<< Series Start

Gabriel Gambetta

Fast-Paced Multiplayer (Part I): Client-Server Game Architecture

Introduction

This is the first in a series of articles exploring the techniques and algorithms that make fast-paced multiplayer games possible. If you're familiar with the concepts behind multiplayer games, you can safely skip to the next article – what follows is an introductory discussion.

Developing any kind of game is itself challenging; multiplayer games, however, add a completely new set of problems to be dealt with. Interestingly enough, the core problems are human nature and physics!

The problem of cheating

It all starts with cheating.

As a game developer, you usually don't care whether a player cheats in your single-player game – his actions don't affect anyone but him. A cheating player may not experience the game exactly as you planned, but since it's their game, they have the right to play it in any way they please.

Multiplayer games are different, though. In any competitive game, a cheating player isn't just making the experience better for himself, he's also making the experience worse for the other players. As the developer, you probably want to avoid that, since it tends to drive players away from your game.

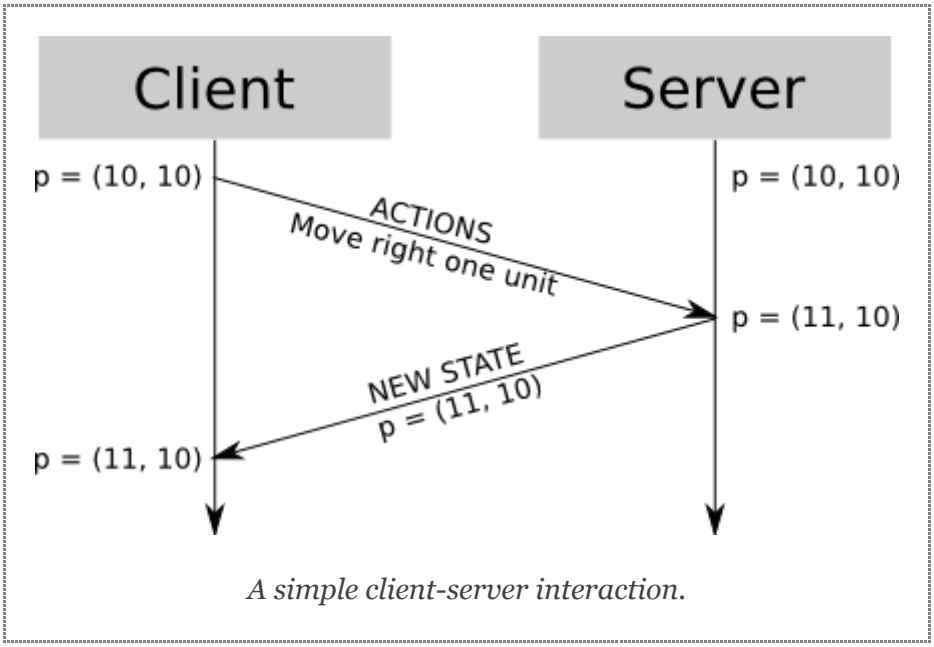
There are many things that can be done to prevent cheating, but the most important one (and probably the only really meaningful one) is simple : *don't trust the player*. Always assume the worst – that players *will* try to cheat.

Authoritative servers and dumb clients

This leads to a seemingly simple solution – you make everything in your game happen in a central server under your control, and make the clients just privileged spectators of the game. In other words, your game client sends inputs (key presses, commands) to the server, the server runs the game, and you send the results back to the clients. This is usually called using an *authoritative server*, because the one and only authority regarding everything that happens in the world is the server.

Of course, your server can be exploited for vulnerabilities, but that's out of the scope of this series of articles. Using an authoritative server does prevent a wide range of hacks, though. For example, you don't trust the client with the health of the player; a hacked client can modify its local copy of that value and tell the player it has 10000% health, but the server *knows* it only has 10% – when the player is attacked it will die, regardless of what a hacked client may think.

You also don't trust the player with its position in the world. If you did, a hacked client would tell the server "**I'm at (10,10)**" and a second later "**I'm at (20,10)**", possibly going through a wall or moving faster than the other players. Instead, the server *knows* the player is at (10,10), the client tells the server "**I want to move one square to the right**", the server updates its internal state with the new player position at (11,10), and then replies to the player "**You're at (11, 10)**":



A simple client-server interaction.

In summary: the game state is managed by the server alone. Clients send their actions to the server. The server updates the game state periodically, and then sends the new game state back to clients, who just render it on the screen.

Dealing with networks

The dumb client scheme works fine for slow turn based games, for example strategy games or poker. It would also work in a LAN setting, where communications are, for

all practical purposes, instantaneous. But this breaks down when used for a fast-paced game over a network such as the internet.

Let's talk physics. Suppose you're in San Francisco, connected to a server in the NY. That's approximately 4,000 km, or 2,500 miles (that's roughly the distance between Lisbon and Moscow). Nothing can travel faster than light, not even bytes on the Internet (which at the lower level are pulses of light, electrons in a cable, or electromagnetic waves). Light travels at approximately 300,000 km/s, so it takes 13 ms to travel 4,000 km.

This may sound quite fast, but it's actually a very optimistic setup – it assumes data travels at the speed of light in a straight path, with is most likely not the case. In real life, data goes through a series of jumps (called *hops* in networking terminology) from router to router, most of which aren't done at lightspeed; routers themselve introduce a bit of delay, since packets must be copied, inspected, and rerouted.

For the sake of the argument, let's assume data takes 50 ms from client to server. This is close to a best-case scenario – what happens if you're in NY connected to a server in Tokyo? What if there's network congestion for some reason? Delays of 100, 200, even 500 ms are not unheard of.

Back to our example, your client sends some input to the server ("**I pressed the right arrow**"). The server gets it 50 ms later. Let's say the server processes the request and sends back the updated state immediately. Your client gets the new game state ("**You're now at (1, 0)**") 50 ms later.

From your point of view, what happened is that you pressed the right arrow but nothing happened for a tenth of a second; then your character finally moved one square to the right. This perceived *lag* between your inputs and its consequences may not sound like much, but it's noticeable – and of course, a lag of half a second isn't just noticeable, it actually makes the game unplayable.

Summary

Networked multiplayer games are incredibly fun, but introduce a whole new class of challenges. The authoritative server architecture is pretty good at stopping most cheats, but a straightforward implementation may make games quite unresponsive to the player.

In the following articles, we'll explore how can we build a system based on an authoritative server, while minimizing the delay experienced by the players, to the point of making it almost indistinguishable from local or single player games.

Part II: Client-Side Prediction and Server Reconciliation >>