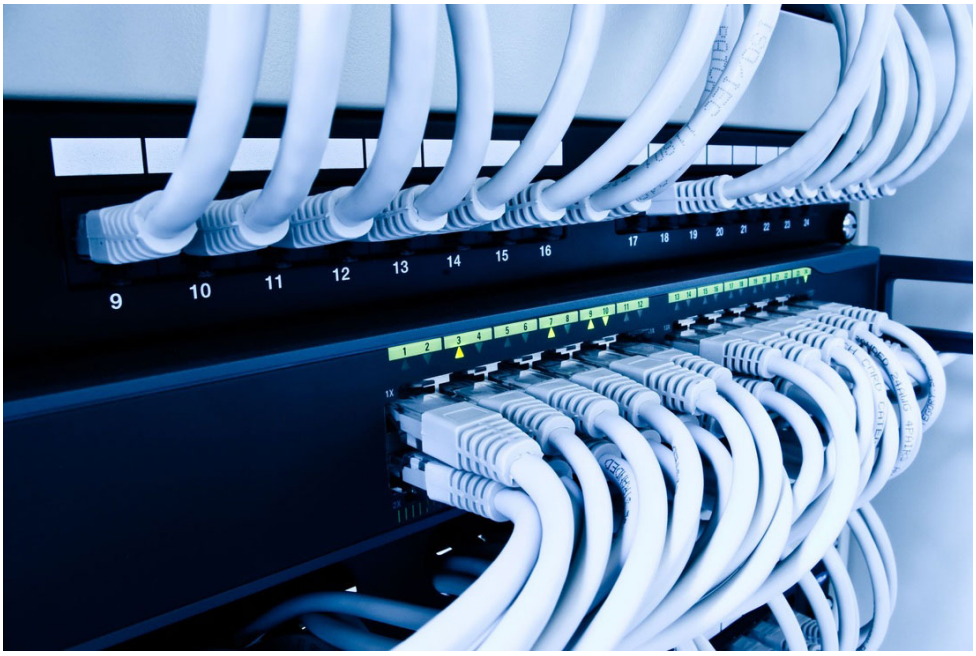
Networking Tutorial 1: Introduction to Networks



Summary

Basic properties of networks are introduced and described, with some reference to available libraries and software. Sockets are presented, along with the concepts of the server and the client. An event-based network library is discussed, and some of its properties are outlined with accompanying code samples.

New Concepts

Sockets, Server and Client, ENet

Introduction

The concept of games, in a general sense, has historically been connected to a multi-player experience. Chess against yourself can be enjoyable, but the game is intended to be played against an opponent. Early computer gaming is fairly unique in that regard, as many PC and console games were a single user experience with the player pit against a virtual opponent (defined by rules and state machines, as discussed in our AI tutorials).

In terms of computer gaming, we can categorise the nature of our multi-player experience by several criteria. Is the game executing in real-time (e.g., a multi-player FPS)? Is the game turn-based (e.g., chess)? Are players sharing a platform, or playing on multiple platforms? Are they geographically co-located, or on opposite sides of the planet?

The simplest form of multi-player gaming, if we assume all players are using the same platform, involves no networking whatsoever - consider playing a racing game in the lounge with a few of your

friends around visiting, or a football management simulator played turn-by-turn. Increasingly, however, games are expected to provision a multi-player experience for players across the internet.

To do that requires an understanding of both the technologies involved in message-passing using a network connection, and the theoretical issues facing any real-time distributed simulation. This tutorial addresses the former, and the second tutorial in this series will address the latter. We begin with discussion of sockets, and the approaches we can take to pass messages between systems, before moving onto the server-client relationship, and present a specific API which provisions event-based message passing from system to system.

Sockets

A socket is a mechanism for allowing communication between processes, be it programs running on the same machine or different computers connected on a network. More specifically, Internet sockets provide a programming interface to the network protocol stack that is managed by the operating system. Using this API, a programmer can quickly initialise a socket and send messages without having to worry about issues such as packet framing or transmission control.

There are a number of different types of sockets available, but we are only really interested in two specific Internet sockets. These are:

- Stream sockets
- Datagram sockets

What differentiates these two socket types is the transport protocol used for data transmission. A stream socket uses the Transmission Control Protocol (TCP) for sending messages. TCP provides an ordered and reliable connection between two hosts. This means that for every message sent, TCP guarantees that the message will arrive at the host in the correct order. This is achieved at the transport layer so the programmer does not have to worry about this, it is all done for you.

A datagram socket uses the User Datagram Protocol (UDP) for sending messages. UDP is a much simpler protocol as it does not provide any of the delivery guarantees that TCP does. Messages, called datagrams, can be sent to another host without requiring any prior communication or a connection having been established. As such, using UDP can lead to lost messages or messages being received out of order. It is assumed that the application can tolerate an occasional lost message or that the application will handle the issue of retransmission.

There are advantages and disadvantages to using either protocol and it will be highly dependant of the application context. For example, when transferring a file you want to ensure that, upon receipt, the file has not become corrupted. TCP will handle all the error checking and guarantee that it will arrive as you sent it. On the other hand, imagine you are sending 1000 messages detailing player position data every second in a computer game. The application will be able to tolerate missing messages here so UDP would be more suitable.

Addressing

Using Internet Sockets, we can identify a host using an IP address and a port number. The IP address uniquely identifies a machine while the port number identifies the application we want to contact at that machine. There are a range of well known port numbers, such as port 80 for HTTP, in the range 0 - 1023. When choosing port numbers, anything above the well know port number range should be fine, but you cannot guarantee that another application will not be already using it.

Programming with Sockets

This section introduces the Windows Sockets API. Note that the sample networking code in the framework conceals this from the programmer, and it is discussed here to aid your understanding of

what happens 'under the hood'.

Windows-Specific Functions

All the methods for declaring and using sockets are available in two header files:

```
#include <winsock2.h>
#include <ws2tcpip.h>
```

Windows Socket Header Files

You will also need to link the `ws2_32.lib` library in your project settings.

The first thing we have to do before being able to declare or use a socket is make a call to the `WSAStartup()` method.

```
int WSAStartup(
   WORD      version,     // highest version of winsock
   WSADATA   *data        // pointer to WSADATA struct
);
```

WSAStartup() function declaration

This method must be called first before any other calls involving the sockets library. This method simply allows the programmer to define the version of the Windows sockets specification to be used and stores the result in the `WSADATA` struct.

Complimentary to `WSAStartup()`, we also have `WSACleanup()` which should be called at the end of your program when you no longer require the use of the socket library.

```
int WSACleanup(void);
```

WSACleanup() function declaration

You will also find `WSAGetLastError()` useful when you need to perform some error checking or debug your code:

```
int WSAGetLastError(void);
```

WSAGetLastError() function declaration

This method takes no parameters and returns the error status for the last socket operation that failed. The range of error codes that can be returned is too extensive to discuss here. You find a detailed list of all return codes and values in the Sockets documentation.

Addressing Information

Having initialised the API we now need to set up the address we will be listening on (if we are the server) or the one we will be sending to (if we are the client). The function `getaddrinfo()` is going to help us out here.

```
int getaddrinfo(
   const char               *name,          // host name or IP
   const char               *service,       // service name or port
   const struct addrinfo    *hints,         // socket info
   struct addrinfo          *result         // result struct
);
```

getaddrinfo() function declaration

We either provide `NULL` to the name parameter if we are creating our server and in the case of the client we want to provide a host name string or IP address. The service parameter takes either a string service name (e.g. http translates to port 80) or the port number. We then have two `addrinfo` structs. We need to populate the `hints` struct with some some option settings before calling the function. The `result` struct stores the results after the function has been executed.

```
typedef struct addrinfo {
    int                 ai_flags;       // socket options
    int                 ai_family;      // address family
    int                 ai_socktype;    // socket type
    int                 ai_protocol;    // protocol type
    size_t              ai_addrlen;     // ai_addr length
    char                *ai_canonname;  // canonical name
    struct sockaddr     *ai_addr;       // pointer to sockaddr struct
    struct addrinfo     *ai_next;       // next addrinfo struct
};
```

addrinfo struct typedef

There are quite a few fields here so we look a little closer at the important ones:

- `ai_flags` - Here we can set some flags to change the socket options. There are a number of options available that you can find in the API. The only one we will need for now is the `AI_PASSIVE` flag. This indicates that the socket we will be creating will be used in a call to the function bind(). We must bind to a socket if we plan on listening for messages, i.e. we are creating a server program.
- `ai_family` - This is the address family we will be using. For IPv4 we would use the flag `AF_INET` and for IPv6 the flag is `AI_INET6`. We do not have to specify the address family if we want to be able to use both versions. The flag for this is `AF_UNSPEC`.
- `ai_socktype` - Heres where we can specify the socket type, stream (`SOCK_STREAM`) or datagram (`SOCK_DGRAM`).

These are the fields that you will want to populate for the `hints` struct. The `results` struct is a linked list as the call to the host may return multiple results. Often only one struct will be returned but you cannot guarantee that the one you want will be the first one if more than one has been found. As such, it is recommended that you traverse the list and check for the one you require, making sure it has been initialised correctly.

Binding a Socket

We have declared our socket but if we plan to receive messages using it then we need to bind to it using the `bind()` function. The purpose of binding is to associate a local address with a socket. The server always needs to call the bind function as it is going to listen for new connection requests and serve them. For clients it is enough to connect to the server without a prior call to bind.

```
int bind(
    SOCKET                  s,          // socket to bind to
    const struct sockaddr   *name,      // local address info
    int                     namelen     // length of *name
);
```

bind() function

Again we have most of the parameters already defined. The socket descriptor comes from our call to `socket()` and the `sockaddr` struct is stored within our `addrinfo` struct that we created with the call to the `getaddrinfo()` function. If no error has occurred, `bind()` returns zero.

Connecting to a Socket

For a client to send messages to a remote host (i.e. the server) we need to connect to that host.

```
int connect(
    SOCKET                  s,          // socket to connect to
    const struct sockaddr   *name,      // remote address info
    int                     namelen     // length of *name
```

```
);
```

connect() function

Notice that the function signature of `connect()` is exactly the same to `bind()`. The difference between the two functions is that `bind()` is used by the server and creates a *local* socket using *local* address information, while `connect()` uses the information of the *remote* host that we want to connect to. If no error has occurred, `connect()` returns zero.

Listening for Connections

For our server, having already specified our address information, created a socket and bound to it, we now need to listen for new connections requests from clients.

```
int listen(
    SOCKET                  s,          // socket to listen on
    int                     backlog     // max. incoming queue length
);
```

listen() function

The `backlog` integer is used to specify the maximum length of the incoming connection queue. A client needs to connect to the server and this will only be successful when the server accepts the connection request. Here we can specify how many connections can be queued waiting to be accepted. This value will vary given the network conditions for the server.

The flag `SOMAXCONN` can be used which instructs the underlying service provided (i.e. the operating system) to set the length of the queue to a some reasonable value. If a client attempts to connect to a server whose backlog queue is full, it will receive a connection refused error. If no error has occurred, `listen()` returns zero.

Accepting Connections

We have told our server to listen on a specific socket for incoming connection requests. When a connection request is received the server to needs to accept it to service the client.

```
SOCKET accept(
    SOCKET                  s,       // socket the server listens on
    struct sockaddr         *addr    // incoming connection details
    int                     *addrlen // length of *addr
);
```

accept() function

Calling `accept()` will cause the first connection on the pending queue to be taken and a new socket created for it. This new socket will handle all interaction between the server and the client that requested the connection. The original socket still exists and is used to service new connection requests.

Sending and Receiving

Now we are at the point where our server has accepted a clients connection request and created a new socket for it to accept and send messages.

```
int send(
    SOCKET          s,       // socket to send with
    const char      *buf     // data we're sending
    int             len      // length of *buf
    int             flags    // sending options - see API
);
```

send() function

```
int recv(
    SOCKET          s,          // socket to receive from
    const char      *buf        // buffer to receive incoming data
    int             len         // length of *buf
    int             flags       // receiving options - see API
);
```

recv() function

The two functions are very similar, essentially mirroring one another. A socket is required to either send to or receive from and then data in the buffer is either sent or data is received into an empty buffer. The available flags can be found in the API and let you alter the actions of the associated function call. For example, we can set a flag to peek at the packet of the queue without actually removing it from the queue.

If we are using datagram sockets we have two different functions for sending and receiving.

```
int sendto(
    SOCKET                  s,          // socket to send with
    const char              *buf        // data we're sending
    int                     len         // length of *buf
    int                     flags       // sending options - see API
    const struct sockaddr   *to         // address of target socket
    int                     tolen       // length of *to
);
```

sendto() function

```
int recvfrom(
    SOCKET                  s,          // socket to receive from
    const char              *buf        // buffer to receive incoming data
    int                     len         // length of *buf
    int                     flags       // receiving options - see API
    const struct sockaddr   *from       // info of client receiving from
    int                     fromlen     // length of *from
);
```

recvfrom() function

Again, these are both very similar to the stream socket functions but we have the addition of the `sockaddr` struct. For `sendto()` the struct contains address information for the host we want to send to. This is used as an input to the function so it has to be constructed prior to the call.

With the `recvfrom()` function, the `sockaddr` struct is used as an output and stores details about the machine that the message was sent from. As we do not create any connections for a datagram socket, we have no information about the host we are receiving from. After receiving a message the details (address, port) are stored in the struct.

Tidying Up

When we are finished with a socket, we need to close it.

```
int closesocket(
    SOCKET   s  // socket we are going to close
);
```

closesocket() function

We provide the socket descriptor of the socket we want to close. This ensures that the socket is closed cleanly and the memory it has used is reclaimed. If any subsequent function calls using a socket descriptor that has been closed will result in a socket error. Also remember that when you has completely finished using the Windows sockets API you want to call the `WSACleanup()` method.

Clients and Servers

The difference between client and server is, largely, contextual, particularly where games are concerned. The server can normally be considered the host of most operations, and the arbiter of conflicting data. In that sense, for example, the server manages the interactions between clients, and resolves differences between their world views.

If we consider the example on an MMO, the server controls NPC AI and, in cases where NPC AI routines are duplicated on the client (more on this in our second tutorial), has authority where two results differ. As an example, if the server says an NPC is moving at velocity `v`, and the client says the NPC is moving at velocity `u`, the server view normally has priority and the client would interpolate towards that result. The same is true if multiple clients all differ from the server view; in this way, we tend towards consistency of perception.

In terms of the actual engineering, the differences generally lie in the fact that the server is expected to receive many connections (one for each client, at least), while the client anticipates only one (its connection to the server). This does not hold true for peer-to-peer architectures, where clients are expected to maintain many connections, but in such cases the engineering differences between client and server reduce significantly (if, indeed, there is a server at all).

What is universally to be expected, however, is that whichever you are writing - client, or server - it should expect to send and receive data messages in some fashion. As such, many of the socket functions discussed above will be present in both your server and client code, if you opt to use the Windows API.

For the purposes of accessibility, the framework includes sample code which abstracts away from this low-level governance of sockets, providing the developer with a transparent means of communicating between two systems. This event-driven library, `ENet`, is the subject of the second half of this tutorial.

ENet Library

ENet is a UDP (ergo, datagram) network library designed to facilitate quick networking software development. The library is described as thin, simple, and robust, providing event-based communication between peers (which can adopt roles of server and/or client). In addition, it has the option to mimic the properties of stream sockets, which enables experimentation with TCP-esque behaviours.

Built into the framework you have been provided is example code for a server and a client employing ENet. Much of this is self-explanatory, but this hand-out explains the basic premises on which it is built.

Set-Up

To use ENet in a new project, you should include the header file, and ensure you have the requisite .lib (this is provided for you in the framework download). You should also ensure you have your Visual Studio project set to include the relevant directories.

```
1 #include <enet/enet.h>
```

Before using any ENet functions, your program should call the `enet_initialize()` function. This function returns zero if no errors occur. When you exit your program, or are no longer going to employ the ENet library, you should call the `enet_deinitialize()` function, which frees up any resources the library has reserved.

Creating Servers and Clients

ENet refers to a server as a host with a specified address (to receive data and connections), and generates it through the function `enet_host_create()`. Within our framework, this function is found

inside the `NetworkBase::Initialize()` function, which populates the properties of the server. The code excerpt below illustrates the inputs for `enet_host_create()`.

```
bool NetworkBase::Initialize(uint16_t external_port_number,
    size_t max_peers)
{
    ENetAddress address;
    address.host = ENET_HOST_ANY;
    address.port = external_port_number;

    m_pNetwork = enet_host_create(
        (external_port_number == 0) ? NULL : &address,  //the address
        // at which other peers may connect to this host. If NULL,
        // then no peers may connect to the host.
        max_peers,  // the maximum number of peers that should be
        // allocated for the host.
        1, // the maximum number of channels allowed; if 0, then
        // this is equivalent to ENET_PROTOCOL_MAXIMUM_CHANNEL_COUNT
        0, // downstream bandwidth of the host in bytes/second; if 0,
        // ENet will assume unlimited bandwidth.
        0);   //upstream bandwidth of the host in bytes/second; if 0,
        // ENet will assume unlimited bandwidth.

    if (m_pNetwork == NULL)
    {
        NCLERROR("Unable to initialise Network Host!");
        return false; // checks to ensure the server was initialized
    }

    return true;
}
```

NetworkBase::Initialise() function

Triggering this function will generate a server with the parameters provided to the create function. Using these parameters, you can dictate how many peers (clients, in context) can connect to your host - if an instance of the game hosted on your server can only hold 16 players, for example, you might set `max_peers` to 16. On the other hand, if you have a lobby system, where players can jump into an empty slot if another player disconnects, you might use a higher value and allocate players on the 'wait list' to a lobby.

A client in ENet is simply a host that lacks a specified host address (e.g., `address=NULL`). It is created in much the same way as a server, using the `enet_host_create()` function. The variables dictating bandwidth (upstream and downstream) refer, instead, to the client's bandwidth limits (e.g., 57600 to emulate a 56k modem).

Connecting to and Disconnecting from a Host

A client initialises a connection to a foreign host using the function `enet_host_connect()`. In our framework, this is called within the function `NetworkBase::ConnectPeer()`, which accepts integers which define the IP address you wish to connect to, along with the port number. You will notice that `enet_host_connect()` itself accepts four parameters. They are, in order, the client, the address, and the channels you wish to allocate.

```
ENetPeer* NetworkBase::ConnectPeer(uint8_t ip_part1, uint8_t ip_part2,
    uint8_t ip_part3, uint8_t ip_part4, uint16_t port_number)
{
    if (m_pNetwork != NULL)
    {
        ENetAddress address;
```

```
7         address.port = port_number;
8
9         //Host IP4 address must be condensed into a 32 bit integer
10        address.host = (ip_part4 << 24) | (ip_part3 << 16)
11           | (ip_part2 << 8) | (ip_part1);
12
13        ENetPeer* peer = enet_host_connect(m_pNetwork, &address, 2, 0);
14        if (peer == NULL)
15        {
16           NCLERROR("Unable to connect to peer: %d.%d.%d.%d:%d",
17              ip_part1, ip_part2, ip_part3, ip_part4, port_number);
18        }
19
20        return peer;
21     }
22     else
23     {
24        NCLERROR("Unable to connect to peer: Network not
25           initialized!");
26        return NULL;
27     }
28  }
```

NetworkBase::ConnectPeer() function

Clients can be gently disconnected from a host using the `enet_peer_disconnect()` function. This function sends a disconnect request to a foreign host, and the peer will then wait for an acknowledgement from the server before disconnecting; this will generate an event of `ENET_EVENT_TYPE_DISCONNECT` type - see below.

Another option to disconnect from a server, without waiting for acknowledgement, is provided by the `enet_peer_reset()` function. In this case, the server receives no notification that the peer has disconnected, and the connection will eventually time out.

The third option, employed by our framework, is the `enet_peer_disconnect_now()` function. In this case, a disconnect notification is sent to the host, but the client does not wait for confirmation before disconnecting. Should the server not receive the notification, the connection will eventually time out (as with `enet_peer_reset()`).

Event Management

ENet employs polled event management to inform the program when something has occurred that should be reacted to in some fashion. These events are polled using the function `enet_host_service()`, where an optional ping can be stipulated (in ms) to dictate how long ENet should look for an event. If zero is used, `enet_host_service()` will return immediately if there are no events queued for dispatch.

Most network stack processing is handled through this function. As such, both ends of the connection (server and client) need to regularly call the function in order to make sure data is being sent, received, and responded to. One solution to this issue is to poll at the beginning of every loop of your game engine, but a more circumspect approach can be employed so long as it is handled consistently.

A switch statement is normally used to delineate responses to different types of event. Within our framework, `enet_host_service()` is called inside the function `NetworkBase::ServiceNetwork()`.

```
1  ENetPeer* NetworkBase::ConnectPeer(uint8_t ip_part1, uint8_t ip_part2,
2     uint8_t ip_part3, uint8_t ip_part4, uint16_t port_number)
3  {
4     if (m_pNetwork != NULL)
5     {
6        ENetAddress address;
```

```
        address.port = port_number;

        //Host IP4 address must be condensed into a 32 bit integer
        address.host = (ip_part4 << 24) | (ip_part3 << 16)
           | (ip_part2 << 8) | (ip_part1);

        ENetPeer* peer = enet_host_connect(m_pNetwork, &address, 2, 0);
        if (peer == NULL)
        {
           NCLERROR("Unable to connect to peer: %d.%d.%d.%d:%d",
              ip_part1, ip_part2, ip_part3, ip_part4, port_number);
        }

        return peer;
    }
    else
    {
        NCLERROR("Unable to connect to peer: Network not initialized!");
        return NULL;
    }
}
```

NetworkBase::ServiceNetwork() function

If an event is enqueued, the sample code in the `Net1_Client::ProcessNetworkEvent()` function responds to it. Here you can find the switch statement addressing each case that can be returned as an event by the ENet API. We discuss each case below:

```
void Net1_Client::ProcessNetworkEvent(const ENetEvent& evnt)
{
    switch (evnt.type)
    {
        // New connection request or an existing peer accepted
        // our connection request
        case ENET_EVENT_TYPE_CONNECT:
        {
            if (evnt.peer == m_pServerConnection)
            {
                NCLDebug::Log("Network: Successfully
                   connected to server!");

                // Send a 'hello' packet
                char* text_data = "Hellooo!";
                ENetPacket* packet = enet_packet_create(text_data,
                   strlen(text_data) + 1, 0);
                enet_peer_send(m_pServerConnection, 0, packet);
            }
        }
        break;
```

Net1_Client::ProcessNetworkEvent() function: CONNECT

This is the response to a peer connection - either a new connection request, or a peer has accepted the connection request we've sent. In our sample code, a simple message is employed to signify a connection has been made. A more sophisticated response to this event might trigger a series of functions to generate data structures that we expect to be passing back and forth between server and client.

```
        //Server has sent us a new packet
        case ENET_EVENT_TYPE_RECEIVE:
```

```
3         {
4             if (evnt.packet->dataLength == sizeof(Vector3))
5             {
6                 Vector3 pos;
7                 memcpy(&pos, evnt.packet->data, sizeof(Vector3));
8                 m_pObj->Physics()->SetPosition(pos);
9             }
10            else
11            {
12                NCLERROR("Recieved Invalid Network Packet!");
13            }
14
15        }
16        break;
```

Net1_Client::ProcessNetworkEvent() function: RECEIVE

This event type is triggered when we have received a packet from a peer. In the sample code, we validate that the packet is of the size we anticipated (a `Vector3`), and if it is not then an error message is returned. Again, you should be able to see where a more sophisticated series of responses could be triggered to suit your coursework and game development requirements.

```
1         //Server has disconnected
2         case ENET_EVENT_TYPE_DISCONNECT:
3         {
4             NCLDebug::Log("Network: Server has disconnected!");
5         }
6         break;
7     }
8 }
```

Net1_Client::ProcessNetworkEvent() function

This event type is received when the server disconnects. In our sample code, a simple log is made of the event, but you could envision a scenario where this message triggered some other function - such as pausing progress of a game, while an attempt is made to reconnect to the server.

Sending Packets

The entire purpose of our network protocol is the sending and receiving of data of some form between systems. We refer to this data as a *packet*. The ENet library generates packets using the `enet_packet_create()` function. This function requires us to specify the size of the packet (in our framework example, found in `main.cpp` of the Network Server tutorial, this is the size of a `Vector3`).

We can optionally employ flags when we create a packet to assign certain properties to the packet. As an example, the flag `ENET_PACKET_FLAG_RELIABLE` forces the packet to employ reliable delivery. This means that once the packet is received, a confirmation of receipt shall be returned to the sender. Several attempts will be made to deliver the packet if no confirmation of receipt is forthcoming. If a specified number of retry attempts passes, ENet assumes that the peer has disconnected, forcefully resetting the connection.

A packet is sent using the `enet_peer_send()` function, which accepts a destination, a broadcast channel ID, and the packet to be sent. An example of the use of `enet_peer_send()` is found in the code sample outlining the response to a `ENET_EVENT_TYPE_CONNECT` event, above.

A host can use the `enet_host_broadcast()` function to send a packet to every connected peer across a chosen channel. An example of this code can be found in `main.cpp` of the Network Server tutorial.

Lastly, we can optionally resize a packet using the `enet_packet_resize()` function. If, for example, our packet size is dependent upon the number of clients connected to the server (as would be the case in a multi-player FPS updating the positions of every other client with each packet), we could resize our packet on the fly to accommodate connecting and disconnecting clients.

Similarly, the nature of our packets can obviously be more sophisticated than a simple `Vector3`. We can define our own structure of required data, or an array of such structures, facilitating significantly more detailed communication. In optimised network engineering, compressed encoding at the server side and decoding at the client are common practise, if the amount of data being sent uncompressed would cause a greater performance bottleneck than the act of encoding and decoding it.

Implementation

Explore the framework sample code for the `Tuts_Network_Client` and `Tuts_Network_Server` projects. Investigate ways to integrate server-client functionality into your existing physics system, to implement a client-driven entity within your host environment. Consider how you might construct a client to respond in a meaningful manner to data received from the server.

Tutorial Summary

We have introduced the concept of netcode through socket programming, as a means of facilitating multi-player gameplay. We have discussed the low-level socket control provisioned by the Winsock API, before presenting an accessible, higher-level event-based networking library with sample code.