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 – their 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, they're 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.

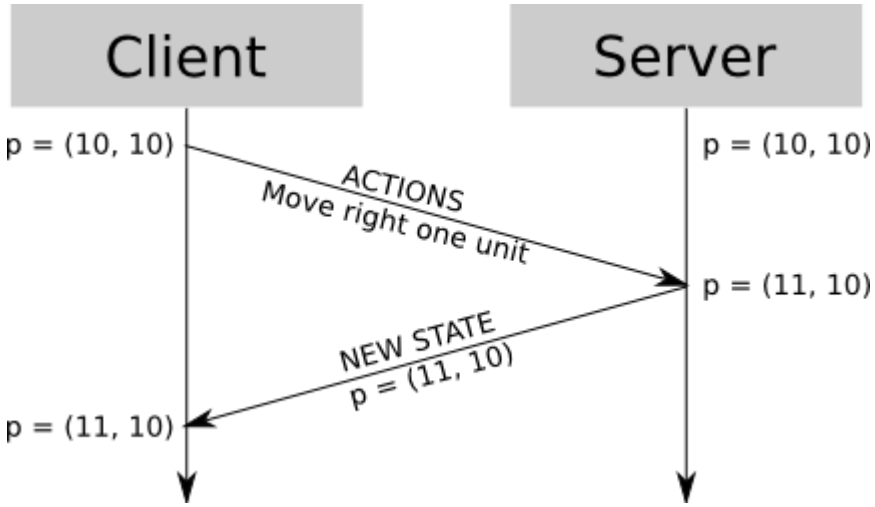
Authoritative servers and dumb clients

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 >>

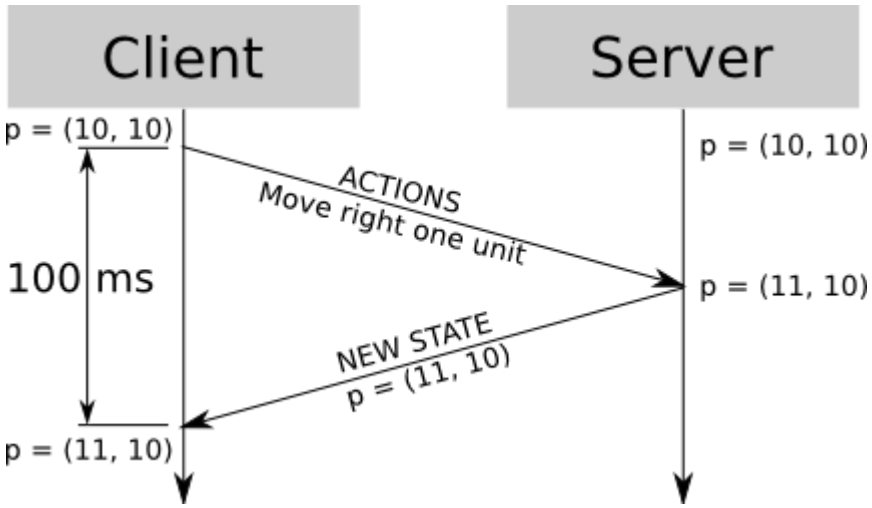
Gabriel Gambetta

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

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.

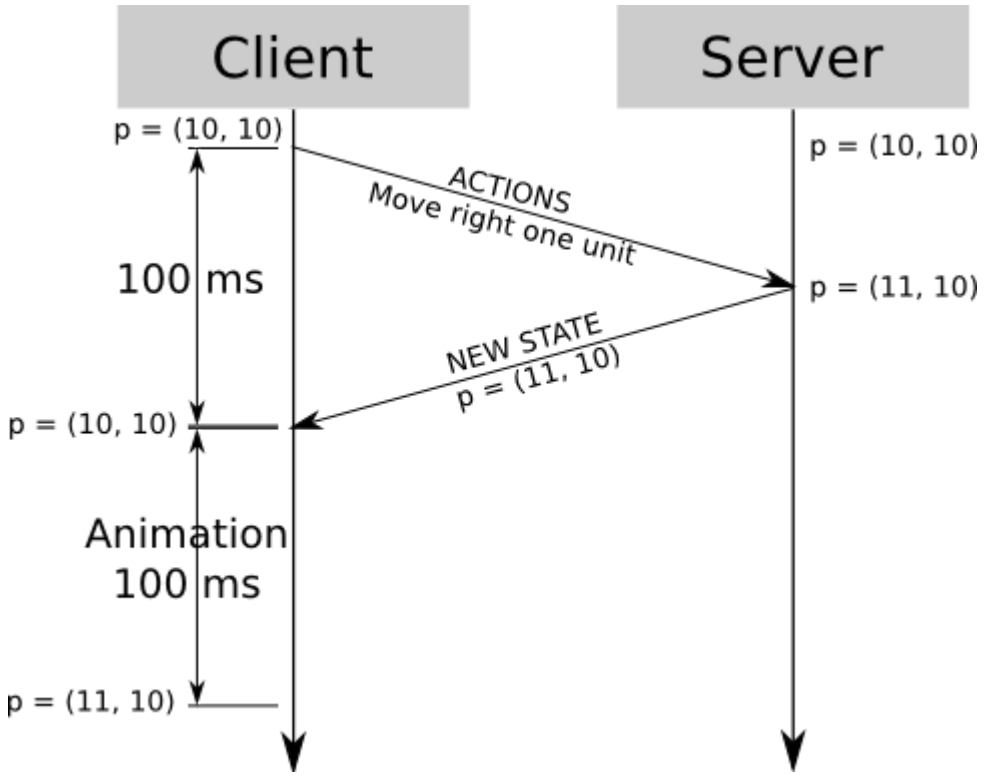
In a networked environment such as the internet, where delays can be ten or hundreds of milliseconds, a game may feel unresponsive at best, or in the worst case, be completely unplayable. In this article, we'll find ways to minimize or even eliminate that problem.

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), we can send the inputs to the server and *immediately* process them on the client - that is, we *predict* what the game state will be after the server has processed the inputs; this eliminates the delay between receiving an input and rendering its effect. Furthermore, most of the time this prediction will be accurate, so there will not be any visible mismatch once the server does send the updated game state.

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:

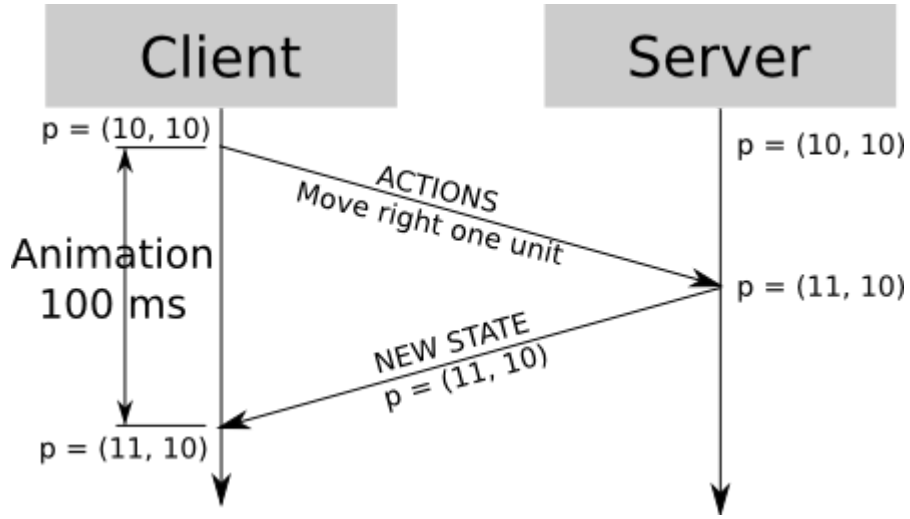


Network delay + animation.

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 :



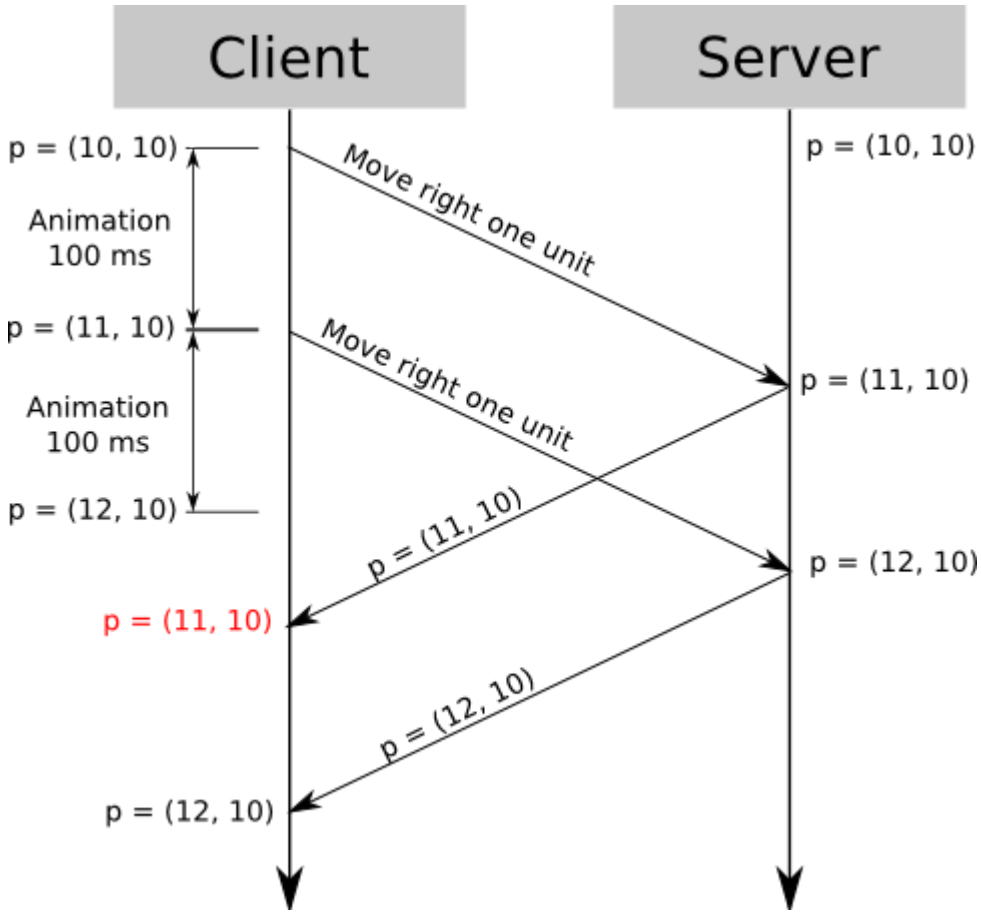
Animation plays while the server confirms the action.

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.

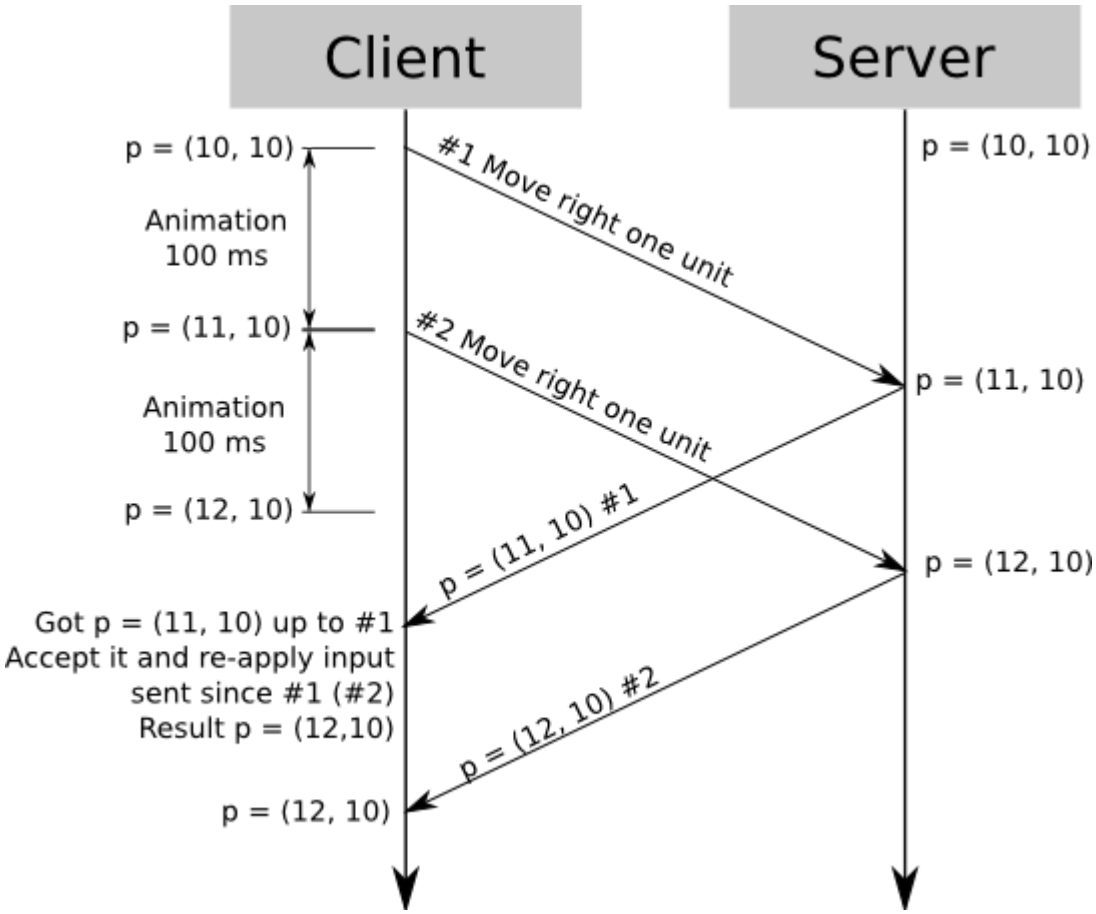
We run into an interesting problem at **t = 250 ms**, when the new game state arrives. The predicted state at the client is **x = 12**, but the server says the new game state is **x = 11**. Because the server is authoritative, the client must move the character back to **x = 11**. But then, a new server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

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.

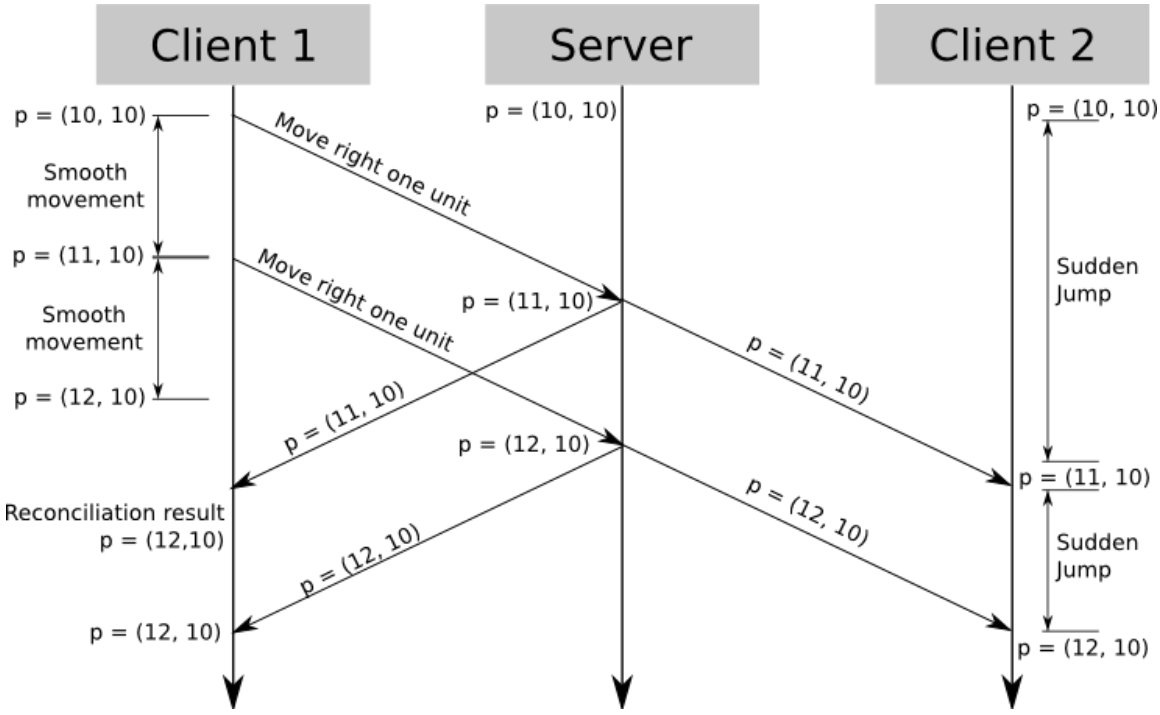
A better approach is to queue the client inputs as they are received, without any processing. Instead, the game world is updated periodically at low frequency, for example 10 times per second. The delay between every update, 100ms in this case, is called the *time step*. In every update loop iteration, 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 those 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.

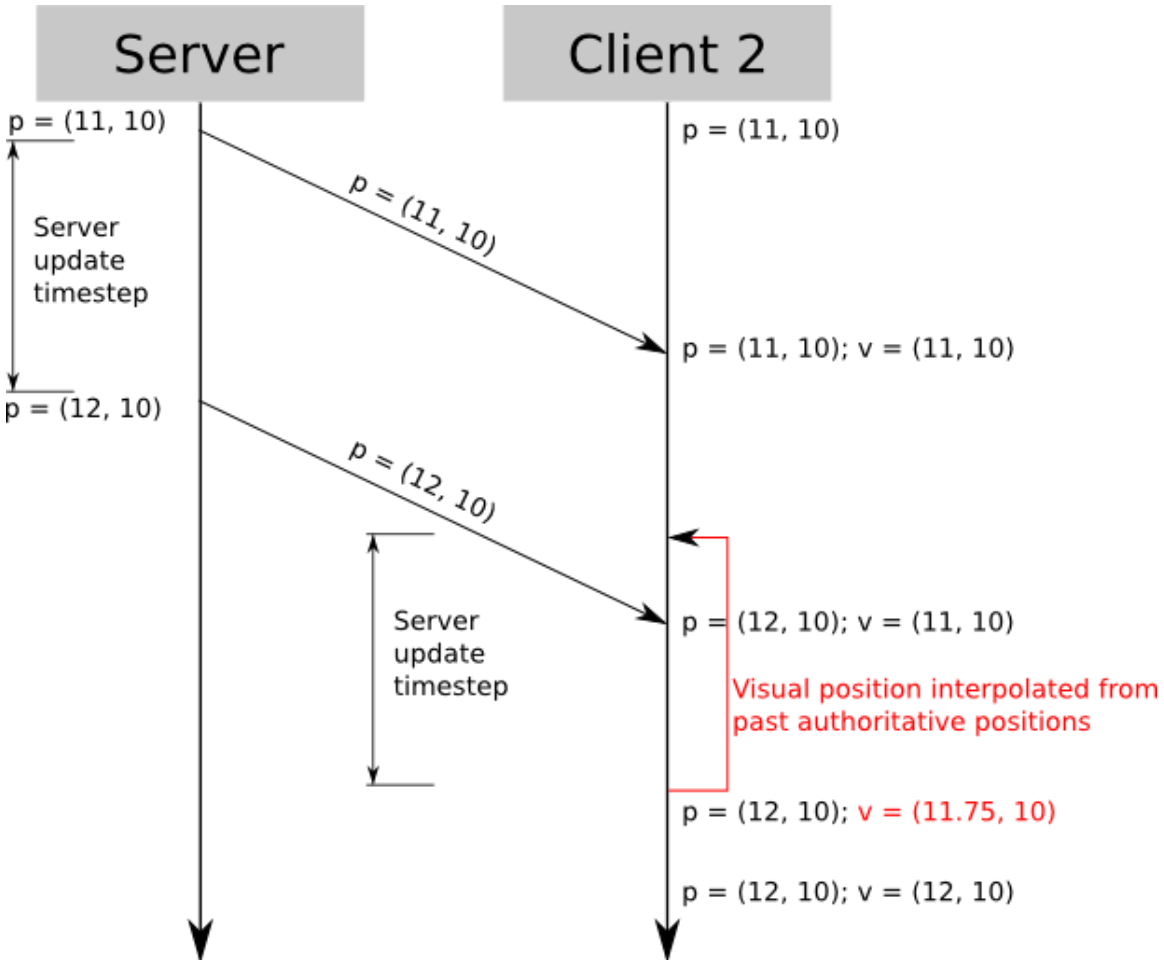
Entity interpolation

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 >>](#)**

Gabriel Gambetta

Fast-Paced Multiplayer (Part IV): Lag Compensation

Introduction

The previous three articles explained a client-server game architecture which can be summarized as follows:

- Server gets inputs from all the clients, with timestamps
- Server processes inputs and updates world status
- Server sends regular world snapshots to all clients
- Client sends input and simulates their effects locally
- Client get world updates and
    - Syncs predicted state to authoritative state
    - Interpolates known past states for other entities

From a player's point of view, this has two important consequences:

- Player sees **himself** in the **present**
- Player sees **other entities** in the **past**

This situation is generally fine, but it's quite problematic for very time- and space-sensitive events; for example, shooting your enemy in the head!

Lag Compensation

So you're aiming perfectly at the target's head with your sniper rifle. You shoot - it's a shot you can't miss.

But you miss.

Why does this happen?

Because of the client-server architecture explained before, you were aiming at where the enemy's head was 100ms *before* you shot - *not* when you shot!

In a way, it's like playing in an universe where the speed of light is really, really slow; you're aiming at the past position of your enemy, but they're long gone by the time you squeeze the trigger.

Fortunately, there's a relatively simple solution for this, which is also pleasant for *most* players *most* of the time (with the one exception discussed below).

Here's how it works:

- When you shoot, client sends this event to the server with full information: the exact timestamp of your shot, and the exact aim of the weapon.
- **Here's the crucial step**. Since the server gets all the input with timestamps, it can authoritatively reconstruct the world at any instant in the past. In particular, it can reconstruct the world exactly as it looked like to any client at any point in time.
- This means the server can know exactly what was on your weapon's sights the instant you shot. It was the *past* position of your enemy's head, but the server knows it was the position of their head in *your* present.
- The server processes the shot *at that point in time*, and updates the clients.

And everyone is happy!

The server is happy because it's the server. It's always happy.

You're happy because you were aiming at your enemy's head, shot, and got a rewarding headshot!

The enemy may be the only one not entirely happy. If they were standing still when he got shot, it's their fault, right? If they were moving… wow, you're a really awesome sniper.

But what if they were in an open position, got behind a wall, and *then* got shot, a fraction of a second later, when they thought they were safe?

Well, that can happen. That's the tradeoff you make. Because you shoot at him in the past, they may still be shot up to a few milliseconds after they took cover.

It is somewhat unfair, but it's the most agreeable solution for everyone involved. It would be much worse to miss an unmissable shot!

Conclusion

This ends my series on Fast-paced Multiplayer. This kind of thing is clearly tricky to get right, but with a clear conceptual understanding about what's going on, it's not exceedingly difficult.

Although the audience of these articles were game developers, it found another group of interested readers: gamers! From a gamer point of view it's also interesting to understand why some things happen the way they happen.

Further Reading

As clever as these techniques are, I can't claim any credit for them; these articles are just an easy to understand guide to some concepts I've learned from other sources, including articles and source code, and some experimentation.

The most relevant articles about this topic are What Every Programmer Needs to Know About Game Networking and Latency Compensating Methods in Client/Server In-game Protocol Design and Optimization.

[<< Part III: Entity Interpolation](#) · **Live Demo >>**