Gaffer on Games | What every programmer needs to know about game networking

Introduction

You're a programmer. Have you ever wondered how multiplayer games work?

From the outside it seems magical: two or more players sharing a consistent experience across the network like they actually exist together in the same virtual world. But as programmers we know the truth of what is actually going on underneath is quite different from what you see. It turns out that it's all an illusion. A massive sleight-of-hand. What you perceive as a shared reality is only an approximation unique to your own point of view and place in time.

Peer-to-Peer Lockstep

In the beginning games were networked peer-to-peer, with each each computer exchanging information with each other in a fully connected mesh topology. You can still see this model alive today in RTS games, and interestingly for some reason, perhaps because it was the first way – it's still how most people think that game networking works.

The basic idea is to abstract the game into a series of turns and a set of command messages when processed at the beginning of each turn direct the evolution of the game state. For example: move unit, attack unit, construct building. All that is needed to network this is to run exactly the same set of commands and turns on each player's machine starting from a common initial state.

Of course this is an overly simplistic explanation and glosses over many subtle points, but it gets across the basic idea of how networking for RTS games work. You can read more about this networking model here: 1500 Archers on a 28.8: Network Programming in Age of Empires and Beyond.

It seems so simple and elegant, but unfortunately there are several limitations.

First, it's exceptionally difficult to ensure that a game is completely deterministic; that each turn plays out identically on each machine. For example, one unit could take slightly a different path on two machines, arriving sooner to a battle and saving the day on one machine, while arriving later on the other and erm. not saving the day. Like a butterfly flapping it's wings and causing a hurricane on the other side of the world, one *tiny difference* results in complete desynchronization over time.

The next limitation is that in order to ensure that the game plays out identically on all machines it is necessary to wait until all player's commands for that turn are received *before* simulating that turn. This means that each player in the game has latency equal to the most lagged player. RTS games typically hide this by providing audio feedback immediately and/or playing cosmetic animation, but ultimately any truly game affecting action may occur only after this delay has passed.

The final limitation occurs because of the way the game synchronizes by sending just the command messages which change the state. In order for this to work it is necessary for all players to start from the same initial state. Typically this means that each player must join up in a lobby before commencing play, although it is technically possible to support late join, this is not common due to the difficulty of capturing and transmitting a completely deterministic starting point in the middle of a live game.

Despite these limitations this model naturally suits RTS games and it still lives on today in games like "Command and Conquer", "Age of Empires" and "Starcraft". The reason being that in RTS games the game state consists of many thousands of units and is simply too large to exchange between players. These games have no choice but to exchange the commands which drive the evolution of the game state.

But for other genres, the state of the art has moved on. So that's it for the deterministic peer-to-peer lockstep networking model. Now lets look at the evolution of action games starting with Doom, Quake and Unreal.

Client/Server

In the era of action games, the limitations of peer-to-peer lockstep became apparent in Doom, which despite playing well over the LAN played *terribly* over the internet for typical users:

> *Although it is possible to connect two DOOM machines together across the Internet using a modem link, the resulting game will be slow, ranging from the unplayable (e.g. a 14.4Kbps PPP connection) to the marginally playable (e.g. a 28.8Kbps modem running a Compressed SLIP driver). Since these sorts of connections are of only marginal utility, this document will focus only on direct net connections. (faqs.org)*

The problem of course was that Doom was designed for networking over LAN only, and used the peer-to-peer lockstep model described previously for RTS games. Each turn player inputs (key presses etc.) were exchanged with other peers, and before any player could simulate a frame all other player's key presses needed to be received.

In other words, before you could turn, move or shoot you had to wait for the inputs from the most lagged modem player. Just imagine the wailing and gnashing of teeth that this would have resulted in for the sort of folks who wrote above that "these sorts of connections are of only marginal utility". 🙂

In order to move beyond the LAN and the well connected elite at university networks and large companies, it was necessary to change the model. And in 1996, that's exactly what John Carmack did when he released Quake using client/server instead of peer-to-peer.

Now instead of each player running the same game code and communicating directly with each other, each player was now a "client" and they all communicated with just one computer called the "server". There was no longer any need for the game to be deterministic across all machines, because the game really only existed on the server. Each client effectively acted as a *dumb terminal* showing an approximation of the game as it played out on the server.

In a pure client/server model you run no game code locally, instead sending your inputs such as key presses, mouse movement, clicks to the server. In response the server updates the state of your character in the world and replies with a packet containing the state of your character and other players near you. All the client has to do is interpolate between these updates to provide the illusion of smooth movement and *BAM* you have a networked client/server game.

This was a great step forward. The quality of the game experience now depended on the connection between the client and the server instead of the most lagged peer in the game. It also became possible for players to come and go in the middle of the game, and the number of players increased as client/server reduced the bandwidth required on average per-player.

But there were still problems with the pure client/server model:

> *While I can remember and justify all of my decisions about networking from DOOM through Quake, the bottom line is that I was working with the wrong basic assumptions for doing a good internet game. My original design was targeted at < 200ms connection latencies. People that have a digital connection to the internet through a good provider get a pretty good game experience. Unfortunately, 99% of the world gets on with a slip or ppp connection over a modem, often through a crappy overcrowded ISP. This gives 300+ ms latencies, minimum. Client. User's modem. ISP's modem. Server. ISP's modem. User's modem. Client. God, that sucks.*
>
> *Ok, I made a bad call. I have a T1 to my house, so I just wasn't familliar with PPP life. I'm addressing it now.*

The problem was of course latency.

What John did next when he released QuakeWorld would change the industry forever.

Client-Side Prediction

In the original Quake you felt the latency between your computer and the server. Press forward and you'd wait however long it took for packets to travel to the server and back to you before you'd actually start moving. Press fire and you wait for that same delay before shooting.

If you've played any modern FPS like Call of Duty: Modern Warfare, you know this is no longer what happens. So how exactly do modern FPS games seem to remove the latency on your own actions in multiplayer?

This problem was historically solved in two parts. The first part was client-side prediction of movement developed by John Carmack for QuakeWorld, and later incorporated as part of Unreal's networking model by Tim Sweeney. The second part was latency compensation developed by Yahn Bernier at Valve for Counterstrike. In this section we'll focus on that first part – hiding the latency on player movement.

When writing about his plans for the soon to be released QuakeWorld, John Carmack said:

> *I am now allowing the client to guess at the results of the users movement until the authoritative response from the server comes through. This is a biiiig architectural change. The client now needs to know about solidity of objects, friction, gravity, etc. I am sad to see the elegant client-as-terminal setup go away, but I am practical above idealistic.*

So now in order to remove the latency, the client runs more code than it previously did. It is no longer a dumb terminal sending inputs to the server and interpolating between state sent back. Instead it is able to predict the movement of your character locally and *immediately* in response to your input, running a subset of the game code for your player character on the client machine.

Now as soon as you press forward, there is no wait for a round trip between client and server – your character start moving forward right away.

The difficulty of this approach is not in the prediction, for the prediction works just as normal game code does – evolving the state of the game character forward in time according to the player's input. The difficulty is in applying the correction back from the server to resolve cases when the client and server disagree about where the player character should be and what it is doing.

Now at this point you might wonder. Hey, if you are running code on the client – why not just make the client authoritative over their player character? The client could run the simulation code for their own character and simply tell the server where they are each time they send a packet. The problem with this is that if each player were able to simply tell the server "here is my current position" it would be trivially easy to hack the client such that a cheater could instantly dodge the RPG about to hit them, or teleport instantly behind you to shoot you in the back.

So in FPS games it is absolutely necessary that the server is the authoritative over the state of each player character, in-spite of the fact that each player is locally predicting the motion of their own character to hide latency. As Tim Sweeney writes in The Unreal Networking Architecture: "The Server Is The Man".

Here is where it gets interesting. If the client and the server disagree, the client *must* accept the update for the position from the server, but due to latency between the client and server this correction is necessarily in the past. For example, if it takes 100ms from client to server and 100ms back, then any server correction for the player character position will appear to be 200ms in the past, relative to the time up to which the client has predicted their own movement.

If the client were to simply apply this server correction update verbatim, it would yank the client back in time such that the client would completely undo any client-side prediction. How then to solve this while still allowing the client to predict ahead?

The solution is to keep a circular buffer of past character state and input for the local player on the client, then when the client receives a correction from the server, it first discards any buffered state older than the corrected state from the server, and replays the state starting from the corrected state back to the present "predicted" time on the client using player inputs stored in the circular buffer. In effect the client invisibly "rewinds and replays" the last n frames of local player character movement while holding the rest of the world fixed.

This way the player appears to control their own character without any latency, and provided that the client and server character simulation code is deterministic – giving exactly the same result for the same inputs on the client and server – it is rarely corrected. It is as Tim Sweeney describes:

> *... the best of both worlds: In all cases, the server remains completely authoritative. Nearly all the time, the client movement simulation exactly mirrors the client movement carried out by the server, so the client's position is seldom corrected. Only in the rare case, such as a player getting hit by a rocket, or bumping into an enemy, will the client's location need to be corrected.*

In other words, only when the player's character is affected by something external to the local player's input, which cannot possibly be predicted on the client, will the player's position need to be corrected. That and of course, if that player is attempting to cheat 🙂

Next: Introduction To Networked Physics