gafferongames.com

Virtual Connection over UDP

9 min read • original

Introduction

Hi, I'm Glenn Fiedler. Welcome to the third article in Networking for Game Programmers.

In the previous article, I showed you how to send and receive packets using UDP.

Since UDP is connectionless, one UDP socket can be used to exchange packets with any number of different computers. In multiplayer games however, we usually only want to exchange packets between a small set of connected computers.

As the first step towards a general connection system, we'll start with the simplest case possible: creating a virtual connection between two computers on top of UDP.

But first, we're going to dig in a bit deeper about how the Internet really works!

The Internet not a series of tubes

In 2006, Senator Ted Stevens made internet history with his famous speech on the net neutrality act:

> "The internet is not something that you just dump something on. It's not a big truck. It's a series of tubes"

When I first started using the Internet, I was just like Ted. Sitting in the computer lab in University of Sydney in 1995, I was "surfing the web" with this new thing called Netscape Navigator, and I had absolutely no idea what was going on.

You see, I thought each time you connected to a website there was some actual connection going on, like a telephone line. I wondered, how much does it cost each time I connect to a new website? 30 cents? A dollar? Was somebody from the university going to tap me on the shoulder and ask me to pay the long distance charges? ☺

Of course, this all seems silly now.

There is no switchboard somewhere that directly connects you via a physical phone line to the other computer you want to talk to, let alone a series of pneumatic tubes like Sen. Stevens would have you believe.

No direct connections

Instead your data is sent over Internet Protocol (IP) via packets that hop from computer to computer.

A packet may pass through several computers before it reaches its destination. You cannot know the exact set of computers in advance, as it changes dynamically depending on how the network decides to route packets. You could even send two packets A and B to the same address, and they may take different routes. This is the source of the non-guaranteed order of packet delivery in UDP by the way.

On unix-like systems can inspect the route that packets take by calling "traceroute" and passing in a destination hostname or IP address.

On windows, replace "traceroute" with "tracert" to get it to work.

Try it with a few websites like this:

```
traceroute slashdot.org
traceroute amazon.com
traceroute google.com
traceroute bbc.co.uk
traceroute news.com.au
```

Take a look and you should be able to convince yourself pretty quickly that there is no direct connection.

How packets get delivered

In the first article, I presented a simple analogy for packet delivery, describing it as somewhat like a note being passed from person to person across a crowded room.

While this analogy gets the basic idea across, it is much too simple. The Internet is not a flat network of computers, it is a network of networks. And of course, we don't just need to pass letters around a small room, we need to be able to send them anywhere in the world.

It should be pretty clear then that the best analogy is the postal service!

When you want to send a letter to somebody you put your letter in the mailbox and you trust that it will be delivered correctly. It's not really relevant to you *how* it gets there, as long as it does. Somebody has to physically deliver your letter to its destination of course, so how is this done?

Well first off, the postman sure as hell doesn't take your letter and deliver it personally! It seems that the postal service is not a series of tubes either. Instead, the postman takes your letter to the local post office for processing.

If the letter is addressed locally then the post office just sends it back out, and another postman delivers it directly. But, if the address is is non-local then it gets interesting! The local post office is not able to

deliver the letter directly, so it passes it "up" to the next level of hierarchy, perhaps to a regional post office which services cities nearby, or maybe to a mail center at an airport, if the address is far away. Ideally, the actual transport of the letter would be done using a big truck.

Lets be complicated and assume the letter is sent from Los Angeles to Sydney, Australia. The local post office receives the letter and given that it is addressed internationally, sends it directly to a mail center at LAX. The letter is processed again according to address, and gets routed on the next flight to Sydney.

The plane lands at Sydney airport where an *entirely different postal system* takes over. Now the whole process starts operating in reverse. The letter travels "down" the hierarchy, from the general, to the specific. From the mail hub at Sydney Airport it gets sent out to a regional center, the regional center delivers it to the local post office, and eventually the letter is hand delivered by a mailman with a funny accent. Crikey! ☺

Just like post offices determine how to deliver letters via their address, networks deliver packets according to their IP address. The low-level details of this delivery and the actual routing of packets from network to network is actually quite complex, but the basic idea is that each router is just another computer, with a routing table describing where packets matching sets of addresses should go, as well as a default gateway address describing where to pass packets for which there is no matching entry in the table. It is routing tables, and the physical connections they represent that define the network of networks that is the Internet.

The job of configuring these routing tables is up to network administrators, not programmers like us. But if you want to read more about it, then this article from ars technica provides some fascinating insight into how networks exchange packets between each other via

peering and transit relationships. You can also read more details about routing tables in this linux faq, and about the border gateway protocol on wikipedia, which automatically discovers how to route packets between networks, making the internet a truly distributed system capable of dynamically routing around broken connectivity.

Virtual connections

Now back to connections.

If you have used TCP sockets then you know that they sure *look* like a connection, but since TCP is implemented on top of IP, and IP is just packets hopping from computer to computer, it follows that TCP's concept of connection must be a *virtual connection*.

If TCP can create a virtual connection over IP, it follows that we can do the same over UDP.

Lets define our virtual connection as two computers exchanging UDP packets at some fixed rate like 10 packets per-second. As long as the packets are flowing, we consider the two computers to be virtually connected.

Our connection has two sides:

- One computer sits there and *listens* for another computer to connect to it. We'll call this computer the server.
- Another computer *connects* to a server by specifying an IP address and port. We'll call this computer the client.

In our case, we only allow one client to connect to the server at any time. We'll generalize our connection system to support multiple simultaneous connections in a later article. Also, we assume that the IP address of the server is on a fixed IP address that the client may directly connect to. We'll cover matchmaking and NAT punch-through in later articles.

Protocol id

Since UDP is connectionless our UDP socket can receive packets sent from any computer.

We'd like to narrow this down so that the server only receives packets sent from the client, and the client only receives packets sent from the server. We can't just filter out packets by address, because the server doesn't know the address of the client in advance. So instead, we prefix each UDP packet with small header containing a 32 bit protocol id as follows:

```
[uint protocol id]
(packet data...)
```

The protocol id is just some unique number representing our game protocol. Any packet that arrives from our UDP socket first has its first four bytes inspected. If they don't match our protocol id, then the packet is ignored. If the protocol id does match, we strip out the first four bytes of the packet and deliver the rest as payload.

You just choose some number that is reasonably unique, perhaps a hash of the name of your game and the protocol version number. But really you can use anything. The whole point is that from the point of view of our connection based protocol, packets with different protocol ids are ignored.

Detecting connection

Now we need a way to detect connection.

Sure we could do some complex handshaking involving multiple UDP packets sent back and forth. Perhaps a client "request connection" packet is sent to the server, to which the server responds with a "connection accepted" sent back to the client, or maybe an "i'm busy" packet if a client tries to connect to server which already has a connected client.

Or... we could just setup our server to take the first packet it receives with the correct protocol id, and consider a connection to be established.

The client just starts sending packets to the server assuming connection, when the server receives the first packet from the client, it takes note of the IP address and port of the client, and starts sending packets back.

The client already knows the address and port of the server, since it was specified on connect. So when the client receives packets, it filters out any that don't come from the server address. Similarly, once the server receives the first packet from the client, it gets the address and port of the client from "recvfrom", so it is able to ignore any packets that don't come from the client address.

We can get away with this shortcut because we only have two computers involved in the connection. In later articles, we'll extend our connection system to support more than two computers in a client/server or peer-to-peer topology, and at this point we'll upgrade our connection negotiation to something more robust.

But for now, why make things more complicated than they need to be?

Detecting disconnection

How do we detect disconnection?

Well if a connection is defined as receiving packets, we can define disconnection as *not* receiving packets.

To detect when we are not receiving packets, we keep track of the number of seconds since we last received a packet from the other side of the connection. We do this on both sides.

Each time we receive a packet from the other side, we reset our accumulator to 0.0, each update we increase the accumulator by the amount of time that has passed.

If this accumulator exceeds some value like 10 seconds, the connection "times out" and we disconnect.

This also gracefully handles the case of a second client trying to connect to a server that has already made a connection with another client. Since the server is already connected it ignores packets coming from any address other than the connected client, so the second client receives no packets in response to the packets it sends, so the second client times out and disconnects.

Conclusion

And that's all it takes to setup a virtual connection: some way to establish connection, filtering for packets not involved in the connection, and timeouts to detect disconnection.

Our connection is as real as any TCP connection, and the steady stream of UDP packets it provides is a suitable starting point for a multiplayer action game.

We also have some insight into how the Internet routes packets. For example, we now know the reason UDP packets sometimes arrive out of order is because they took different routes over IP! Take a look at a map of the internet, isn't it amazing that your packets arrive at all? If you would like to dig a bit deeper into all this, a great starting point is this article on wikipedia.

Now that you have your virtual connection over UDP, you can easily setup a client/server relationship for a two player multiplayer game without needing to use TCP.

You can see an implementation of this in the example source code for this article.

It's a simple client/server program that exchanges 30 packets per-second. You can run the server on any machine you like, provided it has a public IP address, because we don't support NAT punch-through at this point.

Run the client like this:

```
./Client 205.10.40.50
```

And it will attempt to connect to the address you specify on the command line. By default it attempts to connect to 127.0.0.1 if you don't specify the address.

While one client is connected you can try to connect another, and you'll notice that it fails to connect. This is intentional. For the moment, only one client may be connected at a time.

You could also try stopping the client or server while they are connected and you should notice that after 10 seconds the other side will timeout and disconnect. When the client times out it exits to the shell, but the server returns to a listening state ready for another client to connect.

If you enjoyed this article please consider making a small donation.
Donations encourage me to write more articles!

Original URL:

http://gafferongames.com/networking-for-game-programmers/virtual-connection-over-udp/