<< Series Start

Gabriel Gambetta

Fast-Paced Multiplayer (Part III): Entity Interpolation

Client-Server Game Architecture | Client-Side Prediction and Server Reconciliation | Entity Interpolation | Lag Compensation | Live Demo

Translations: Korean | Russian

Introduction

In the first article of the series, we introduced the concept of an *authoritative server* and its usefulness to prevent client cheats. However, using this technique naively can lead to potentially showstopper issues regarding playability and responsiveness. In the second article, we proposed *client-side prediction* as a way to overcome these limitations.

The net result of these two articles is a set of concepts and techniques that allow a player to control an in-game character in a way that feels exactly like a single-player game, even when connected to an authoritative server through an internet connection with transmission delays.

In this article, we'll explore the consequences of having other player-controled characters connected to the same server.

Server time step

In the previous article, the behavior of the server we described was pretty simple – it read client inputs, updated the game state, and sent it back to the client. When more than one client is connected, though, the main server loop is somewhat different.

In this scenario, several clients may be sending inputs simultaneously, and at a fast pace (as fast as the player can issue commands, be it pressing arrow keys, moving the mouse or clicking the screen). Updating the game world every time inputs are received from each client and then broadcasting the game state would consume too much CPU and bandwidth.

A better approach is to queue the client inputs as they are received, without any processing. Instead, the game world is updated periodically at low frequency, for

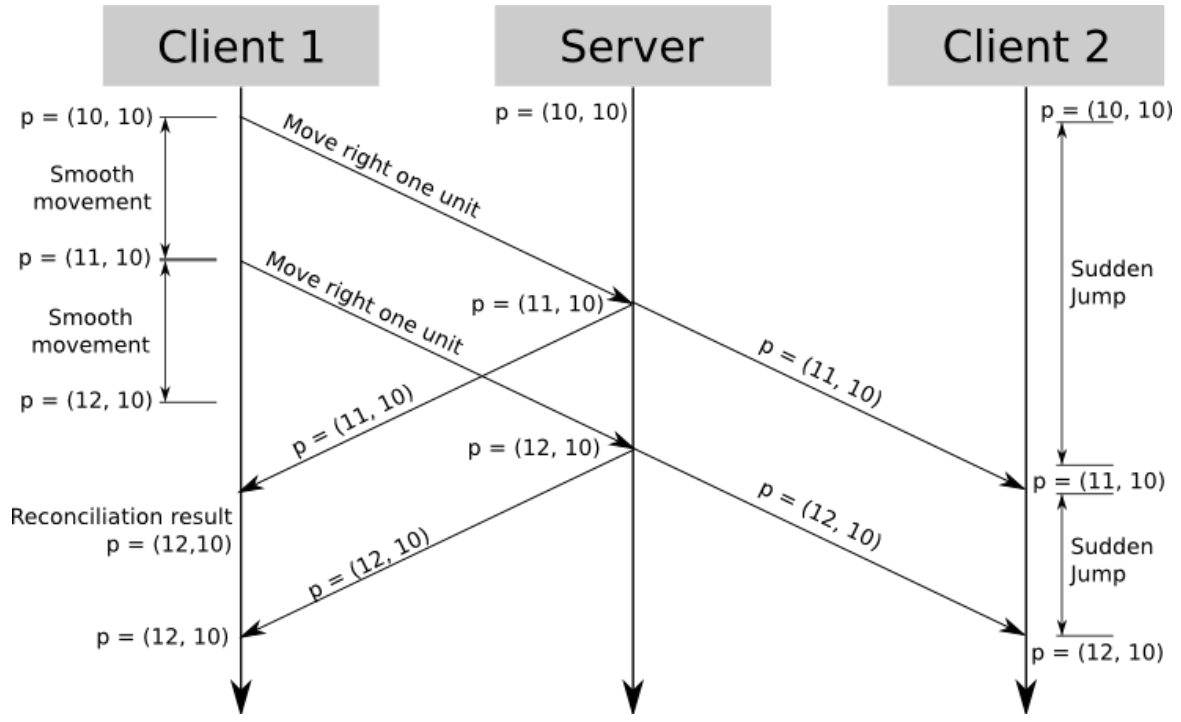
example 10 times per second. The delay between every update, 100ms in this case, is called the*time step*. In every update loop iteration, all the unprocessed client input is applied (possibly in smaller time increments than the time step, to make physics more predictable), and the new game state is broadcast to the clients.

In summary, the game world updates independent of the presence and amount of client input, at a predictable rate.

Dealing with low-frequency updates

From the point of view of a client, this approach works as smoothly as before – client-side prediction works independently of the update delay, so it clearly also works under predictable, if relatively infrequent, state updates. However, since the game state is broadcast at a low frequency (continuing with the example, every 100ms), the client has very sparse information about the other entities that may be moving throughout the world.

A first implementation would update the position of other characters when it receives a state update; this immediately leads to very choppy movement, that is, discrete jumps every 100ms instead of smooth movement.



Client 1 as seen by Client 2.

Depending on the type of game you're developing there are many ways to deal with this; in general, the more predictable your game entities are, the easier it is to get it right.

Dead reckoning

Suppose you're making a car racing game. A car that goes really fast is pretty predictable – for example, if it's running at 100 meters per second, a second later it will be roughly 100 meters ahead of where it started.

Why "roughly"? During that second the car could have accelerated or decelerated a bit, or turned to the right or to the left a bit – the key word here is "a bit". The maneuverability of a car is such that at high speeds its position at any point in time is highly dependent on its previous position, speed and direction, regardless of what the player actually does. In other words, a racing car can't do a 180º turn instantly.

How does this work with a server that sends updates every 100 ms? The client receives authoritative speed and heading for every competing car; for the next 100 ms it won't receive any new information, but it still needs to show them running. The simplest thing to do is to assume the car's heading and acceleration will remain constant during that 100 ms, and run the car physics locally with that parameters. Then, 100 ms later, when the server update arrives, the car's position is corrected.

The correction can be big or relatively small depending on a lot of factors. If the player does keep the car on a straight line and doesn't change the car speed, the predicted position will be exactly like the corrected position. On the other hand, if the player crashes against something, the predicted position will be extremely wrong.

Note that dead reckoning can be applied to low-speed situations – battleships, for example. In fact, the term "dead reckoning" has its origins in marine navigation.

Entity interpolation

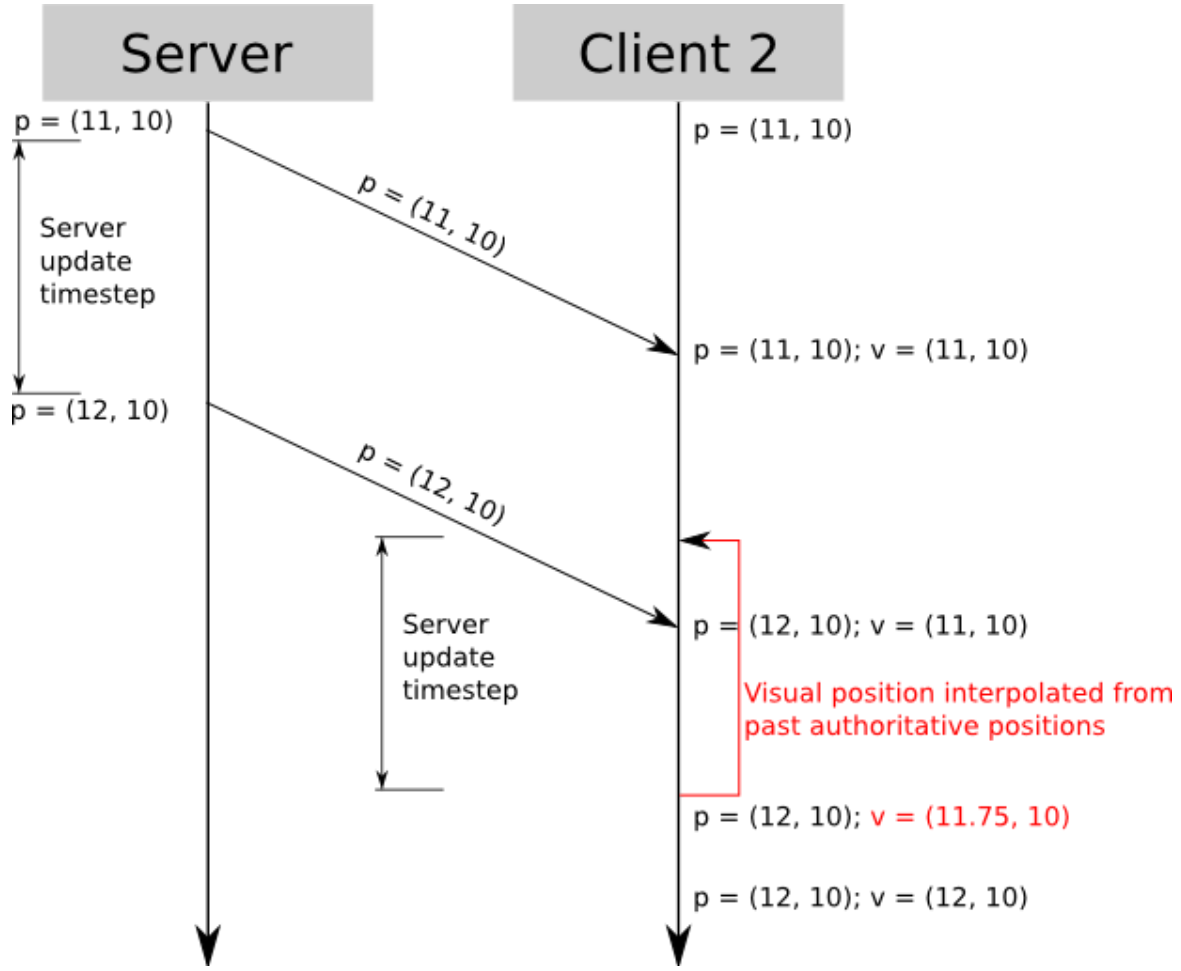
There are some situations where dead reckoning can't be applied at all – in particular, all scenarios where the player's direction and speed can change instantly. For example, in a 3D shooter, players usually run, stop, and turn corners at very high speeds, making dead reckoning essentially useless, as positions and speeds can no longer be predicted from previous data.

You can't just update player positions when the server sends authoritative data; you'd get players who teleport short distances every 100 ms, making the game unplayable.

What you do have is authoritative position data every 100 ms; the trick is how to show the player what happens inbetween. The key to the solution is to show the other players *in the past* relative to the user's player.

Say you receive position data at **t = 1000**. You already had received data at **t = 900**, so you know where the player was at **t = 900** and **t = 1000**. So, from **t = 1000** and **t = 1100**, you show what the other player did from **t = 900** to **t = 1000**. This way

you're always showing the user *actual movement data*, except you're showing it 100 ms "late".



Client 2 renders Client 1 "in the past", interpolating last known positions.

The position data you use to interpolate from **t = 900** to **t = 1000** depends on the game. Interpolation usually works well enough. If it doesn't, you can have the server send more detailed movement data with each update – for example, a sequence of straight segments followed by the player, or positions sampled every 10 ms which look better when interpolated (you don't need to send 10 times more data – since you're sending deltas for small movements, the format on the wire can be heavily optimized for this particular case).

Note that using this technique, every player sees a slightly different rendering of the game world, because each player sees itself *in the present* but sees the other entities *in the past*. Even for a fast paced game, however, seeing other entities with a 100 ms isn't generally noticeable.

There are exceptions – when you need a lot of spatial and temporal accuracy, such as when the player shoots at another player. Since the other players are seen in the past, you're aiming with a 100 ms delay – that is, you're shooting where your target was 100 ms ago! We'll deal with this in the next article.

Summary

In a client-server environment with an authoritative server, infrequent updates and network delay, you must still give players the illusion of continuity and smooth movement. In part 2 of the series we explored a way to show the user controlled player's movement in real time using client-side prediction and server reconciliation; this ensures user input has an immediate effect on the local player, removing a delay that would render the game unplayable.

Other entities are still a problem, however. In this article we explored two ways of dealing with them.

The first one, *dead reckoning*, applies to certain kinds of simulations where entity position can be acceptably estimated from previous entity data such as position, speed and acceleration. This approach fails when these conditions aren't met.

The second one, *entity interpolation*, doesn't predict future positions at all – it uses only real entity data provided by the server, thus showing the other entities slightly delayed in time.

The net effect is that the user's player is seen *in the present* and the other entities are seen *in the past*. This usually creates an incredibly seamless experience.

However, if nothing else is done, the illusion breaks down when an event needs high spatial and temporal accuracy, such as shooting at a moving target: the position where Client 2 renders Client 1 doesn't match the server's nor Client 1's position, so headshots become impossible! Since no game is complete without headshots, we'll deal with this issue in the next article.

<< Part II: Client-Side Prediction and Server Reconciliation · **Part IV: Lag Compensation >>**