<< Series Start

Gabriel Gambetta

Fast-Paced Multiplayer (Part II): Client-Side Prediction and Server Reconciliation

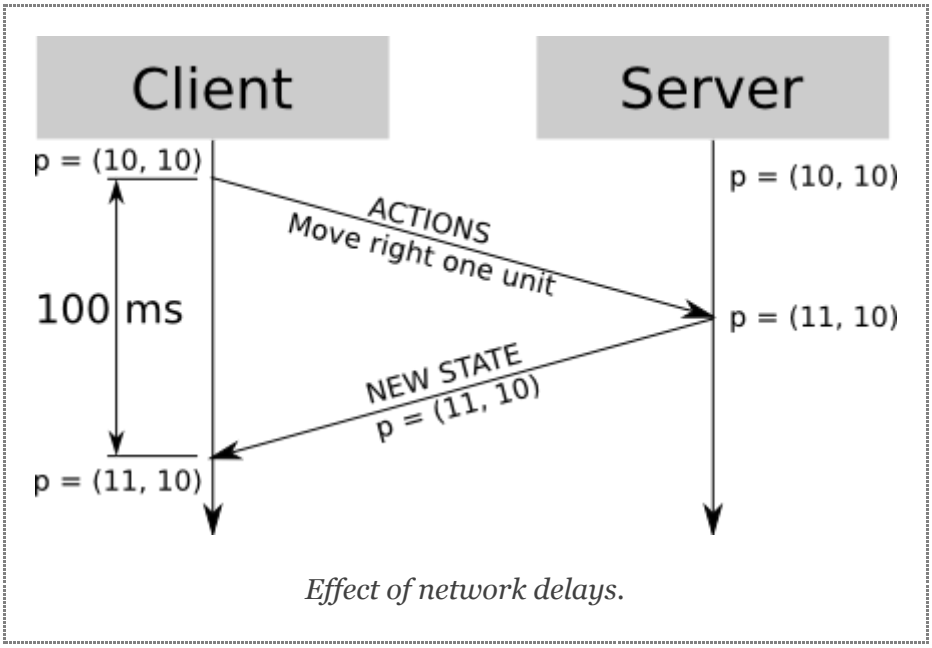
Client-Server Game Architecture | Client-Side Prediction and Server Reconciliation | Entity Interpolation | Lag Compensation | Live Demo

Translations: Korean | Russian

Introduction

In the first article of this series, we explored a client-server model with an authoritative server and dumb clients that just send inputs to the server and then render the updated game state when the server sends it.

A naive implementation of this scheme leads to a delay between user commands and changes on the screen; for example, the player presses the right arrow key, and the character takes half a second before it starts moving. This is because the client input must first travel to the server, the server must process the input and calculate a new game state, and the updated game state must reach the client again.



Effect of network delays.

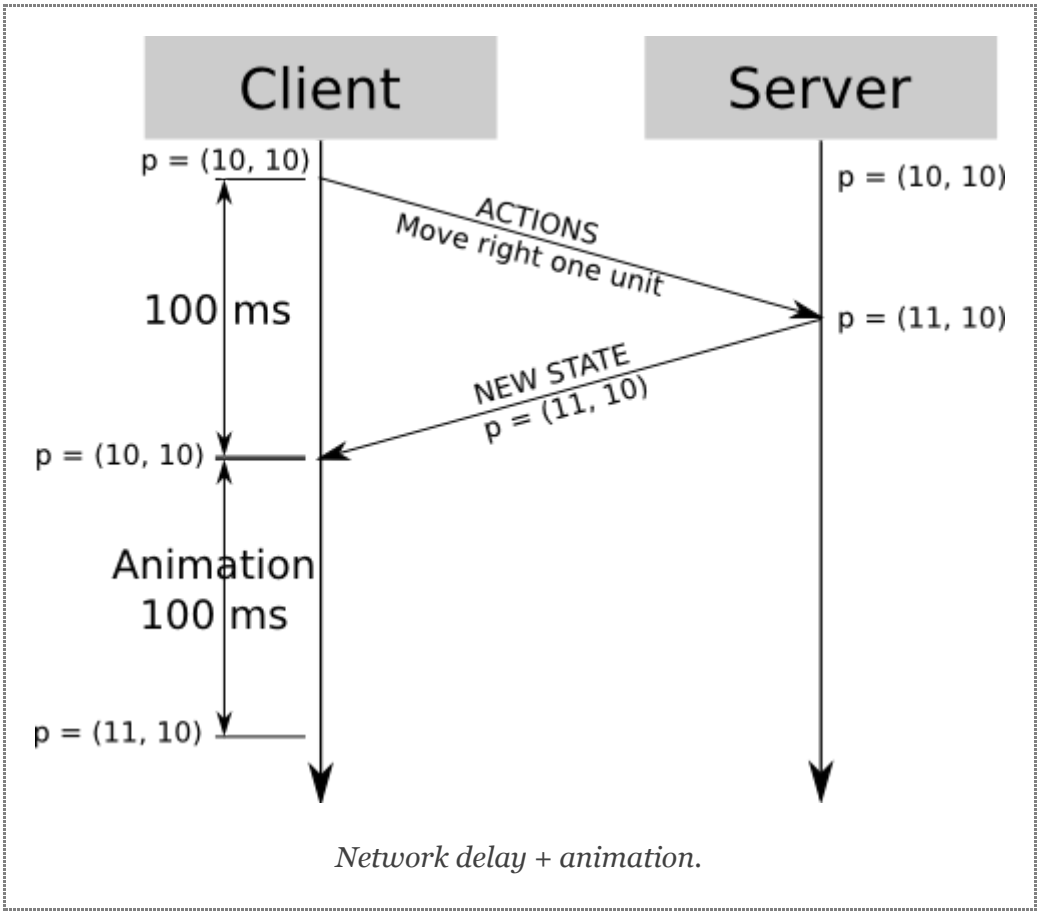
In a networked environment such as the internet, where delays can be in the orders of tenths of a second, a game may feel unresponsive at best, or in the worst case, be rendered unplayable. In this article, we'll find ways to minimize or even eliminate that problem.

Client-side prediction

Even though there are some cheating players, most of the time the game server is processing valid requests (from non-cheating clients and from cheating clients who aren't cheating at that particular time). This means most of the input received will be valid and will update the game state as expected; that is, if your character is at (10, 10) and the right arrow key is pressed, it will end up at (11, 10).

We can use this to our advantage. If the game world is *deterministic* enough (that is, given a game state and a set of inputs, the result is completely predictable),

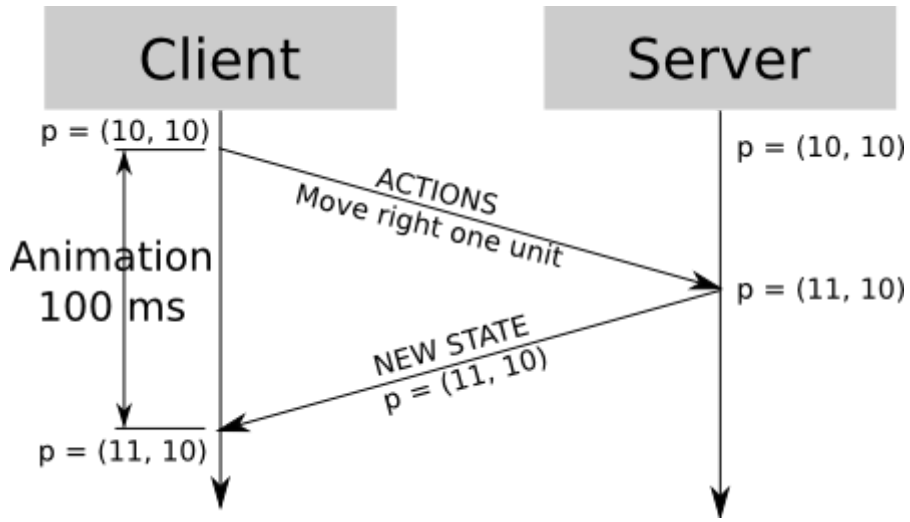
Let's suppose we have a 100 ms lag, and the animation of the character moving from one square to the next takes 100 ms. Using the naive implementation, the whole action would take 200 ms:



Network delay + animation.

Since the world is deterministic, we can assume the inputs we send to the server will be executed successfully. Under this assumption, the client can predict the state of the game world after the inputs are processed, and most of the time this will be correct.

Instead of sending the inputs and waiting for the new game state to start rendering it, we can send the input and start rendering the outcome of that inputs as if they had succeded, while we wait for the server to send the "true" game state – which more often than not, will match the state calculated locally :



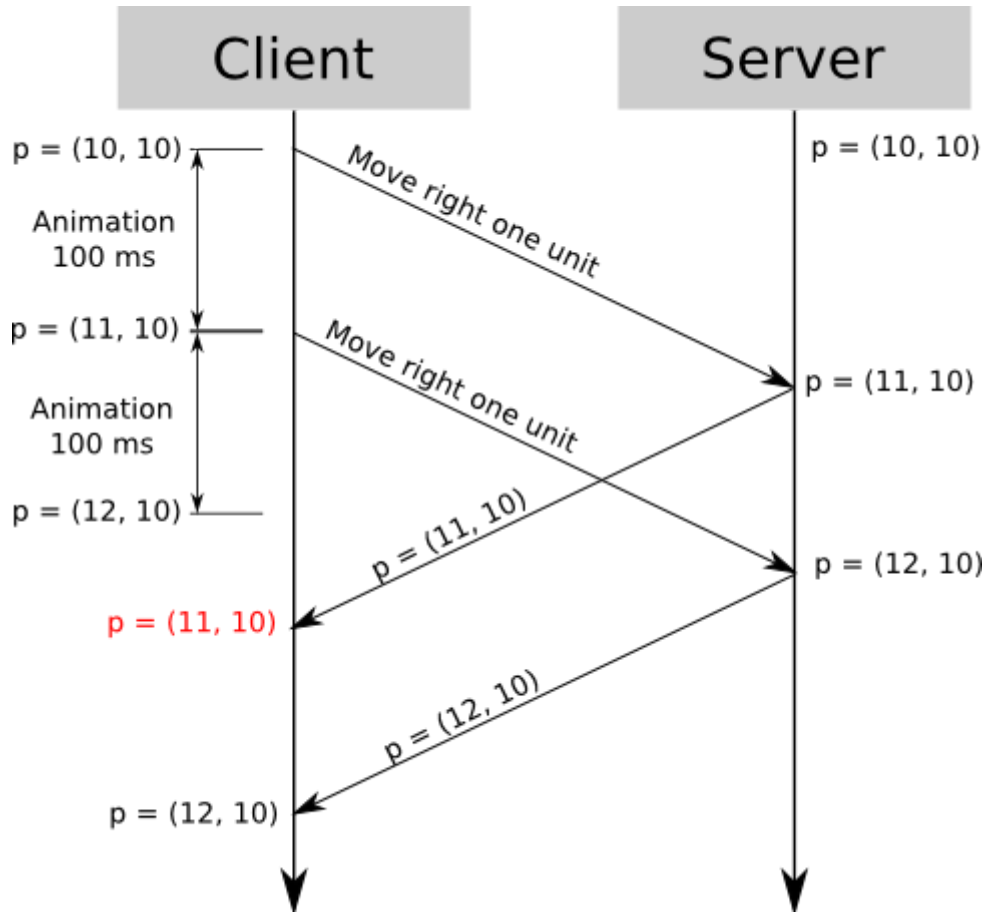
Animation plays while the server confirms the action.

Now there's absolutely no delay between the player's actions and the results on the screen, while the server is still authoritative (if a hacked client would send invalid inputs, it could render whatever it wanted on the screen, but it wouldn't affect the state of the server, which is what the other players see).

Synchronization issues

In the example above, I chose the numbers carefully to make everything work fine. However, consider a slightly modified scenario: let's say we have a 250 ms lag to the server, and moving from a square to the next takes 100 ms. Let's also say the player presses the right key 2 times in a row, trying to move 2 squares to the right.

Using the techniques so far, this is what would happen:



Predicted state and authoritative state mismatch.

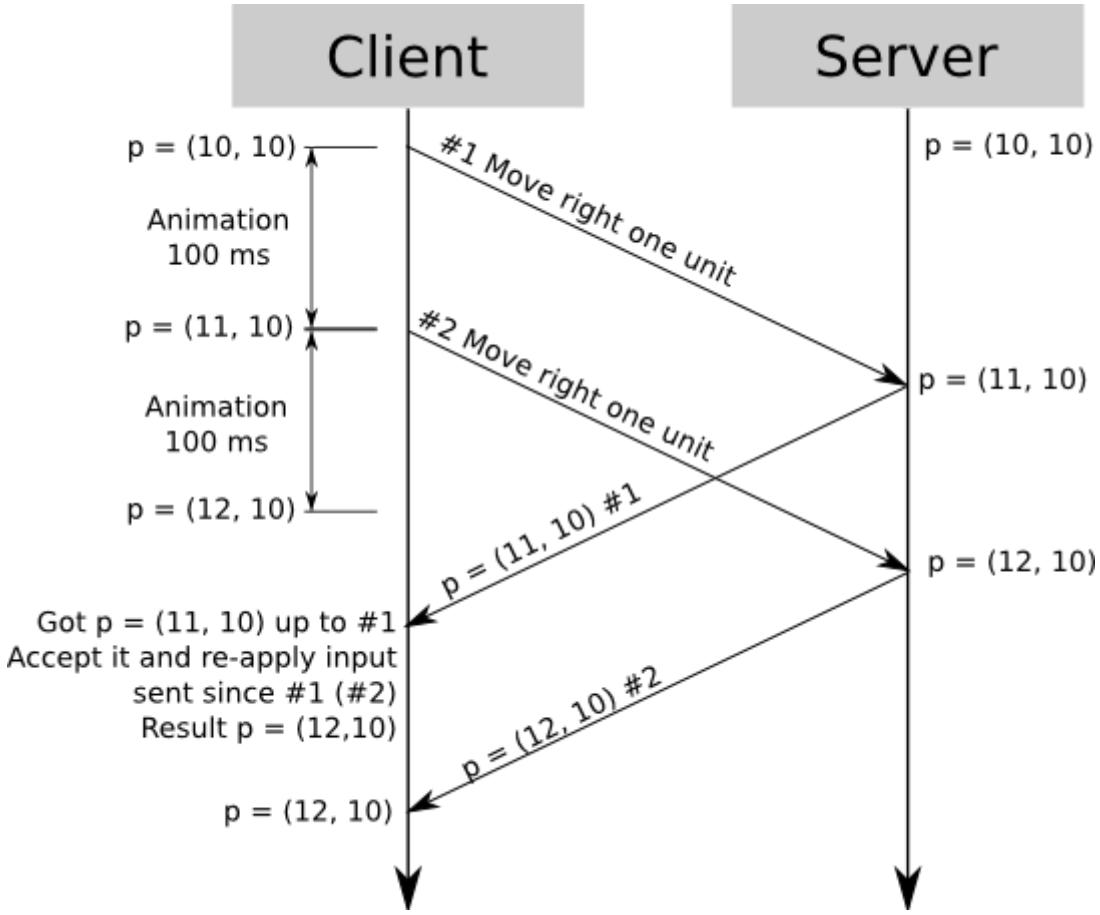
We run into an interesting problem at **t = 250 ms**, when the new game state arrives. The predicted state at the client is **x = 12**, but the server says the new game state is **x = 11**. Because the server is authoritative, the client must move the character back to **x = 11**. But then, a new server state arrives at **t = 350**, which says **x = 12**, so the character jumps again, forward this time.

From the point of view of the player, he pressed the right arrow key twice; the character moved two squares to the right, stood there for 50 ms, jumped one square to the left, stood there for 100 ms, and jumped one square to the right. This, of course, is unacceptable.

Server reconciliation

The key to fix this problem is to realize that the client sees the game world *in present time*, but because of lag, the updates it gets from the server are actually the state of the game *in the past*. By the time the server sent the updated game state, it hadn't processed all the commands sent by the client.

This isn't terribly difficult to work around, though. First, the client adds a sequence number to each request; in our example, the first key press is request #1, and the second key press is request #2. Then, when the server replies, it includes the sequence number of the last input it processed:



Client-side prediction + server reconciliation.

Now, at **t = 250**, the server says "**based on what I've seen up to your request #1, your position is x = 11**". Because the server is authoritative, it sets the character position at **x = 11**. Now let's assume the client keeps a copy of the requests it sends to the server. Based on the new game state, it knows the server has already processed request #1, so it can discard that copy. But it also knows the server still has to send back the result of processing request #2. So applying client-side prediction again, the client can calculate the "present" state of the game based on the last authoritative state sent by the server, plus the inputs the server hasn't processed yet.

So, at **t = 250**, the client gets "**x = 11, last processed request = #1**". It discards its copies of sent input up to #1 – but it retains a copy of #2, which hasn't been acknowledged by the server. It updates its internal game state with what the server sent, **x = 11**, and then applies all the input still not seen by the server – in this case, input #2, "move to the right". The end result is **x = 12**, which is correct.

Continuing with our example, at **t = 350** a new game state arrives from the server; this time it says "**x = 12, last processed request = #2**". At this point, the client discards all input up to #2, and updates the state with **x = 12**. There's no unprocessed input to replay, so processing ends there, with the correct result.

Odds and ends

The example discussed above implies movement, but the same principle can be applied to almost anything else. For example, in a turn-based combat game, when the player attacks another character, you can show blood and a number representing the damage done, but you shouldn't actually update the health of the character until the server says so.

Because of the complexities of game state, which isn't always easily reversible, you may want to avoid killing a character until the server says so, even if its health dropped below zero in the client's game state (what if the other character used a first-aid kit just before receiving your deadly attack, but the server hasn't told you yet?)

This brings us to an interesting point – even if the world is completely deterministic and no clients cheat at all, it's still possible that the state predicted by the client and the state sent by the server don't match after a reconciliation. The scenario is impossible as described above with a single player, but it's easy to run into when several players are connected to the server at once. This will be the topic of the next article.

Summary

When using an authoritative server, you need to give the player the illusion of responsiveness, while you wait for the server to actually process your inputs. To do this, the client simulates the results of the inputs. When the updated server state arrives, the predicted client state is recomputed from the updated state and the inputs the client sent but the server hasn't acknowledged yet.

<< Part I: Client-Server Game Architecture · **Part III: Entity Interpolation >>**