checksum - the data to be sent is *hashed*, providing a number that can in some way be recalculated from the data by the receiver - if the receiver hashes the data and gets a different checksum, then the data is in some way corrupted, and must be resent.

To understand why we have certain other bits of data in that header, we should probably start to think about an example of sending some data using TCP. When we download a file from a website, we do so using TCP packets, so let's imagine we have been given a link to a 1 megabyte file, and wish to download it. A small TCP packet is sent to the server, pretty much saying 'can you start sending me the file, please?', to which the server will respond by sending packets of data back to the same port it received the request on. You won't receive a single packet with 1MiB worth of file inside it, though, but a series of packets - each TCP segment can have up to 65,355 bytes of data inside it, determined by the *windowSize* variable in our theoretical packet structure above. When the receiving machine sends TCP packets, it fills this *windowSize* variable to let the sending machine know how much data it can successfully receive at a time; it takes storage and processing to handle the incoming data, so this window size lets both ends of the connection know how much data can be processed, and not to send to much data at once.

So does that mean we will receive 16 or more packets, each with a chunk of the total file? Probably not! The internet is an unreliable place, where the routing of the packets being sent around the world might sometimes go wrong in a number of ways, including a packet being 'lost' altogether. Packet loss means having to send more packets to complete a file, and more packets means more wasted bandwidth sending those headers. To combat this, a download via will start with small packet sizes, and as packets are successfully received, the packet size will increase, as the connection is temporarily 'trusted' more to handle the data being sent.

A TCP connection won't keep blindly sending packets - the receiver must acknowledge their successful arrival by sending an 'ack' packet back, or the server will stop and wait, in case their is a problem with the connection. This is known as a 'sliding window' - if packets are being successfully sent by the server, the window size increases, allowing more packets to be sent without stopping and waiting for an 'ack' from the client, and each packet will in addition start to have more data within it. If at some point during transmission the client does not sent an ack packet, the sliding window will shrink down, and less packets will be sent (so as to avoid wasting the processing of them), and the packets will have less data within them. This 'sliding window' effect is why sometimes when downloading a file, the download speed seems to increase over time - as the connection becomes more trusted, the sliding window opens larger, and so more data can be sent for every acknowledgement packet, reducing the bandwidth being 'wasted' on those TCP packets.

As packets are sent across the network, they may take different paths to get from the server to the client - there's no guaranteed ordering, so a packet containing the end of our 1MiB file might reach us before the first packet. To detect this, packets have a sequence number, identifying which bit of a transfer they are. If the client receives part 'B' before part 'A', it will sit and wait for part 'A', and after a timeout period, will re-ask the server for a packet matching the expected sequence number of part 'A'. Only when all of the expected packets have been successfully received will a client then ask for part 'C'. This means that the client program itself will seemingly never receive packets out of order - the OS will 'hide' them until the correct order can be arranged. We don't have to manually program this behaviour ourselves, as it is part of the networking stack of your operating system, but we do have to be aware of some side-effects of this behaviour, which we'll cover shortly.

UDP

The other popular IP data protocol is UDP. It's a much simpler protocol than TCP, which has both advantages and disadvantages. To get a feel for what UDP can and cannot do, here's the logical header for the protocol:

```
struct UDP_Header {
   unsigned short srcPort;
   unsigned short destPort;
   unsigned short dataLength;
   unsigned short checksum;
};

struct UDP_Packet {
   UDP_Header header;
   char data[DATA_LENGTH_SIZE]; //Actual size defined by header
};
```

Conceptual UDP packet structure

The header is *much* smaller! It has a source and destination port for the same reasons as TCP, along with a checksum for validity checking. But that's pretty much it - there's no concept of an ordering of UDP packets, and no mechanism to prevent packet B from appearing at the client before packet A. What a UDP packet can do, though, is contain up to 65535 bytes of data - there's no sliding window rate limiting the download speed or anything, it's just a simple 'fire and forget' data structure, known as a *datagram*. UDP is used when small messages are required as fast as possible from a client - the DNS system mentioned earlier uses UDP, as the packet the client sends only needs a small amount of data for the hostname (smaller hostnames are more human readable, so making them more efficient in UDP, too!), and the data the server sends back is only 4 bytes of an IP address.

If, for whatever reason, the DNS server in the example above doesn't ever send anything back (or the packet gets lost along the way), the client can just send another UDP packet after a timeout and ask again - the packet is much smaller, and therefore more efficient to process.

It gets harder to use UDP for larger scenarios, though. If we wanted to send 1MiB of data using UDP, the program receiving it would have to just 'guess' that the packets it receives were in the correct order, and did not contain any errors, and that none were missing; UDP simply does not contain the data for making any informed decisions about what it is receiving, only that there's a packet of data, that it should forward on to a program listening on a particular port. Anything beyond that must be encoded within the datagram's data when it is created.

TCP vs UDP

So far, it sounds like TCP is the clearly superior option for transmitting data - as far as the program is concerned, it provides packets reliably, and in-order, so a transmission *will* get to its target eventually. That 'eventually' clause is actually something of a problem for a certain type of application, though - real-time, dynamically changing programs such as games. In an FPS, we don't care if we didn't get a packet containing an object's position 10 seconds ago, if we did get a packet containing an object's position 5 seconds ago - the old data has become pointless, and waiting for it is wasteful. There's no real way to state this relationship between data packets in the TCP model, once one end of the connection has decided that some data should be sent, it will either successfully send, or the connection will be terminated. The sliding window of TCP proves to be a problem, too. The window cannot progress to the next set of packets until the previous ones have successfully been acknowledge - or to put it in terms of our example, the packet of our object's current position might not be able to be transmitted at all, until the packet containing its position 10 seconds ago has been acknowledged. If that packet is *never* acknowledged (some unfortunate packet loss affecting only that packet?), the new position will never be sent, and that object will never update correctly.

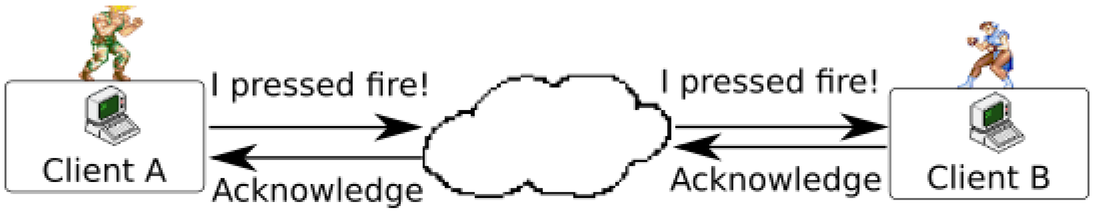
Due to this, it's uncommon for a real-time game to use TCP for transmission of game state. It might be fine (and maybe even desirable) to send a whole turn's worth of data in a turn-based game in TCP packets, but for a game like an RTS or FPS, the game needs to know what is going on as fast as possible. That's not to say that TCP is *never* used though. Certain information *does* need to be reliable, and is not as time sensitive. Some games will have online accounts that the player logs on to, which will record stats such as achievements unlocked, winner and loser of a match, and so on. Data like this can be repeatedly sent 'until it works' - a spinning 'logging in' animation is worth it to maintain a consistent game state - if a combination of packet loss and corrupted packets means that the server means that *both* players have won a 1v1 game, then the stats information cannot be relied upon. Also, just because UDP doesn't contain acknowledgements or sequence numbers in its header, doesn't mean that the game can't encode such things in the data being sent within a UDP packet. It's just that now the responsibility for what data is required for the consistency of the game is down to the network programmer of the game (that's **you** for the next couple of tutorials!).

Networking in games

Different games use TCP or UDP in different ways in which to create their multiplayer experience. We'll be looking at the specifics of two popular implementations of an RTS and an FPS in a later tutorial, but we can at least look at the broad differences between the two most common methods, *peer to peer*, and *client / server.*

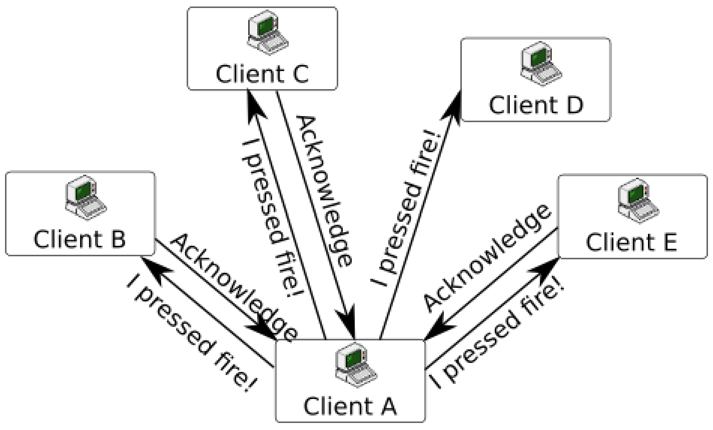
Peer to Peer

Many games allow direct connection to the machines of the other players you are playing with / against. This is known as a peer to peer connection, meaning that every connected player is as important as the others when it comes to who should send or receive information. An example of a peer to peer connected game would be a '1v1' type game, like *Street Fighter V*, or *StarCraft*. Both players send packets directly to one another, describing the actions that they are performing:



A peer to peer model might still contain specific acknowledgement packets so that each player knows they are up-to-date, but this is now game-specific, and entirely programmed within the game, with no help from UDP.

One of the problems with this model is *scalability*. The more players are connected in the same gameplay session, the more times the same data must be sent to everyone:

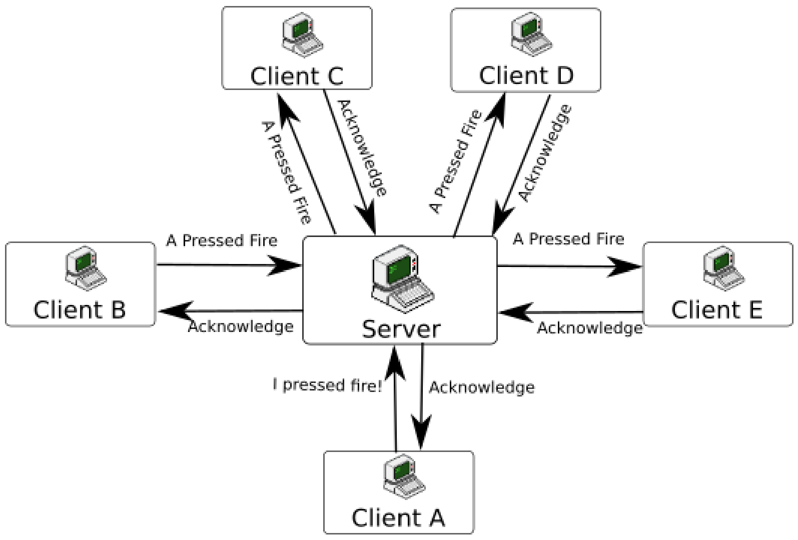


As most home connections are *asymettric*, in that they have far more download capacity than they have upload, this can mean that in larger games, the ability to send messages to the other players runs out before the ability to receive them. In extreme cases, this could lead to the player with the 'best' upload speed having an advantage over other players - as the players on constrained connections are having to queue up messages to send due to the limited bandwidth to send them, the player with the best connection can just send messages straight away, which the other players can then still receive (due to the asymmetric connection), and their in-game character gets defeated before they can react.

In this particular case, client *D* never acknowledged that client *A* pressed fire - if *D* never received the packet, that means on their screen, player A isn't pressing 'fire', and the game breaks down (everyone else might think that *A* shot *D*, but *D* thinks they are still moving through the world). We'll be covering game states in more detail in another tutorial, but for now, know that if you've ever been told that the game has 'desynchronised' and kicked you back to the main menu in an RTS game, this is probably at least partially why.

Client / server

An alternate to the peer to peer model is to instead have a dedicated server machine, which arbitrates the communication between all of the clients playing the game. In such a model, each client only has to send and receive data from one point, making for a far more scalable solution:



Generally, the server runs a special copy of the game that does not include any graphics, or take in any mouse / keyboard input. Instead, it just processes the data coming in from players, works out a new game state based upon it, and then lets the players know what the new state should be. Servers can be placed in datacentres with large amounts of bandwidth (or even spawned when necessary from cloud service providers), so the download speed of the players becomes the only contention - if there's so much going on in the world that they can't receive new game states quick enough, they'll still end up with a 'laggy' experience. Usually games that provide a client/server model for their multiplayer component come with some sort of central server, which the game connects to on startup. From there, a list of active games being played can be transmitted to players (possible via TCP, as outlined earlier), and the player can then select a server to join.

In some games, one of the player's machines could be *both* a client **and** a server - they both receive everyone else's data, and transmit it to everyone else, but client *B* and client *C* no longer have to directly talk to each other, only the server. This is beneficial to game publishers, in that they don't have to provide dedicated servers for the players to connect to. However, there are side-effects to such a model. What if the player chosen to be the server has bad upload bandwidth? If the data transmission is greater than their usable bandwidth, now *everyone* will have a bad time, waiting for the server to transmit new information. The game state will be *consistent* (everyone will only do things when the server says so), but it'll also feel *laggy*, or time appears to move slower than it should.

Tutorial Code

We now know (very broadly speaking) what the difference between TCP and UDP is, what data is contained within a packet, and how a server differs from a client. What we *don't* really know though, is how to actually program any of the data transmission we need between endpoints in a game. For this first tutorial, we're going to encode some simple messages, and use UDP to send them out of one port, and receive them on another. In a later networking tutorial we'll see how to use this basic functionality to send gamestate information to allow an actual game to be played.

eNet Networking Library

To send UDP packets, we're going to rely upon a 3rd part library called *eNet*. This lets us quickly set up a UDP connection, as either a server (something that will have many incoming connections from a number of different sources) or a client (something with a single end point for connection). Strictly speaking, *eNet* gives us some hidden extra functionality to make UDP a little more reliable, but for the purposes of seeing how networking works, we're going to ignore this, and assume that our UDP packets *could* go missing as they move from point to point through our network - if packets were to go missing on their way to *localhost*, you'd probably have bigger problems than a laggy game, though!

In this tutorial, we're going to take a look at 3 classes designed to help us make networked games. To encapsulate the workings of a game client, and a game server, we have the **GameServer** and **GameClient** classes, and these both derive from a **NetworkingBase** class, containing some of the workings common to both.

GamePacket struct and PacketReceiver Class

The job of a networking system in a game is to handle *packets* - either sending or receiving them. The eNet library contains methods for handling the sending and receiving of data, but we're also going to look at ways in which we can get the right parts of the networking system getting those packets and operating on them - an entirely different section of code in your program might handle the receiving of a text message from an opponent than handles the receiving of a set of leaderboard entries to look at after the game, for instance. To encapsulate this concept of objects in our code handling packets, we're going to create a **GamePacket** struct to represent the data being sent over the network, and a **PacketReceiver** class to handle any packets received from an endpoint in eNet.

Here's how the provided **GamePacket** struct looks (declared in NetworkBase.h):

```
struct GamePacket {
   short size;
   short type;

   GamePacket() {
      type     = BasicNetworkMessages::None;
      size     = 0;
   }
   GamePacket(short type) {
      this->type   = type;
   }
   int GetTotalSize() {
      return sizeof(type) + sizeof(size) + size;
   }
};
```

GamePacket struct header

We're going to assume that every type of data we want to send in our game is of a specific *type*, and of a specific *size*. For reasons that we'll get to later on, this size *might* not match up to a **sizeof** on a **struct**, and so there's a provided *GetTotalSize* method to work out how many bytes a packet is made up out of.

As packets are received by either a server or a client, they must be redirected to the correct piece of code within the program for further processing. To enable this, both clients and servers will maintain a list of **PacketReceivers** - classes that have a **virtual** *ReceivePacket* method, allowing for the correct systems within a game to receive data and work on it. Here's how this is represented in the codebase:

```
class PacketReceiver {
public:
   virtual void ReceivePacket(int type,
                     GamePacket* payload, int source = -1) = 0;
};
```

NetworkBase class header

Perhaps the smallest class you've ever seen! We'll see how to derive a subclass from this later on, with an overridden *ReceivePacket* method allowing for processing of sent packets.

NetworkBase Class

Both the **GameClient** and **GameServer** classes derive from **NetworkBase**, which contains methods for initialising the eNet library, and routing packets received from the network port to the **PacketReceivers** that care about those packets. Here's how it looks:

```cpp
class NetworkBase {
public:
   static void Initialise();
   static void Destroy();

   static int GetDefaultPort() {return 1234;}

   void RegisterPacketHandler(int msgID, PacketReceiver* receiver) {
      packetHandlers.insert(std::make_pair(msgID, receiver));
   }
```

NetworkBase class header

```cpp
protected:
   NetworkBase();
   ~NetworkBase();

   bool ProcessPacket(GamePacket* p, int peerID = -1);

   typedef std::multimap<int, PacketReceiver*>::const_iterator
         PacketHandlerIterator;

   bool GetPacketHandlers(int msgID, PacketHandlerIterator& first,
                  PacketHandlerIterator& last) const {
      auto range = packetHandlers.equal_range(msgID);

      if (range.first == packetHandlers.end()) {
         return false; //no handlers for this message type!
      }
      first = range.first;
      last  = range.second;
      return true;
   }

   ENetHost* netHandle;
   std::multimap<int, PacketReceiver*> packetHandlers;
};
```

NetworkBase class header

Our classes store a **std::multimap** of **PacketHandlers** - this is a special type of map that allows for a number of objects to be mapped to a single *key*; we're going to use this to allow for multiple objects in our code to potentially be interested in a certain packet *type* integer. Along with this, and the methods for adding new **PacketHandlers**, and getting iterators out of the multimap for using when sending packets to objects, there's not a lot else - the static *Initialise* and *Destroy* methods handle the startup and teardown of the eNet library. As both clients and servers deal with the processing of packets, we also have a **ProcessPacket** method, which looks like this:

```cpp
 1 bool NetworkBase::ProcessPacket(GamePacket* packet, int peerID) {
 2    PacketHandlerIterator firstHandler;
 3    PacketHandlerIterator lastHandler;
 4
 5    bool canHandle = GetPacketHandlers(packet->type,
 6                     firstHandler, lastHandler);
 7    if (canHandle) {
 8       for (auto i = firstHandler; i != lastHandler; ++i) {
 9          i->second->ReceivePacket(packet->type, packet, peerID);
10       }
11       return true;
12    }
13    std::cout << __FUNCTION__ << " no handler for packet type "
14              << packet->type << std::endl;
15    return false;
16 }
```

NetworkBase::ProcessPacket method

It uses the *GetPacketHandlers* method to fill in a couple of iterators, and then tries to send the packet to all of the registered **PacketReceivers** for a particular packet type by following through those iterators (line 8). Nothing else needs to be done, the actual logic of which packet is which, and what it should do, is entirely down to the game code which receives the packet - our networking classes otherwise don't know (or care!) what these packets are.

GameClient Class

We'll now take a look at how to create a game client - this encapsulates the process of sending and receiving packets to a single end point, and most closely matches to the types of processing that the average network-enabled game. The *Connect* method of the **GameClient** class takes in 5 numbers - 4 are the 'dotted decimal' format of the IP address to connect to, and one is the *port number* - for our computer's network stack to route network packets to the right end point, we need one of these to uniquely identify our game's network traffic. While dotted decimal is nice to look at, it's also common for an IP address to be stored as a single integer, and this is how eNet stores addresses, so on line 5 of the method, we use some bit shifting, to move each of the address bytes a multiple of 8 bits across, so they can be logical ORd together. Once that's done, we let eNet know where we want to connect, and hopefully receive a pointer to an **ENetHost** - this is our 'handle' into using eNet, similarly to how we got a context from OpenGL.

```cpp
 1 bool GameClient::Connect(uint8_t a, uint8_t b,
 2                          uint8_t c, uint8_t d, int portNum) {
 3    ENetAddress address;
 4    address.port = portNum;
 5    address.host = (d << 24) | (c << 16) | (b << 8) | (a);
 6
 7    netPeer = enet_host_connect(netHandle, &address, 2, 0);
 8
 9    return netPeer != nullptr;
10 }
```

GameClient::Connect() method

In many ways a client is the easier of the two end points to program, as its traffic can only go to or come from a single place - the server connected to. To encapsulate sending of packets from a client to a server, then, we only need one function - the *SendPacket* method takes in a reference to our **GamePacket** struct, and then uses the *enet_peer_send* method to try and transmit it to our server - note that the first parameter is the *netPeer* variable that we obtained from eNet when connecting to the server.

```cpp
void GameClient::SendPacket(GamePacket&  payload) {
   ENetPacket* dataPacket = enet_packet_create(&payload,
                            payload.GetTotalSize(), 0);
   enet_peer_send(netPeer, 0, dataPacket);
}
```

GameClient::SendPacket method definition

Sending data is one thing, but what about *receiving*? The eNet library works by generating a series of events onto a queue, which can be popped off whenever convenient by the client. To see this in action, let's look at the *UpdateClient* method. On line 7 we have a while loop, which tries to pop off an event from eNet's internal queue, which if successful will fill up the event variable on line 6. These events have a type (which is **not** the same as the *type* variable in the **GamePacket** struct!), which can then be tested against. So far in our client we only care about two types of eNet event - a successful connection to the server (handled on line 8), and more importantly, receiving a packet from the server (handled on line 11). On a packet event, the event variable will have a filled in *data* variable, pointing at a set of bytes; eNet has no idea what these bytes are, only that they were transmitted via UDP on the correct port. However, as long as *we* know what that data is, we can safely cast it to any other type! As the only data we ever send is a subclass of **GamePacket**, we can always cast the data to a pointer to a **GamePacket** (line 13), and then send it to the *ProcessPacket* method of the **NetworkBase** class.

```cpp
 1 void GameClient::UpdateClient() {
 2    if (netHandle == nullptr)  {
 3       return;
 4    }
 5    //Handle all incoming packets
 6    ENetEvent event;
 7    while (enet_host_service(netHandle, &event, 0) > 0)   {
 8       if (event.type == ENET_EVENT_TYPE_CONNECT) {
 9          std::cout << "Connected to server!" << std::endl;
10       }
11       else if (event.type == ENET_EVENT_TYPE_RECEIVE) {
12          std::cout << "Client: Packet recieved..." << std::endl;
13          GamePacket* packet = (GamePacket*)event.packet->data;
14          ProcessPacket(packet);
15       }
16       enet_packet_destroy(event.packet);
17    }
18 }
```

GameClient::UpdateClient() method

Once the packet has been handled by a function, we're safe to then discard any packet data (line 16) so that we don't leak any memory.

For now at least, that's all we need to send and receive packets! As it was with state machines, we're building up the generic machinery that allows us to then plug in more interesting functionality using sub classes, and virtual functions.

GameServer Class

Creating a server isn't too much more difficult than a client in eNet. The process of setting one up is contained within the **GameServer::Initialise** method, which looks like this:

```cpp
bool GameServer::Initialise() {
   ENetAddress address;
   address.host = ENET_HOST_ANY;
   address.port = port;

   netHandle = enet_host_create(&address, clientMax, 1, 0, 0);

   if (!netHandle) {
      std::cout << __FUNCTION__ <<
         " failed to create network handle!" << std::endl;
      return false;
   }

   return true;
}
```

GameServer::Initialise() method

Not that much! We don't need to specify an IP address this time around, just a port that the server will listen on for incoming connections. If it successfully initialises, we'll get another one of those *netHandle* pointers that eNet uses for communication with the program.

More interesting is the ability to send packets to all of the clients connected to a server; this is wrapped up in two *SendGlobalPacket* methods - one allows the sending of a full **GamePacket** to all clients, and one uses this to allow the sending of just a single integer. This might not sound like much, but it's all that we need to do a lot of the time in networking to tell clients that an important event has occurred - receiving a value of '33' could match up to an enum value stating that the current game round has ended, while another could indicate that a powerup has been picked up somewhere (or any other mechanics you think you might need for your game!).

```cpp
bool GameServer::SendGlobalPacket(int msgID) {
   GamePacket packet;
   packet.type = msgID;
   return SendGlobalPacket(packet);
}

bool GameServer::SendGlobalPacket(GamePacket& packet) {
   ENetPacket* dataPacket = enet_packet_create(&packet,
                     packet.GetTotalSize(), 0);
   enet_host_broadcast(netHandle, 0, dataPacket);
   return true;
}
```

GameServer::Initialise() method

This should look pretty familiar to you, as its very similar to the **GameClient::SendPacket** method, but with a new method being used - *enet_host_broadcast* sends a copy of the packet to every client connected to the server.

As with clients, the eNet provides servers with a message queue, allowing it to inform the server application when players connect and disconnect, and when packets have been received. To handle the processing of this queue, there's an *UpdateServer* method, which is pretty similar to the *Update-Client* method from earlier:

```
 1 void GameServer::UpdateServer() {
 2    if (!netHandle) { return;  }
 3    ENetEvent event;
 4    while (enet_host_service(netHandle, &event, 0) > 0)   {
 5       int type    = event.type;
 6       ENetPeer* p = event.peer;
 7       int peer    = p->incomingPeerID;
 8
 9       if (type == ENetEventType::ENET_EVENT_TYPE_CONNECT) {
10          std::cout << "Server: New client connected" << std::endl;
11       }
12       else if (type == ENetEventType::ENET_EVENT_TYPE_DISCONNECT) {
13          std::cout << "Server: A client has disconnected" <<std::endl;
14       }
15       else if (type == ENetEventType::ENET_EVENT_TYPE_RECEIVE) {
16          GamePacket* packet = (GamePacket*)event.packet->data;
17          ProcessPacket(packet, peer);
18       }
19       enet_packet_destroy(event.packet);
20    }
21 }
```

GameServer::UpdateServer() method

Unlike with clients, where it will always be the server send the message, we don't immediately know which client sent a data packet to the server. The eNet library does, though, and lets us obtain an integer to identify which client sent the packet (line 7), which can then be sent to the *ProcessPacket* method - this way the server will know which player fired the rocket / took damage / etc / and can run code on the correct internal class representing this. That's it for our simple game server! At least for now - we'll be taking a look at how to structure data transfer for unreliable connections in another tutorial.

A simple client / server example

So far, we've just been implementing the base mechanics of a networked application - how do we actually send data from one point to another, and what should that data be? To see networking in action, we're going to expand the provided game code to create a client and a server communicating on localhost, and use the connection to send some strings from client to server, and from server to client. To do so we're going to need a new packet type, so define the following class inside the **NetworkBase.h** file, below the declaration of the **GamePacket** struct:

```
struct StringPacket : public GamePacket {
   char  stringData[256];

   StringPacket(const std::string& message) {
      type     = BasicNetworkMessages::String_Message;
      size     = (short)message.length();

      memcpy(stringData, message.data(), size);
   };

   std::string GetStringFromData() {
      std::string realString(stringData);
      realString.resize(size);
      return realString;
   }
};
```

StringPacket struct header

We're assuming we can send up to 256 characters of string data, and using the constructor to copy a source **std::string** data into this array. You might be wondering why we aren't just storing a **std::string** in the packet, as that's the 'natural' way to keep them in C++. The answer is all down to memory - a **std::string** stores a pointer to the actual text character bytes, so they are somewhere else entirely in memory. If we transmit a **std::string** from point A to point B, point B receives an invalid pointer! It's much nicer to instead have a single contiguous set of bytes to give to the networking API for transferring to another machine, so by declaring an array of bytes right there in the struct, they will be part of the same block of memory as the *type* and *size* variables of the **GamePacket** parent class. This also finally reveals why the **GamePacket** class has a size variable! We might not always want to send all 256 bytes of our **StringPacket**, as this is wasteful (more bytes = more bandwidth = more processing time = more latency!), so we can instead use the size variable to make the **StringPacket** struct appear to be slightly smaller than it really is when it comes to actually transmitting it - as the 'real' size is also sent, its easy to convert the bytes back into a string later on once the packet has been received using the *GetStringFromData* method, which returns the bytes back into the 'easy' **std::string** form.

Main File

Now to actually handle the sending, receiving, and handling of the new **StringPacket** struct. In the main file, above the currently empty *TestNetworking* method, add in the following new class declaration:

```cpp
class TestPacketReceiver : public PacketReceiver {
public:
   TestPacketReceiver(string name) {
      this->name = name;
   }

   void ReceivePacket(int type, GamePacket* payload, int source) {
      if (type == String_Message) {
         StringPacket* realPacket = (StringPacket*)payload;

         string msg = realPacket->GetStringFromData();

         std::cout <<name<< " received message: " << msg <<std::endl;
      }
   }
protected:
   string name;
};
```

StringPacket struct header

It derives from the **PacketReceiver** class that the **GameClient** and **GameServer** use to handle incoming packets, and overrides the *ReceivePacket* method. In its method, it checks against the incoming message type (perhaps a **PacketReceiver** cares about a number of different packet types?), and if its of type **String_Message**, can cast the received **GamePacket** to a **StringPacket**, safe in the knowledge that it is actually of that type - from there it can get a string out of it, and send it to the console. The class also has a *name* variable, which we can use to simulate either a client or a server receiving this particular type of packet in the console output.

Time to fill in the last method for this tutorial, the empty **TestNetworking** method above the main file inside the **main.cpp** file. This method is going to instantiate a **GameClient**, a **GameServer**, and a couple of **TestPacketReceivers**, and connect them together, to create some very simple message passing over a network connection. Add the following code into the *TestNetworking* method:

```
 1 void TestNetworking() {
 2    NetworkBase::Initialise();
 3
 4    TestPacketReceiver serverReceiver("Server");
 5    TestPacketReceiver clientReceiver("Client");
 6
 7    int port = NetworkBase::GetDefaultPort();
 8
 9    GameServer* server = new GameServer(port, 1);
10    GameClient* client = new GameClient();
11
12    server->RegisterPacketHandler(String_Message, &serverReceiver);
13    client->RegisterPacketHandler(String_Message, &clientReceiver);
14
15    bool canConnect = client->Connect(127, 0, 0, 1, port);
16
17    for (int i = 0; i < 100; ++i) {
18       server->SendGlobalPacket(
19          StringPacket("Server says hello! " + std::to_string(i)));
20
21       client->SendPacket(
22          StringPacket("Client says hello! " + std::to_string(i)));
23
24       server->UpdateServer();
25       client->UpdateClient();
26
27       std::this_thread::sleep_for(std::chrono::milliseconds(10));
28    }
29
30    NetworkBase::Destroy();
31 }
```

TestNetworking function

Before any eNet functions can be called, we need to call the *Initialise* method of **NetworkBase**. Creating a server and a client is then as simple as just instantiating them (lines 9-10), and telling them to redirect string packets to a specific object (the **TestPacketReceivers**, send to the client and server on line s12 and 13). We can then tell the client to connect to the server, via the special *localhost* IP address on line 15. We can then simulate some string passing by repeatedly telling the server to send a global packet, and the client to send a packet to the server; as long as we keep calling the *Update* methods for the client and server, they should receive those messages, and pass the packet on to the **TestPacketReceivers**, which will then output the string to the console. Once we're done we just close down the eNet library with the *Destroy* method.

Conclusion

If you call the new *TestNetworking* function from within the main function somewhere, you should hopefully be able to see some messages in the console window! Sending strings backwards and forwards is only the very start of network programming for games, but its enough to let us see the fundamental concepts behind it. We've seen how computers communicate via *IP addresses*, with unique communications between addresses being identified via a *port*. We've seen a little bit on the differences between *UDP* and *TCP*, and how they allow the transmission of *packets* from one machine to another. While we do now have enough to allow communication of arbitrary packets of data, and can therefore allow the server to send gameplay information to clients, there's still more to networking yet, and in the next networking tutorial, we'll be taking a look at how to represent game state in RTS and FPS games, and how to start making intelligent decisions on what data should and should not be sent over the network, and how to get it to influence the world of the games that you make.

Further Work

1) Try making multiple clients connecting to the same server in the *TestNetworking* method - you should just need a new **TestPacketReceiver**, and a new **GameClient**. Don't forget to *Update* the new client, and set the server up to allow multiple clients!

2) The **GameServer** currently cannot send packets to just an individual client. Try making a new **GameServer** method SendPacketToPeer, that takes in an integer, and works out the correct eNetPeer to include in a call to the **enet_peer_send** method. You may need to modify the **UpdateServer** method so that it can keep track of incoming connections, and terminated connections.

3) Try making some new subclasses of **GamePacket**, and make some handlers for them. A good one to test would be a 'CalculatorPacket' that has 2 integers and an operation byte (to represent subtraction, addition, multiplication, etc). A server receiving this should calculate the answer, and return a *CalculatorAnswerPacket* to the client.

4) Moving beyond this, maybe a *TestSumPacket*, which has a varying amount of data - an integer for a count, and then a number of integers, that should be sent to the server, and then the server sums the answer and returns it.