UDP vs. TCP

Oct 1, 2008 · 13 min read · Game Networking

Introduction

Hi, I'm Glenn Fiedler and welcome to **Networking for Game Programmers**.

In this article we start with the most basic aspect of network programming: sending and receiving data over the network. This is perhaps the simplest and most basic part of what network programmers do, but still it is quite intricate and non-obvious as to what the best course of action is.

You have most likely heard of sockets, and are probably aware that there are two main types: TCP and UDP. When writing a network game, we first need to choose what type of socket to use. Do we use TCP sockets, UDP sockets or a mixture of both? Take care because if you get this wrong it will have terrible effects on your multiplayer game!

The choice you make depends entirely on what sort of game you want to network. So from this point on and for the rest of this article series, I assume you want to network an action game. You know, games like Halo, Battlefield 1942, Quake, Unreal, CounterStrike and Team Fortress.

In light of the fact that we want to network an action game, we'll take a very close look at the properties of each protocol, and dig a bit into how the internet actually works. Once we have all this information, the correct choice is clear.

TCP/IP

TCP stands for "transmission control protocol". IP stands for "internet protocol". Together they form the backbone for almost everything you do online, from web browsing to IRC to email, it's all built on top of TCP/IP.

If you have ever used a TCP socket, then you know it's a reliable connection based protocol. This means you create a connection between two machines, then you exchange data much like you're writing to a file on one side, and reading from a file on the other.

TCP connections are reliable and ordered. All data you send is guaranteed to arrive at the other side and in the order you wrote it. It's also a stream protocol, so TCP automatically splits your data into packets and sends them over the network for you.

IP

The simplicity of TCP is in stark contrast to what actually goes on underneath TCP at the IP or "internet protocol" level.

Here there is no concept of connection, packets are simply passed from one computer to the next. You can visualize this process being somewhat like a hand-written note passed from one person to the next across a crowded room, eventually, reaching the person it's addressed to, but only after passing through many hands.

There is also no guarantee that this note will actually reach the person it is intended for. The sender just passes the note along and hopes for the best, never knowing whether or not the note was received, unless the other person decides to write back!

Of course IP is in reality a little more complicated than this, since no one computer knows the exact sequence of computers to pass the packet along to so that it reaches its destination quickly. Sometimes IP passes along multiple copies of the same packet and these packets make their way to the destination via different paths, causing packets to arrive out of order and in duplicate.

This is because the internet is designed to be self-organizing and self-repairing, able to route around connectivity problems rather than relying on direct connections between computers. It's actually quite cool if you think about what's *really* going on at the low level. You can read all about this in the classic book TCP/IP Illustrated.

UDP

Instead of treating communications between computers like writing to files, what if we want to send and receive packets directly?

We can do this using UDP.

UDP stands for “user datagram protocol” and it’s another protocol built on top of IP, but unlike TCP, instead of adding lots of features and complexity, UDP is a very thin layer over IP.

With UDP we can send a packet to a destination IP address (eg. 112.140.20.10) and port (say 52423), and it gets passed from computer to computer until it arrives at the destination or is lost along the way.

On the receiver side, we just sit there listening on a specific port (eg. 52423) and when a packet arrives from *any* computer (remember there are no connections!), we get notified of the address and port of the computer that sent the packet, the size of the packet, and can read the packet data.

Like IP, UDP is an unreliable protocol. In practice however, most packets that are sent *will* get through, but you’ll usually have around 1-5% packet loss, and occasionally you’ll get periods where no packets get through at all (remember there are lots of computers between you and your destination where things can go wrong…)

There is also no guarantee of ordering of packets with UDP. You could send 5 packets in order 1,2,3,4,5 and they could arrive completely out of order like 3,1,2,5,4. In practice, packets tend to arrive in order *most* of the time, but you cannot rely on this!

UDP also provides a 16 bit checksum, which in theory is meant to protect you from receiving invalid or truncated data, but you can’t even trust this, since 16 bits is just not enough protection when you are sending UDP packets rapidly over a long period of time. Statistically, you can’t even rely on this checksum and must add your own.

So in short, when you use UDP you’re pretty much on your own!

TCP vs. UDP

We have a decision to make here, do we use TCP sockets or UDP sockets?

Lets look at the properties of each:

TCP:

- Connection based
- Guaranteed reliable and ordered
- Automatically breaks up your data into packets for you
- Makes sure it doesn't send data too fast for the internet connection to handle (flow control)
- Easy to use, you just read and write data like its a file

UDP:

- No concept of connection, you have to code this yourself
- No guarantee of reliability or ordering of packets, they may arrive out of order, be duplicated, or not arrive at all!
- You have to manually break your data up into packets and send them
- You have to make sure you don't send data too fast for your internet connection to handle
- If a packet is lost, you need to devise some way to detect this, and resend that data if necessary
- You can't even rely on the UDP checksum so you must add your own

The decision seems pretty clear then, TCP does everything we want and its super easy to use, while UDP is a huge pain in the ass and we have to code everything ourselves from scratch.

So obviously we just use TCP right?

Wrong!

Using TCP is the worst possible mistake you can make when developing a multiplayer game! To understand why, you need to see what TCP is actually doing above IP to make everything look so simple.

How TCP really works

TCP and UDP are both built on top of IP, but they are radically different. UDP behaves very much like the IP protocol underneath it, while TCP abstracts everything so it looks like you are reading and writing to a file, hiding all complexities of packets and unreliability from you.

So how does it do this?

Firstly, TCP is a stream protocol, so you just write bytes to a stream, and TCP makes sure that they get across to the other side. Since IP is built on packets, and TCP is built on top of IP, TCP must therefore break your stream of data up into packets. So, some internal TCP code queues up the data you send, then when enough data is pending the queue, it sends a packet to the other machine.

This can be a problem for multiplayer games if you are sending very small packets. What can happen here is that TCP may decide it's not going to send data until you have buffered up enough data to make a reasonably sized packet to send over the network.

This is a problem because you want your client player input to get to the server *as quickly as possible*, if it is delayed or "clumped up" like TCP can do with small packets, the client's user experience of the multiplayer game will be very poor. Game network updates will arrive late and infrequently, instead of on-time and frequently like we want.

TCP has an option to fix this behavior called TCP_NODELAY. This option instructs TCP not to wait around until enough data is queued up, but to flush any data you write to it immediately. This is referred to as disabling Nagle's algorithm.

Unfortunately, even if you set this option TCP still has serious problems for multiplayer games and it all stems from how TCP handles lost and out of order packets to present you with the "illusion" of a reliable, ordered stream of data.

How TCP implements reliability

Fundamentally TCP breaks down a stream of data into packets, sends these packets over unreliable IP, then takes the packets received on the other side and reconstructs the stream.

But what happens when a packet is lost?

What happens when packets arrive out of order or are duplicated?

Without going too much into the details of how TCP works because its super-complicated (please refer to TCP/IP Illustrated) in essence TCP sends out a packet, waits a while until it detects that packet was lost because it didn't receive an ack (or acknowledgement), then resends the lost packet to the other machine. Duplicate packets are discarded on the receiver side, and out of order packets are resequenced so everything is reliable and in order.

The problem is that if we were to send our time critical game data over TCP, whenever a packet is dropped it has to stop and wait for that data to be resent. Yes, even if more recent data arrives, that new data gets put in a queue, and you cannot access it until that lost packet has been retransmitted. How long does it take to resend the packet?

Well, it's going to take *at least* round trip latency for TCP to work out that data needs to be resent, but commonly it takes 2*RTT, and another one way trip from the sender to the receiver for the resent packet to get there. So if you have a 125ms ping, you'll be waiting roughly 1/5th of a second for the packet data to be resent *at best*, and in worst case conditions you could be waiting up to half a second or more (consider what happens if the attempt to resend the packet fails to get through?). What happens if TCP decides the packet loss indicates network congestion and it backs off? Yes it actually does this. Fun times!

Never use TCP for time critical data

The problem with using TCP for realtime games like FPS is that unlike web browsers, or email or most other applications, these multiplayer games have a *real time requirement* on packet delivery.

What this means is that for many parts of a game, for example player input and character positions, it really doesn't matter what happened a second ago, the game only cares about the most recent data.

TCP was simply not designed with this in mind.

Consider a very simple example of a multiplayer game, some sort of action game like a shooter. You want to network this in a very simple way. Every frame you send the input from the client to the server (eg. keypresses, mouse input controller input), and each frame the server processes the input from each player, updates the simulation, then sends the current position of game objects back to the client for rendering.

So in our simple multiplayer game, whenever a packet is lost, everything has to *stop and wait* for that packet to be resent. On the client game objects stop receiving updates so they appear to be standing still, and on the server input stops getting through from the client, so the players cannot move or shoot. When the resent packet finally arrives, you receive this stale, out of date information that you don't even care about! Plus, there are packets backed up in queue waiting for the resend which arrive at same time, so you have to process all of these packets in one frame. Everything is clumped up!

Unfortunately, there is nothing you can do to fix this behavior, it's just the fundamental nature of TCP. This is just what it takes to make the unreliable, packet-based internet look like a reliable-ordered stream.

Thing is we don't want a reliable ordered stream.

We want our data to get as quickly as possible from client to server without having to wait for lost data to be resent.

This is why you should **never** use TCP when networking time-critical data!

Wait? Why can't I use *both* UDP and TCP?

For realtime game data like player input and state, only the most recent data is relevant, but for other types of data, say perhaps a sequence of commands sent from one machine to another, reliability and ordering can be very important.

The temptation then is to use UDP for player input and state, and TCP for the reliable ordered data. If you're sharp you've probably even worked out that you may have multiple "streams" of reliable ordered commands, maybe one about level loading, and another about AI. Perhaps you think to yourself, "Well, I'd really not want AI commands to stall out if a packet is lost containing a level loading command - they are completely unrelated!". You are right, so you may be tempted to create one TCP socket for each stream of commands.

On the surface, this seems like a great idea. The problem is that since TCP and UDP are both built on top of IP, the underlying packets sent by each protocol will affect each other. Exactly how they affect each other is quite complicated and relates to how TCP performs reliability and flow control, but fundamentally you should remember that TCP tends to *induce* packet loss in UDP packets. For more information, read this paper on the subject.

Also, it's pretty complicated to mix UDP and TCP. If you mix UDP and TCP you lose a certain amount of control. Maybe you can implement reliability in a more efficient way that TCP does, better suited to your needs? Even if you need reliable-ordered data, it's possible, provided that data is small relative to the available bandwidth to get that data across faster and more reliably that it would if you sent it over TCP. Plus, if you have to do NAT to enable home internet connections to talk to each other, having to do this NAT once for UDP and once for TCP (not even sure if this is possible...) is kind of painful.

Conclusion

My recommendation is not only that you use UDP, but that you *only* use UDP for your game protocol. Don't mix TCP and UDP! Instead, learn how to implement the specific features of TCP that you need *inside* your own custom UDP based protocol.

Of course, it is no problem to use HTTP to talk to some RESTful services while your game is running. I'm not saying you can't do that. A few TCP connections running while your game is running isn't going to bring everything down. The point is, don't split your *game protocol* across UDP and TCP. Keep your game protocol running over UDP so you are fully in control of the data you send and receive and how reliability, ordering and congestion avoidance are implemented.

The rest of this article series show you how to do this, from creating your own virtual connection on top of UDP, to creating your own reliability, flow control and congestion avoidance.

NEXT ARTICLE: Sending and Receiving Packets

Glenn Fiedler is the founder and CEO of **Network Next**.
Network Next is fixing the internet for games by creating a marketplace for premium network transit.

networking

Related

- Networked Physics (2004)