Sending and Receiving Packets

How to send and receive packets over UDP with BSD sockets

Posted by Glenn Fiedler (https://gafferongames.com/about) on Friday, October 3, 2008

Introduction

Hi, I'm Glenn Fiedler (https://gafferongames.com/about) and welcome to **Networking for Game Programmers** (https://gafferongames.com/categories/game-networking/).

In the previous article (https://gafferongames.com/post/udp_vs_tcp/) we discussed options for sending data between computers and decided to use UDP instead of TCP for time critical data.

In this article I am going to show you how to send and receive UDP packets.

BSD sockets

For most modern platforms you have some sort of basic socket layer available based on BSD sockets.

BSD sockets are manipulated using simple functions like “socket”, “bind”, “sendto” and “recvfrom”. You can of course work directly with these functions if you wish, but it becomes difficult to keep your code platform independent because each platform is slightly different.

So although I will first show you BSD socket example code to demonstrate basic socket usage, we won’t be using BSD sockets directly for long. Once we’ve covered all basic socket functionality we’ll abstract everything away into a set of classes, making it easy to you to write platform independent socket code.

Platform specifics

First let’s setup a define so we can detect what our current platform is and handle the slight differences in sockets from one platform to another:

```
// platform detection

#define PLATFORM_WINDOWS  1
#define PLATFORM_MAC      2
#define PLATFORM_UNIX     3

#if defined(_WIN32)
#define PLATFORM PLATFORM_WINDOWS
#elif defined(__APPLE__)
#define PLATFORM PLATFORM_MAC
#else
#define PLATFORM PLATFORM_UNIX
#endif
```

Now let’s include the appropriate headers for sockets. Since the header files are platform specific, we’ll use the platform #define to include different sets of files depending on the platform:

```
    #if PLATFORM == PLATFORM_WINDOWS

        #include <winsock2.h>

    #elif PLATFORM == PLATFORM_MAC ||
          PLATFORM == PLATFORM_UNIX

        #include <sys/socket.h>
        #include <netinet/in.h>
        #include <fcntl.h>

    #endif
```

Sockets are built in to the standard system libraries on unix-based platforms so we don't have to link to any additonal libraries. However, on Windows we need to link to the winsock library to get socket functionality.

Here is a simple trick to do this without having to change your project or makefile:

```
    #if PLATFORM == PLATFORM_WINDOWS
    #pragma comment( lib, "wsock32.lib" )
    #endif
```

I like this trick because I'm super lazy. You can always link from your project or makefile if you wish.

Initializing the socket layer

Most unix-like platforms (including macosx) don't require any specific steps to initialize the sockets layer, however Windows requires that you jump through some hoops to get your socket code working.

You must call "WSAStartup" to initialize the sockets layer before you call any socket functions, and "WSACleanup" to shutdown when you are done.

Let's add two new functions:

```
bool InitializeSockets()
{
    #if PLATFORM == PLATFORM_WINDOWS
    WSADATA WsaData;
    return WSAStartup( MAKEWORD(2,2),
                       &WsaData )
        == NO_ERROR;
    #else
    return true;
    #endif
}

void ShutdownSockets()
{
    #if PLATFORM == PLATFORM_WINDOWS
    WSACleanup();
    #endif
}
```

Now we have a platform independent way to initialize the socket layer.

Creating a socket

It’s time to create a UDP socket, here’s how to do it:

```
int handle = socket( AF_INET,
                     SOCK_DGRAM,
                     IPPROTO_UDP );

if ( handle <= 0 )
{
    printf( "failed to create socket\n" );
    return false;
}
```

Next we bind the UDP socket to a port number (eg. 30000). Each socket must be bound to a unique port, because when a packet arrives the port number determines which socket to deliver to. Don’t use ports lower than 1024 because they are reserved for the system. Also try to avoid using ports above 50000 because they used when dynamically assigning ports.

Special case: if you don’t care what port your socket gets bound to just pass in “0” as your port, and the system will select a free port for you.

```
    sockaddr_in address;
    address.sin_family = AF_INET;
    address.sin_addr.s_addr = INADDR_ANY;
    address.sin_port = 
        htons( (unsigned short) port );

    if ( bind( handle, 
               (const sockaddr*) &address, 
               sizeof(sockaddr_in) ) < 0 )
    {
        printf( "failed to bind socket\n" );
        return false;
    }
```

Now the socket is ready to send and receive packets.

But what is this mysterious call to “htons” in the code above? This is just a helper function that converts a 16 bit integer value from host byte order (little or big-endian) to network byte order (big-endian). This is required whenever you directly set integer members in socket structures.

You’ll see “htons” (host to network short) and its 32 bit integer sized cousin “htonl” (host to network long) used several times throughout this article, so keep an eye out, and you’ll know what is going on.

Setting the socket as non-blocking

By default sockets are set in what is called “blocking mode”.

This means that if you try to read a packet using “recvfrom”, the function will not return until a packet is available to read. This is not at all suitable for our purposes. Video games are realtime programs that simulate at 30 or 60 frames per second, they can’t just sit there waiting for a packet to arrive!

The solution is to flip your sockets into “non-blocking mode” after you create them. Once this is done, the “recvfrom” function returns immediately when no packets are available to read, with a return value indicating that you should try to read packets again later.

Here's how put a socket in non-blocking mode:

```
#if PLATFORM == PLATFORM_MAC ||
    PLATFORM == PLATFORM_UNIX

    int nonBlocking = 1;
    if ( fcntl( handle,
                F_SETFL,
                O_NONBLOCK,
                nonBlocking ) == -1 )
    {
        printf( "failed to set non-blocking\n" );
        return false;
    }

#elif PLATFORM == PLATFORM_WINDOWS

    DWORD nonBlocking = 1;
    if ( ioctlsocket( handle,
                      FIONBIO,
                      &nonBlocking ) != 0 )
    {
        printf( "failed to set non-blocking\n" );
        return false;
    }

#endif
```

Windows does not provide the "fcntl" function, so we use the "ioctlsocket" function instead.

Sending packets

UDP is a connectionless protocol, so each time you send a packet you must specify the destination address. This means you can use one UDP socket to send packets to any number of different IP addresses, there's no single computer at the other end of your UDP socket that you are connected to.

Here's how to send a packet to a specific address:

```
    int sent_bytes =
        sendto( handle,
                (const char*)packet_data,
                packet_size,
                0,
                (sockaddr*)&address,
                sizeof(sockaddr_in) );

    if ( sent_bytes != packet_size )
    {
        printf( "failed to send packet\n" );
        return false;
    }
```

Important! The return value from "sendto" only indicates if the packet was successfully sent from the local computer. It does *not* tell you whether or not the packet was received by the destination computer. UDP has no way of knowing whether or not the the packet arrived at its destination!

In the code above we pass a "sockaddr_in" structure as the destination address. How do we setup one of these structures?

Let's say we want to send to the address 207.45.186.98:30000

Starting with our address in this form:

```
    unsigned int a = 207;
    unsigned int b = 45;
    unsigned int c = 186;
    unsigned int d = 98;
    unsigned short port = 30000;
```

We have a bit of work to do to get it in the form required by "sendto":

```
    unsigned int address = ( a << 24 ) |
                           ( b << 16 ) |
                           ( c << 8  ) |
                             d;

    sockaddr_in addr;
    addr.sin_family = AF_INET;
    addr.sin_addr.s_addr = htonl( address );
    addr.sin_port = htons( port );
```

As you can see, we first combine the a,b,c,d values in range [0,255] into a single unsigned integer, with each byte of the integer now corresponding to the input values. We then initialize a “sockaddr_in” structure with the integer address and port, making sure to convert our integer address and port values from host byte order to network byte order using “htonl” and “htons”.

Special case: if you want to send a packet to yourself, there’s no need to query the IP address of your own machine, just pass in the loopback address 127.0.0.1 and the packet will be sent to your local machine.

Receiving packets

Once you have a UDP socket bound to a port, any UDP packets sent to your sockets IP address and port are placed in a queue. To receive packets just loop and call “recvfrom” until it fails with EWOULDBLOCK indicating there are no more packets to receive.

Since UDP is connectionless, packets may arrive from any number of different computers. Each time you receive a packet “recvfrom” gives you the IP address and port of the sender, so you know where the packet came from.

Here’s how to loop and receive all incoming packets:

```
    while ( true )
    {
        unsigned char packet_data[256];

        unsigned int max_packet_size =
            sizeof( packet_data );

        #if PLATFORM == PLATFORM_WINDOWS
        typedef int socklen_t;
        #endif

        sockaddr_in from;
        socklen_t fromLength = sizeof( from );

        int bytes = recvfrom( socket,
                              (char*)packet_data,
                              max_packet_size,
                              0,
                              (sockaddr*)&from,
                              &fromLength );

        if ( bytes <= 0 )
            break;

        unsigned int from_address =
            ntohl( from.sin_addr.s_addr );

        unsigned int from_port =
            ntohs( from.sin_port );

        // process received packet
    }
```

Any packets in the queue larger than your receive buffer will be silently discarded. So if you have a 256 byte buffer to receive packets like the code above, and somebody sends you a 300 byte packet, the 300 byte packet will be dropped. You *will not* receive just the first 256 bytes of the 300 byte packet.

Since you are writing your own game network protocol, this is no problem at all in practice, just make sure your receive buffer is big enough to receive the largest packet your code could possibly send.

Destroying a socket

On most unix-like platforms, sockets are file handles so you use the standard file “close” function to clean up sockets once you are finished with them. However, Windows likes to be a little bit different, so we have to use “closesocket” instead:

```
#if PLATFORM == PLATFORM_MAC ||
    PLATFORM == PLATFORM_UNIX
close( socket );
#elif PLATFORM == PLATFORM_WINDOWS
closesocket( socket );
#endif
```

Hooray windows.

Socket class

So we’ve covered all the basic operations: creating a socket, binding it to a port, setting it to non-blocking, sending and receiving packets, and destroying the socket.

But you’ll notice most of these operations are slightly platform dependent, and it’s pretty annoying to have to remember to #ifdef and do platform specifics each time you want to perform socket operations.

We’re going to solve this by wrapping all our socket functionality up into a “Socket” class. While we’re at it, we’ll add an “Address” class to make it easier to specify internet addresses. This avoids having to manually encode or decode a “sockaddr_in” structure each time we send or receive packets.

So let’s add a socket class:

```
class Socket
{
public:

    Socket();

    ~Socket();

    bool Open( unsigned short port );

    void Close();

    bool IsOpen() const;

    bool Send( const Address & destination,
               const void * data,
               int size );

    int Receive( Address & sender,
                 void * data,
                 int size );

private:

    int handle;
};
```

and an address class:

```
class Address
{
public:

    Address();

    Address( unsigned char a,
             unsigned char b,
             unsigned char c,
             unsigned char d,
             unsigned short port );

    Address( unsigned int address,
             unsigned short port );

    unsigned int GetAddress() const;

    unsigned char GetA() const;
    unsigned char GetB() const;
    unsigned char GetC() const;
    unsigned char GetD() const;

    unsigned short GetPort() const;

private:

    unsigned int address;
    unsigned short port;
};
```

Here’s how to to send and receive packets with these classes:

```
    // create socket

    const int port = 30000;

    Socket socket;

    if ( !socket.Open( port ) )
    {
        printf( "failed to create socket!\n" );
        return false;
    }

    // send a packet

    const char data[] = "hello world!";

    socket.Send( Address(127,0,0,1,port), data, sizeof( data ) );

    // receive packets

    while ( true )
    {
        Address sender;
        unsigned char buffer[256];
        int bytes_read =
            socket.Receive( sender,
                            buffer,
                            sizeof( buffer ) );
        if ( !bytes_read )
            break;

        // process packet
    }
```

As you can see it’s much simpler than using BSD sockets directly.

As an added bonus the code is the same on all platforms because everything platform specific is handled inside the socket and address classes.

Conclusion

You now have a platform independent way to send and receive packets. *Enjoy* :)

NEXT ARTICLE: Virtual Connection over UDP (https://gafferongames.com/post/virtual_connection_over_udp/)

● (https://www.linkedin.com/in/glennfiedler/)

● (https://twitter.com/gafferongames)

● (https://github.com/gafferongames)