Introduction to Multiplayer Game Programming

With the surge in popularity of multiplayer games such as Halo, Worlds of Warcraft, and Counter-Strike, it is becoming increasingly important for game developers to support multiple players. Unfortunately information on how to write multiplayer on-line games is lacking -- you can find the occasional article that discusses one aspect, but no single resource exists that touches on all the relevant areas and how they relate to each other.

One of the aspects of multiplayer game programming that makes it so confusing is the sheer number of topics and how they relate to each other. Code must be written that handles everything from high level game specific tasks all the way down to extremely low level tasks such as network packet broadcast and reception.

High Level Architecture

efore delving into the nitty gritty low level aspects of implementing a multiplayer game, we need to take a step back and think about what we're trying to do from a higher level. Specifically, "sending the bits" is a different problem than "which bits to send", so we'll tackle the latter part first.

At some point with any multiplayer game one of the computers has to make the final decision as to the outcome of some action. For example, if two players are competing head to head, then you can't have a situation where Player A thinks he killed Player B while Player B thinks he killed Player A.

The two most common architectures for resolution arbitration are client-server and peer-to-peer. Client-server is both conceptually simpler and easier to implement than peer-to-peer, so we'll cover that first.

Client/Server

In a client-server architecture all the players, or "clients", are connected to a central machine, the server. The server is responsible for all important decisions, managing state and broadcasting this information to the individual clients. A single view of the world is maintained by the server, which obviously helps with keeping things consistent.

As a result, the server becomes a key bottleneck for both bandwidth and computations. Instead of distributing the load among all the players, the server must do all the work. And, of course, it has to send and receive N independent streams of data, so its network connection is similarly taxed.

Sometimes the server will be running on a player's machine as a "local server" or a "listen server". The rules still apply in this case, because the client and server are logically decoupled even if running on the same physical system.

Peer-to-Peer

A peer-to-peer system spreads the computational load out among all the players. If you have 8 players, each with a computer, then it's nice to leverage all the available computing power. The downside, of course, is that "computation" means "decision making", so cheating can become rampant (each client can be hacked to report results favorable to that specific player). In addition, the system is more susceptible to consistency errors since each peer has to make sure that it broadcasts its "decisions" and it must base this on the data provided by the other peers. If a peer falls off the network or doesn't get correct information in a timely manner, synchronization failures can and will occur since it's analogous to a CPU failing in a multiprocessor computer.

The advantage of a peer-to-peer server is that overall bandwidth and computational requirements for each system are reduced, and you don't need a single beefy server responsible for managing the entire game.

Hybrid

In reality, most architectures are hybrid systems as going to an extreme in either direction can lead to significant problems.

For example, in a true client-server system, the client would never move the player until the server responded with a "based on your last input, here is your new position". This is fine, assuming you have client-side prediction (discussed later), but this means that the server is handling all collision detection. This

excessively computationally expensive, to the point that it's not tenable for large worlds.

A compromise would be to allow the clients to manage their own movement, and they in turn report their location to the server (which likely does some basic sanity checking on the reported movement). This leverages the computing power of each client and off loads a tremendous amount of work from the server. But since the client is now authoritative about something (its position), the opportunity for cheating is amplified.

Example: Pure Client/Server

Example: Pure Client-Server

Pure client-server implementations abound, however they are not generally considered particularly advanced or "sexy". The best way to think of a pure client-server model is that each client is a dumb terminal -- the only thing it does is transmit player input to the server and report server messages to the player.

The standard text MUD (multi-user dungeon) is a classic example of a pure client-server architecture. In fact, you can play many (if not most) text MUDs using the dumbest terminal possible -- the telnet program.

The server's main loop would look like this:

```
while not done
   for each player in world
      if input exists
         get player command
         execute player command
         tell player of the results
   simulate the world
   broadcast to all players
```

The client's main loop would be extremely simple:

```
while not done
   if player has typed any text
      send typed text to server
   if output from server exists
      print output
```

Note that the client does not perform any simulation -- it solely handles input and output, and in fact doesn't even necessarily understand anything about the game at all.

Example: Pure Peer-to-Peer

Now let's look at a hypothetical pure peer-to-peer architecture. Imagine a tank game where two players can compete head to head. If such a game was peer-to-peer, you could say that each player is authoritative about his tank's position and state, and is also responsible for determining if they've hit the other tank.

Since it's peer-to-peer, there is no server main loop, only the client main loop, which might be something like this:

```
while not done
   collect player input
   collect network input about other players
   simulate player
      update player's state if informed it's been hit
      inform other clients if they've been hit
      move player
      update other player state (ammo, armor, etc.)
   report player state to other clients
```

If you've ever dealt with on-line cheaters, alarm bells should be going off looking at the above. Specifically, since the client is responsible for simulation, any of those simulation related items can be cheated.

The first is updating the player's state if he's been hit -- it would be relatively easy for a hacker to simply ignore incoming "damage packets" (this can be done by patching the client program or even by hooking up a packet filter to monitor incoming network traffic).

The second is informing other clients they've been hit. It would, once again, be relatively trivial to tell each client they've been hit every frame. And since the other clients are explicitly trusting your client, they'll do the dutiful thing and take the damage.

The third is a simple one -- movement. Since the client is responsible for the player's state, it can be modified to make the player move at arbitrary speed or even teleport to random locations ("hyperspace").

Finally, even mundane things can be hacked -- infinite ammo, infinite armor, infinite powerups, or even something as basic as the score.

The key issue here is that since each client determines a lot of what happens in the world, there are many opportunities for cheating. A client-server architecture can still suffer from cheating, but it would require either active participation on the part of the server operator or fairly lightweight cheating that operates by simulating a perfect player (aiming bots) or providing the player nominally inaccessible information (heads up displays, radar).

As a general rule, peer-to-peer architectures are significantly more difficult to secure against cheaters and hacks.

Of course, not having a server does simplify some things, and no single machine is required to simulate the entire world. Network bandwidth for each individual machine gets worse since they have to communicate with all the other machines directly, but you don't have the concentrated network bandwidth that would be seen at a single server.

Example: Hybrid Model

A hybrid system combines aspects of client-server and peer-to-peer architectures. The idea is that by off-loading some of the work on the clients, each client in turn will enjoy better responsiveness (by not having to wait for the server updates) and the server will also have its workload reduced.

For example, a massively multiplayer game may have the server be authoritative about combat and player statistics, but the client may end up authoritative for movement. Since movement can consume a lot of CPU cycles this will lessen the workload on the server, and at the same time allow smooth movement on the part of the client. Of course, this is open to hacking, such as the various infamous "speed hacks".

Client Side Prediction

If the server (or another peer) are authoritative about the world, there can be a significant amount of delay incurred as they broadcast new state information back to the client. This lag can be jarring, and in some cases completely impractical.

For example, in the case of a first person shooter, it's simply not feasible to have the client transmit "intended view changes" (i.e. moving the mouse) and then wait for a response from the server in order to present those changes locally.

So what you'll normally do is assume that the various movement or orientation changes have been propagated and returned from the server. Then, if the server decides that it's not allowed, you back up and correct.

This is especially important for movement, where you're running around a world and want it to feel smooth. The client can predict movement fairly accurately, and only in exceptional cases will you be corrected. Also, you can ease in corrections gradually instead of making it pop suddenly, reducing the warping and popping artifacts.

Low Level Architecture

Now that we've sorted out the different types of high level architectures, we need to figure out how to do our low level communication architecture. The high level architecture determined how the different computers and elements of the game communicated with each other, and more important, which components were responsible for arbitration and decision making.

The low level network layer, however, doesn't care about any of that. All it worries about is packaging up data and sending it across the wire (and, conversely, receiving data from other machines).

Back in the good old days, there were many different and proprietary network transports. Novell had IPX/SPX; Microsoft had NetBEUI; Apple had AppleTalk; and many other companies had their own proprietary systems. By the early 1990s, however, the TCP/IP (Transport Control Protocol/Internet Protocol) protocol eventually won out as the low level networking layer of choice.

There are many good books and Web pages that describe TCP/IP, the OSI network stack, and other details that aren't directly relevant to this discussion. For us, it suffices that TCP/IP is our network layer of choice. If for some bizarre reason you get stuck in a time warp and have to support LAN play circa 1995, then you'll have to investigate Novell's IPX/SPX protocol support under DOS. But if you're not stuck in said time warp, TCP/IP is the only protocol you'll need to worry about for the forseeable future.

TCP vs. UDP

TCP/IP is actually an umbrella term to cover TCP and IP and a host of protocols (TCP, UDP, and ICMP) that are layered on IP. The relevant part for us are the TCP and UDP protocols, and by association the IP protocol for relaying packets.

The Transmission Control Protocol, or TCP, is a high-level (to the application), connection oriented, reliable, in-order packet delivery system. This means that it logically thinks of one machine "connecting" to another, and that any traffic sent between the two is both guaranteed to arrive and, just as importantly, guaranteed to arrive in order. In addition, TCP is a stream protocol -- distinct packets aren't set, instead a constant stream of data is pumped from the source to the destination (underneath it's still packets, but to the application it looks like a stream of bytes).

The User Datagram Protocol, or UDP, is a more primitive set of features than TCP. It is a connectionless, unreliable datagram protocol. Computers send each other packets of information that may or may not arrive at the destination, and which may or may not arrive in the same order they were sent. There is also a practical limit on the size of the out-going packets.

Structurally speaking, TCP and UDP are sibling protocols since they both sit right on top of the IP layer, but to the programmer UDP is a much lower level protocol since it provides a smaller set of features compared to TCP.

For this reason, TCP is more convenient than UDP, but that convenience comes at a cost. TCP can exhibit significantly lower overall performance than UDP because of the extensive error checking, handshaking, congestion control and acknowledgements that occur as part of a data transfer. With TCP, even if you have received a packet, the kernel/stack will withhold that packet until all previous packets have arrived -- this means that while more up to date information is available, your code may never see it until the TCP stacks decides you can. This may include a 3-second retransmit timer in the case of lost packets.

As tempting as using TCP sounds, just don't do it. It always, and I mean always, leads to heartbreak in the end for real-time games. Numerous networked games have tried to do the expedient thing on TCP and end up switching to UDP -- often painfully -- later.

For this reason, UDP will be our network transport of choice. We'll have to implement a lot of functionality on top of it that TCP provides, but by doing so we can tailor the performance and functionality exactly the way we want.

BSD Sockets and WinSock

Fine, we've determined that UDP is the protocol we want to use, but how do we access it? Most modern operating systems export system level features via APIs (Application Programming Interfaces). Examples of common APIs include OpenGL; DirectX, GDI and Win32 (on Microsoft Windows); and Carbon on the Macintosh.

For our purposes, we're going to be using the sockets network API, also known as "BSD sockets". The BSD sockets API was developed in the early 80s to provide a reasonably portable method of accessing the TCP/IP subsystem (or "stack") on the various Unix flavors available at the time. A variant, WinSock, was eventually developed for Windows.

Functionally, WinSock and BSD sockets are extremely similar. In fact, code can often be shared between

them with a few appropriate definitions and `#ifdefs` around some of the Windows code. There are several more, often subtle, differences between WinSock and BSD sockets, but by and large they're not significant issues for most developers.

UDP Concepts

As mentioned earlier, UDP is a connectionless, unreliable datagram protocol. Let's break that down to see what we're dealing with.

Connectionless

UDP does not have a native concept of a "connection". One computer does not connect to another -- communication is not established then persisted for some duration of time. Instead, communication is performed one packet, or datagram, at a time. It's like mailing letters to someone, you put an address on each envelope and send it off.

Unreliable

Unlike TCP, UDP does not guarantee that a packet sent from one computer to another will actually arrive. In addition, UDP does not guarantee that any packets received will arrive in the same order they were sent. As you can imagine, this is a somewhat inconvenient set of assumptions to deal with, so most games will either have to implement a reliable, in-order transfer mechanism on top of UDP and/or they will have to architect their high level protocols such that reliable, in-order data is unnecessary. I'll talk about the latter a little bit later.

Datagram Protocol

UDP operates by sending discrete sized packets of data, or datagrams. This is different from a stream-oriented protocol such as TCP which sends data a byte at a time, leaving it up to the receiver to parse the byte stream. The proper size for a packet varies depending on various factors, but a good, round number that many people throw out is 1400 bytes since that's a bit smaller than a typical Ethernet MTU (Maximum Transmission Unit).

Addresses and Ports

UDP packets are sent to a destination address consisting of an IP address and a port number (16-bit unsigned value). The destination system must be listening on that port (discussed later) in order to receive the packet.

Choosing a Port

There are three ranges of port values that any server author needs to be aware of: System (Well-Known) Ports (0 through 1023); User/Ephemeral (Registered) Ports (1024 through 49151); and Dynamic/Private Ports (49152 through 65535).

An application should never use port numbers assigned to the System Port range. Dynamic/Private Ports are pretty much fair game, but you cannot rely on the availability of a particular port. So you'll probably end up doing what most people do -- choose a port in the User Port range and just call it your own. On the off-chance that a user is running an application that allocates that same port, you should allow for the port value to be overridden or specified by the user.

Using UDP

As much as I'd like to, I simply don't have the time to do a full UDP library implementation. There is plenty of information on actual socket and WinSock? usage out there, so thankfully you're not at a loss when it comes to the basic stuff of UDP usage. Check the bibliography for references on UDP programming.

However, I highly suggest using the very good ENet open source UDP networking library. It handles most of the nitty gritty UDP level stuff, letting the programmer concentrate on higher level, game specific architectural concerns.

Under the sockets API all communication goes through a socket object allocated using the socket() system call. A socket is used for both sending and receiving data. Once you have a socket allocated, you can begin

sending data to an explicit IP address and port combination using the `sendto()` API:

```
int sendto( SOCKET s,
            const char *buf,
            int len,
            int flags,
            const struct sockaddr *to,
            int tolen );
```

The first parameter is the socket we've previously allocated with the `socket()` call. The second parameter is a pointer to a buffer of data we'd like to send. The third parameter specifies the number of bytes in `buf`. The `flags` parameter specifies any special flags relating to this call. The `to` structure contains the destination address, and `tolen` is the size of the `to` buffer.

This is where it gets a little ugly. Specifically, the whole `sockaddr` situation is a bit of a type casting nightmare, but it's nothing too onerous once you figure it out.

Under UDP, your destination address is defined by a special structure called a `sockaddr_in`. Under WinSock, it looks like this:

```
struct sockaddr_in {
      short   sin_family;
      u_short sin_port;
      struct  in_addr sin_addr;
      char    sin_zero[8];
};
```

We need to fill that in with the appropriate values for `sendto()` to correctly send our chunk of data to the destination. `sin_family` we simply set to `AF_INET`.

`sin_addr` is a little bit trickier. It needs to have a network ordered (big endian) network address. The easiest way to get this is to pass a string representing a standard IP address (e.g. "192.168.1.100") to the socket function `inet_addr()`. This will return a properly formatted value suitable for placement in the `sockaddr_in` structure.

Note that `inet_addr()` doesn't support domain name lookup (i.e. converting "foo.example.com" to an IP address), it only handles "dot" format numeric addresses. If your application needs to support both dot and named IP addresses, the easy way to handle this situation is to use `gethostbyname()` if `inet_addr()` fails:

```
unsigned long
UDP_lookupAddress( const char *kpAddress )
{
   unsigned long a;

   //Try looking up as a raw address first
   if ( ( a = inet_addr( kpAddress ) ) == INADDR_NONE )
   {
      //If it fails (isn't a dotted IP), resolve it
      //through DNS

      hostent* pHE = gethostbyname( kpAddress );

      //Didn't resolve
      if ( pHE == 0 )
      {
          return INADDR_NONE;
      }

      //It did resolve, do some casting ugliness
      a = *((unsigned long*)pHE->h_addr_list[0]);
   }
```

```
   return a;
}
```

Finally, we need to specify the destination port. This too has to be in network ordered format, so we use the helper function `htons()` (host-to-network-short) to convert from host to network format.

Okay, let's put this together in a short snippet of code for filling in the `sockaddr_in` structure:

```
int UDP_fillSockAddrIn( struct sockaddr_in *sin,
                        const char *kpAddr,
                        unsigned short port )
{

   //zero memory
   memset( sin, 0, sizeof( *sin ) );

   //set the family
   sin->sin_family = AF_INET;

   //Set our port -- use "htons" to convert from
   //host endianess to network endian byte order
   sin->sin_port = htons( port );

   //Set our address
   sin->sin_addr.s_addr = UDP_lookupAddress( kpAddr );

   //Make sure the address resolved okay
   if ( sin->sin_addr.s_addr == INADDR_NONE )
      return 0;
   return 1;
}
```

Holy crap, that's a lot of work. So let's tie it all together to send a buffer to some address just so we can see what the big picture is like:

```
void SendSomeData( const char *kpAddress, unsigned short port,
                   const void *kpSrc, int nbytes,
                   SOCKET s )
{
   sockaddr_in sin;

   if ( !UDP_fillSockAddr( &sin, kpAddress, port ) )
      return; //do better error handling, obviously

   sendto( s,
           ( const char * ) kpSrc,
           nbytes,
           0,
           ( const struct sockaddr * ) &sin,
           sizeof( sin ) );
```

Sending Data (Implicit Destination)

The above snippets show how to send data to an explicit destination. H owever you can send data to in implicit destination by binding your socket to a specific address using the `bind()` call. Once you do that, you can now send data using the `send()` API without specifying an explicit destination each time. This may seem like you're connecting to the destination, but don't be fooled, you're not -- you're simply telling your local socket that your destination address will be implicit in the future.

Receiving Data

Data is received on a socket by calling either `recv()` or `recvfrom()` on that socket. The two APIs operate identically, except that `recvfrom()` will tell you the address of the packet's origination. This is handy when you want to respond to that packet and you don't know where it came from.

Blocking vs. Non-Blocking I/O

By default most sockets implementations are "blocking". This means that when asked to do something, they'll wait until they're done doing it. This is bad for a game (unless you're running your network pump in a separate thread, but we don't have the space to get into that and, as a general rule, I would not recommend it) because you may call a sockets function and then hang for an indeterminate period of time.

Windows and BSD sockets have slightly different ways of enabling or disabling non-blocking IO.

Under WinSock you call `ioctlsocket()`:

```
//enable non-blocking IO
arg = 1;
ioctlsocket( s, FIONBIO, &arg );
```

However under BSD sockets you go through a slightly more circuitous route:

```
fcntl( s, F_SETFL, O_NONBLOCK | fcntl( s, F_GETFL ) );
```

Some Unix-like operating systems don't have the `fcntl()` interface, and on those you have to use `ioctl()`, which operates similarly to WinSock's interface:

```
arg = 1;
ioctl( s, FIONBIO, &arg );
```

Once a socket is no longer blocking, it will return a "soft" error in `recv[from]()` if there is no data waiting. It's important to check for this condition instead of just trapping it as an error!

```
int err, result;

char buf[ 2048 ];

while ( 1 )
{
   result = recv( s, buf, sizeof(buf), 0 );

   if ( result == SOCKET_ERROR )
   {
      err = errno; /* under Windows use WSAGetLastError() */

      if ( err == EAGAIN ) /* WSAWOULDBLOCK on Windows */
      {
         /* no data waiting, not really an error */
         sleep( 10 ); //or break, or whatever
      }
      else
      {
         /* a real error occurred! */
      }
   }
   else
   {
      /* do something with data returned from recv() */
      /* should have 'result' number of bytes in buf */
   }
}
```

How Many Ports?

Since an application can allocate many ports, a common question is "How many ports should I use?" There aren't any really strong arguments one way or another, but it basically breaks down like this:

- using multiple ports can simplify demultiplexing incoming packets. For example, you might assign one port per client, and thus assume all incoming traffic on that port is from that one client. Or you might have a port dedicated to "control" data and another port dedicated to "state" data, etc. This simplifies things moderately, but at the expense of creating and maintaining more ports. And you'll still have to verify the validity of each packet, so if you're going to be doing packet inspection anyway, manual demultiplexing won't cost much more.
- each port will usually have its own set of buffers within the operating system. For this reason, if you find that you are accumulating a lot of network traffic, it might make sense to use more ports simply to have more buffer space.
- the more ports you need, the higher the likelihood you will conflict with an existing port used by another service. You will also use up more system resources, and will also have to open/forward more ports on your firewall or NAT box.

The simplest choice is to use a single port and demultiplex packets based on their source address and/or payload.

UDP + TCP

One commonly floated suggestion is "Why not use UDP for unreliable data and TCP for reliable data?" At first glance this seems like a genuinely good idea, however in practice this falls apart. Because of the performance characteristics exhibited by TCP on flaky network connections, it will tend to run "skewed" from the UDP traffic. This means that the two different streams may end up out of phase with each other, often times causing very odd behaviour in game.

Just use a reliable system over UDP like ENet and ditch TCP altogether. Really.

Network Address Translation

Because of the limited allocation of unique IP addresses to businesses and individuals, there has been a surge of popularity of "network address translation" routers, also known as "NAT boxes".

The theory behind a NAT box is fairly simple -- it takes a single, public IP address and multiplexes it to multiple internal computers with private IP addresses. It accomplishes this by keeping track of incoming and outgoing traffic and doing the appropriate address translations on the fly. It's conceptually simple, however it does have a set of drawbacks when dealing with gaming.

I'm not going to get into much detail on how NAT works -- there are a ton of good articles on this available on the net -- but I will discuss briefly how it affects a multiplayer game.

The first, and most obvious, problem is that a game hosted behind a NAT box needs to have its "listening" port forwarded. This is pretty standard stuff, nothing magical there -- you just have to make sure that servers behind a NAT or firewall know to configure their network to forward traffic on that port to the server. In a peer-to-peer system, each player's machine must also have a port opened to accept traffic.

The second problem is that a game cannot assume that all traffic coming from a single IP address is from the same computer (since one public IP may be shared by multiple computers). Instead, it must examine the incoming packet's source address -- including port -- to determine which client is sending that data.

The third problem -- and a fairly rare one at that, but which is insidious for this very reason -- is that some routers/NAT boxes actually change the port they're forwarding dynamically. Since UDP is a connectionless protocol, technically there is no reason for them not to do this, but the practice is rare enough that many developers rely on the assumption that a player's initial IP:port source address will remain constant during the life of a game. Unfortunately, this isn't so.

This can cause problems is if a game server binds client information to a source address.

For example, Biff tries to connect to a game's server. His source address (via NAT) may be 76.54.32.10:4567. Your game says "Okay, this is fine, we know that Biff is always at 76.54.32.10:4567".

The next time you receive a packet from that address, you know it's from Biff because you've logically bound that address to Biff.

Now Charlie connects to your game, and he's playing behind the same NAT box as Biff. His address is 76.54.32.10:6789 (reminder: he has the same IP address since they share an IP via the NAT box). Once again, no problem, we keep cruising like we did before since their two ports are different.

But then the NAT box, for some weird reason, decides it's going to translate Biff's address a little different. All of a sudden it decides that Biff's source address is going to be 76.54.32.10:9876. In a connectionless protocol, this is fine, since in theory a UDP-based server doesn't understand the notion of a "connected" client -- all incoming UDP packets are supposed to be responded via the source address provided by `recvfrom()`. Unfortunately, that's not how a game works.

So your game server gets a weird request from 76.54.32.10:9876, a source address that doesn't map to any known clients -- an error condition. Not only that, but suddenly Biff doesn't seem to be connected anymore.

Detecting this condition is a royal pain in the ass, and can lead to nightmarish support problems where a very small fraction players complain about getting intermittently disconnected for no apparent reason.

One way to solve this is to avoid persistently processing packet data based on the source address port value. Instead, each client should send a unique client ID that can be used to differentiate between multiple clients on a single IP address. I use a 16-bit unique identifier based on the the lower two bytes of the client's private IP address. For example, 192.168.1.10 would generate a unique client ID of `( 1 << 8 ) | 10`, or `0x010A`.

Note: You could theoretically use the lowest byte only and still work in most situations (homes and small businesses), but larger networks with larger (two-byte) private IP ranges would pose a potential source of conflict. For example, if Biff was on 192.168.1.66 and Charlie was on 192.168.2.66 then a system that only looked at the low order bytes would be unable to differentiate between the two clients.

Encryption Note: If the incoming packets are session key encrypted, you obviously can't encrypt the client ID since you won't be able to decrypt it. In these cases, you'll probably have a packet structure where the first 16-bits are clear text with the client ID, which can then be used to look up the client's session key which is in turn used to decrypt the remainder of the payload.

Reliable In-Order Delivery

Let's say you refuse to use ENet and instead want to do your own reliable, in-order protocol. I'll try to briefly describe a simple way to implement this.

The first thing you'll have to do is tag your outgoing packets with a sequence number. This value increments for each packet that goes out on the wire to a particular destination.

When a packet is sent for reliable delivery, you prepend the header (with sequence number) on the packet, then you store this packet in a buffer somewhere. When the packet arrives at the destination, the receiver needs to send an acknowledgement back to the sender saying "I received packet XYZ" (in reality this is fairly inefficient, and what you'll really want to do is just send back "the most recent in-order packet" you received, typically piggy backed on other data).

Until the sender receives that ack, it will continue to periodically resend that packet. When the ack arrives, the copy of the outgoing packet is deleted from the buffer.

Pretty simple stuff, however it doesn't take into account the issue of in-order delivery. Thankfully the sequence number handles this for you as well. When the receiver accepts a new packet, it simply checks the sequence number against the last sequence number received from that source, and if the sequence number is less than or equal to the last one, it just ignores it.

If the new packet's sequence is equal to the next expected sequence number (last sequence + 1), then it's accepted and the sender is notified that the packet arrived safely.

If the new packet's sequence is larger than the expected sequence number, you can do one of two things. The simplest thing to do is ignore it and wait for a resend from the server, however this is not very efficient. Instead, what you'll want to do is buffer that packet for some period of time, hoping that the necessary prior packets eventually show up. When they do, you can then unbuffer the "newer" packets, thereby

minimizing the amount of latency incurred for a missed packet.

The rate at which you resend should be calculated based on the mean round-trip-time (RTT) between the sender and receiver. This allows you to back off on redeliveries as connections experience packet loss, etc.

Connection Quality

One of the things that players will want to know is the "quality" of their connection to a particular server. This is usually quantified in some form as latency/lag and packet loss.

Latency is the amount of time it takes to reach a server from a particular client. Longer latency means slower responsiveness. Packet loss is usually expressed as a percentage of packets that are dropped between the client and server, which manifests itself as little hiccups during play as data has to be resent.

Measuring latency is trivial. On each outgoing packet, simply embed the time the packet was sent as part of the header. When the packet is received, the receiver sends that time back as part of the acknowledgement, so when the sender receives that acknowledgement it can look at the sent time and compare it to the wallclock to get an idea for how long the round trip takes.

This requires slightly more data (the timestamp) per packet, so some people prefer to issue several ping commands per second to measure the on-going latency.

Packet loss can be measured in several different ways, most of them purely arbitrary -- pick a method (percentage of packets received out of order, number of resends required, etc.) and stick with it.

Collating Data into Large Packets

Every UDP packet that goes out is encumbered by a 22-byte private UDP header (stuck on there by the transport layer) that tells everyone between it and its destination where it's trying to go and how big it is. That's a pretty big chunk of space to send out for every packet.

One early optimization that many do is to collate packets into larger packets so that you're not constantly sending these headers all the time. They usually collate up to the "ideal" transmission size, which changes but, empirically speaking, seems to hover right around 1400 bytes.

Obviously you don't want collate for too long, since you can easily incur software induced lag as a result, so you should enforce a broadcast at some interval, e.g. every 100ms. Something else to consider is that users with modem connections may find that a 1400 byte packet takes far too long to broadcast (@ 56k, with a usable speed of 33k, it should take about 300ms to transmit a 1400 byte packet in a best case scenario). If modem connections are important, then a smaller buffer size of, say, 500 bytes might be more appropriate to reduce latency.

Fragmentation and Reassembly

The inverse problem to small, inefficient packets is that of needing to transmit large data chunks that don't fit inside the magical 1400 byte "ideal" MTU size. As a convenience it's nice to provide packet fragmentation and reassembly -- this makes the application programmer's life significantly easier.

How you implement this is up to you, but one direction to explore is to have "sub-sequence" nubmers inside your header, which indicate if the packet is actually part of a larger packet. When a receiver sees these, it knows to just grab those packets and reassemble the data as it arrives in fragments.

Polling vs. Multithreaded

Since data needs to be resent in order to ensure reliable delivery, and since you'll also want to see if new data is coming in so that you can send back reception acknowledgements, you'll need to pump your network code periodically.

One way to do this is to make sure you pump it every N milliseconds, for example at the top of your game's main loop. This is the simplest approach, and in fact the one I suggest for most situations, however it does have the tendency of being hiccup prone if a game loop suddenly takes an unexpectedly long duration (e.g. synchronously loading large assets from disk). (SOLUTION: don't perform any operations in your main loop that can halt the system for extended periods of time, since that will often screw up more than just your

network pump.)

A more complex, but more predictably timed, alternative is to have your network code running in a separate thread. Then you can `sleep()` and/or block pending new events, unaffected by the actions of the main loop. You'll then have to contend with race conditions and deadlocks, so designing your locking system well is critical. The complexities of doing this aren't to be underestimated, so only go this direction if you firmly know what you're doing. In the vast majority of cases, simply pumping your sockets once per "frame" works well enough.

Compression and Bandwidth Management

Network bandwidth can be consumed very quickly, both on the client side (modems) and on the server side (limited pipe). As a result, data compression is important.

From an architetural point of view, the very first thing you should do is minimize the amount of data that you need to send, period. Before any type of fancy compression schemes, ensure that only the data a client needs is actually received. If you have a large world, it is impractical to broadcast the status of all entities to all clients -- bandwidth usage will be beyond comprehension. Each client should have a specific "area of interest" along with some general state, and that is all the server should broadcast to that client. There is no need for someone fighting an orc in the Dungeon of Gloom to receive information about a conversation between a barkeep and patron at the Inn of Happiness six kilometers away.

Once you've determined the subset of relevant information that must be broadcast, you'll want to compress the actual data crossing the wire before the networking library even sees it. This means truncating values that don't need to be so large, etc. For example, if you don't truly need floating point position elements and can live just fine with 16-bit fixed point values, then use that. If you can live with 8-bit angle representations, then send bytes for that information, etc. If you have multiple bit flags, try to put them together into unused areas of your packet header.

Truncation and quantization should give some pretty decent compression, at least 25-50%. The next step is then to compress that data within the packet. Once again I'll bow out of any really detailed discussion, other than to say -- there's a lot of information out there about data compression. This should be worth a few more percentage points.

The third step you'll want to take is delta compression, or sending only changes instead of absolute state. For example, say you send over 12-bytes of data to represent position information. If you do that every frame, even if the player isn't moving, that's wasteful. It is much more efficient to broadcast the position when it has changed. Take that concept and expand it to entire state blocks, and that's what delta compression gives you.

Delta compression is done by keeping a copy of the local state and the last state that was successfully received by the other machine, and sending only the changed values and notating which values are being sent through a "delta flags" header.

Security and Encryption

One of the biggest concerns with on-line gaming has to do with cheating and security. One cheater can ruin the game for hundreds or even thousands of other players.

Cheating

Cheating can occur whenever players leverage information they should not have or, even worse, when they can alter events within the game because their client is authoritative in some area.

Access to ostensibly limited information is one of the easiest hacks someone can perform. This takes advantage of the fact that many games send more information than the client is required to know, and have the client filter this information out appropriately.

For example, certain players may be invisible. This information might be sent over to the client as "entity X is invisible, so don't draw him". By hacking or modifying the client, the cheater can now simply say "ignore invisibility flags" and know where invisible players are located.

Players may also use available information in order to build client-side assistants, or bots. For example, since

a player's heading is rarely sent to a server constantly (it changes too frequently), the server must ensure that the player's client has a full set of relevant information for his entire surrounding area. Someone can write a bot that displays a radar of the player's immediate surroundings in that type of situation, providing a huge tactical advantage.

Finally, there's the example where you have a client that is authoritative in some area, such as player movement. In those situations the player, with a suitably modified client, can pass outlandish values back to the server, giving him the ability to fly or run at ridiculously high speed. This is a serious problem for on-line worlds where there is a strong emphasis on game balance and perceived fairness. Ideally this is beaten by simply disallowing the client to be authoritative about anything, but sometimes this isn't a practical approach.

If the player is cheating by using a hacked client, i.e. the client itself has been modified, then there's not much you can do short of a bunch of checksums on the executable, etc. But this is a losing proposition -- if someone has Soft-ICE or a similar debugger hooked up to your program, you're going to lose in the long run.

Slightly easier to beat is when someone has installed a proxy, or a program that is intercepting network traffice between the server and the client. Proxies rely on knowing the format and contents of the packets sent between the client and server, so if you can remove this knowledge, you're part of the way there.

Which means encryption.

Encryption

Encrypting the traffic between a server and client is extremely important, not just for cheating, but for basic security. For example, if a player is sending personal information such as credit card numbers or passwords in cleartext, then someone using a simple packet sniffer in proximity to that player will be able to read this information trivially.

Even ignoring the issue of privacy, there's the problem of cheating. Cleartext packets are easy to examine and reverse engineer, so encryption is important to deter cheating via proxies/sniffers.

For our purposes, there are two key forms of encryption that we need to be concerned about -- symmetric key encryption and assymetric key encryption.

Symmetric key encryption works by sharing a single key between the client and server. Most symmetric key algorithms, such as Blowfish, are significantly faster than asymmetric key algorithms by several orders of magnitude.

Asymmetric key encryption no longer shares a single key. Instead, each party has a key used to encrypt and decrypt information. Possession of one key does not allow you to decrypt messages encrypted with that same key. Asymmetric key algorithms such as PGP and RSA are extremely expensive and are not suitable for packet encryption.

Ideally everything is encrypted with a symmetric key algorithm, but that presents a significant problem -- how do you exchange the key safely?

One mechanism is to somehow generate the key based off of client data known to the client and server both, for example the client's name and address. Both sides can generate an identical symmetric key based on this without having to exchange it in cleartext.

This works fine when trying to avoid packet sniffers, but it does not prevent the situation where someone is hacking the client directly, finds the key value, then stores it in the proxy so that it, too, can decrypt the packet stream. In addition, this requires a priori knowledge of client details on the part of the server, and for non-persistent games that don't require registration this may not be feasible.

So that's where asymmetric encryption comes in. With an asymmetric system, the server has a private key and a public key. The public key is well known to all clients. So the client randomly generates a per-session symmetric encryption key, encrypts with the server's public key, and sends that over.

Nothing in between the server and client can decrypt this data, since it would require knowledge of the server's private key, which is effectively under lock and key. When the packet arrives, the server decrypts it with its private key, retrieves the session key, and begins sending traffic back to the client using the symmetric encryption algorithm.

This provides a good balance of security and performance. In theory the session key can still be retrieved, but it would require an actively hacked client instead of just hacking the client (or its data files) a single time and storing the key. It's far from fool proof, but it raises the bar significantly.

Portability Issues

When dealing with networked multiplayer games, there is a chance (if you support it) that you will be exchanging information between heterogenous systems. For example, your servers may be running on Sparc/Solaris, but your clients might be on x86/Windows.

Most issues to do with portability apply just as strictly when dealing with the network as they do with data files. Just like persisting data to disk, you have to be aware of size, endianess and alignment issues when persisting data to the network. In addition, keep in mind that some areas -- such as floating point -- may not evaluate the same on two systems with different processors or that had clients compiled with different compilers.

Because endianess plays such a key role in network communications (and UDP is big-endian internally), there are several socket library functions designed to convert from "network" endianess (big-endian) to host endianess. These functions are:

- `ntohl()` - network-to-host byte ordering, long (32-bits)
- `ntohs()` - network-to-host byte ordering, short (16-bits)
- `htonl()` - host-to-network byte ordering, long (32-bits)
- `htons()` - host-to-network byte ordering, short (16-bits)

For a more robust solution, I would recommend that you handle endianess issues at a higher level, using your own macros/functions. Alternatively, you can look at the POSH headers.

Summary

Multiplayer network game programming is an extremely complex and daunting task, because it's difficult to learn "just part of it". There are so many interrelated issues that without a gestalt understanding of the entire situation, it almost seems like an impossible task at first.

The goal of this document is to provide a high-altitude view of how the different pieces in a networked game environment come together. While no actual source code is provided, that is rarely going to be the difference between success or failure -- understanding the key topics is really the important part, because the code can and will change depending on your design criteria.

Contributors

Thanks to Jeremy Noetzelman, Bruce Mitchener, Jon Watte and Jonathan Blow for feedback and comments. Thanks to Gavin Doughtie for pointing out some grammatical errors.

Recommended Reading

Stevens, W. Richard, "UNIX Network Programming, Vol. 1", Pearson Education