Multiplayer Networking for Games

This series of lectures/notes talk about how to design and implement multiplayer networking for interactive (non turn-based) style games. Interactive games with fast-paced player control, like shooters. Interactive, or turn-based, games like card playing do not need (most of) these techniques.

These notes based on this web site: http://www.gabrielgambetta.com/client-server-game-architecture.html

Introduction

There are two major issues to deal with in networked multiplayer games:

1. Network lag (and possibly bandwidth)
2. Trust (i.e., Cheating)

The fundamental issue with a network is that it takes some amount of time to transmit data from one computer to another. Furthermore, the time it takes to send a packet of data from Player A to Player B is different from the time it takes to send a packet of data from Player C to Player B. It is difficult to impossible to have each client in a (non turn-based) networked game stay in sync and give each player the same view of the game.

The fundamental issue with trust is that you can not trust the client for very much, due to the potential for cheating (really, network lag plays into this anyway, because of timing issues). A client can't be trusted to report its position, resources (ammo, health, etc), whether it collected a game resource, destroyed a game resource, and so on. A client can't be trusted for anything other than the player inputs. Even those have to be validated to make sure they don't violate the rules of the game. For example, the rate at which a weapon can be fired versus how fast inputs come from the client. If "fire" inputs come from the client faster than the rate the weapon can be fired, the server has to reject inputs that are faster than the allowable rate.

These two issues result in the need to create a client-server model for networked multiplayer games.

Server

- Runs THE authoritative model/simulation of the game.
- Collects and processes inputs from connected clients.
- Sends game state updates to connected clients.

Client

- Maintains a client perspective model of the game, based on local player inputs and model updates from the server.
- Collects and sends inputs to the server.

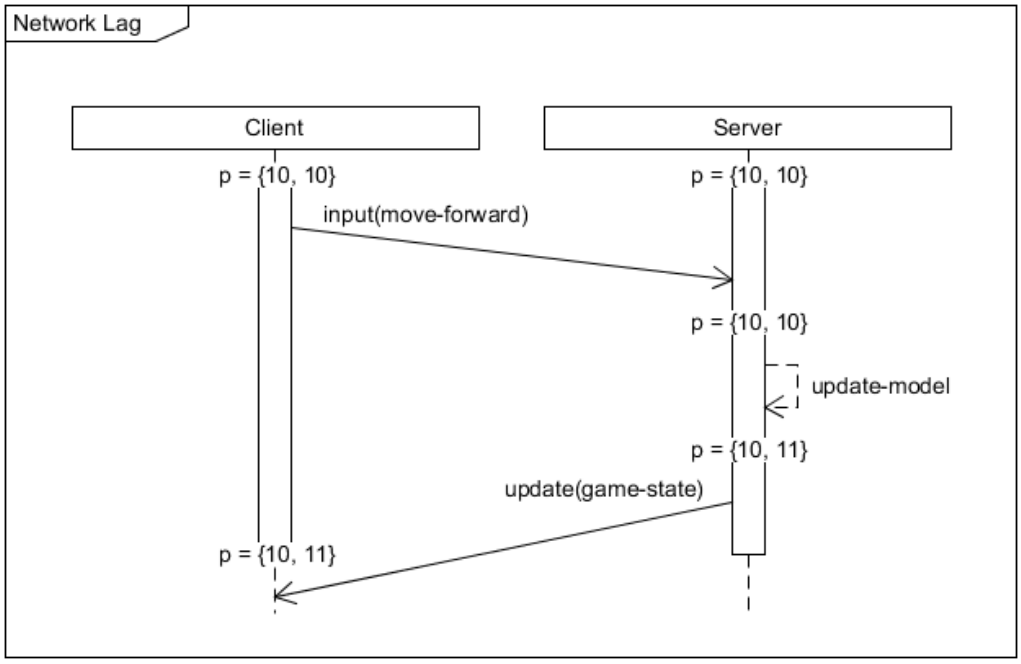
There are two models of the game. The server runs the authoritative model of the game, game loop, full simulation, and all. The client runs a lightweight model of the game with the state of the game provided by updates from the server, with some predictive simulation taking place.

Network Lag, plus other delays

The speed of light is an actual limitation in playing games. It takes time for the network packets to propagate through the various physical and wireless connections, routers, bridges, network stacks, resulting in meaningful effects on interactivity. For example, I just did a ping to www.mit.edu and see an average of 25 ms; round trip time to send and receive a packet. That's not too bad, could send/receive 40 updates per second.

Let's add in the time to simulate the game (update) and add that in, both the server and the client updates. Let's say the server is capable of 30 updates per second, and the client the same.

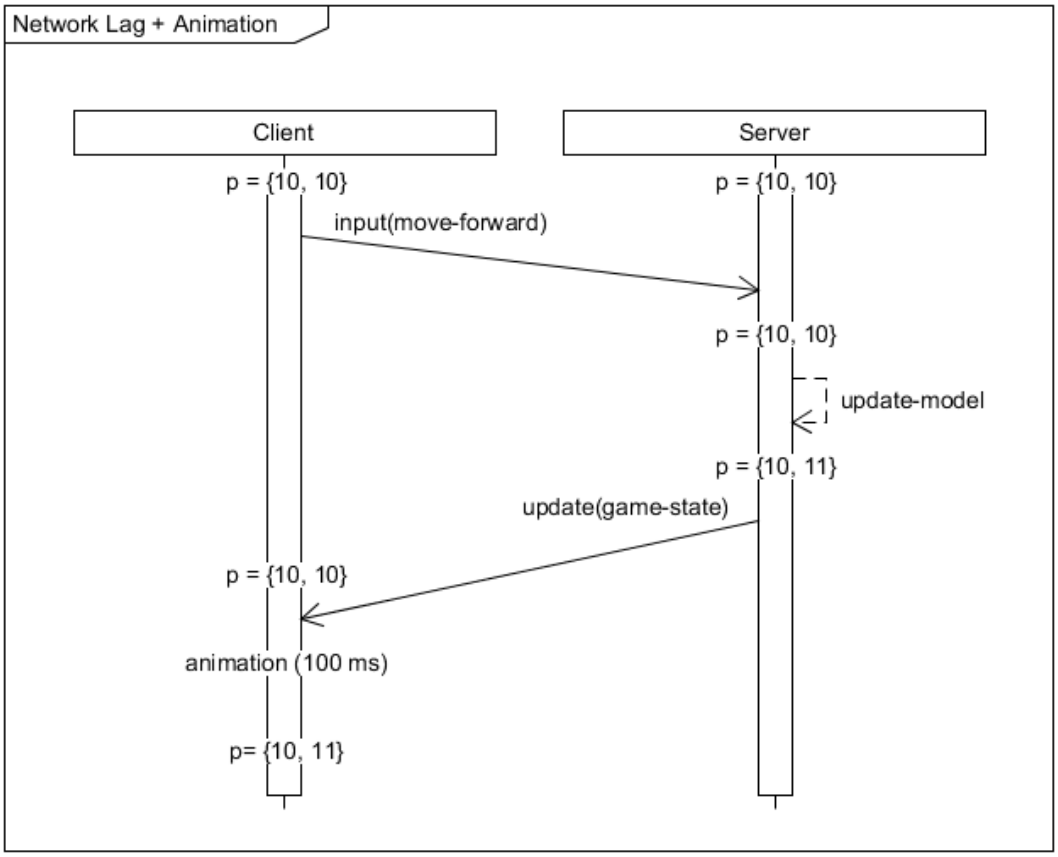
- 12.5 ms to send an input to the server
- 33.33 ms to update game at server; assuming my packet arrived exactly at the start of the game model update
- 12.5 ms to send an updated game state to the client
- 33.33 ms to update & render the game at the client
- Total time from input to seeing a change on the screen: 91.66 ms; 11 times per second, um yea.



Worst case scenario

- Use input right after the start of client update/render. Wait to send input until looping back around: 33.33 ms
  - Argues for sending update as soon as it happens, rather than collecting all and sending in a single update.
- 12.5 ms to send input to the server
- Input packet showed up just after start of server updating game model: 33.33 ms
- 33.33 ms to update game at server
- 12.5 ms to send an updated game state to the client
- Update showed up just as client started its own local update and render: 33.33 ms
- 33.33 ms to update & render the game at the client
- Total time from input to seeing a change on the screen: 191.65 ms; 5 times per second, yuck!

Actually, it is worse than this, consider the next sequence diagram…

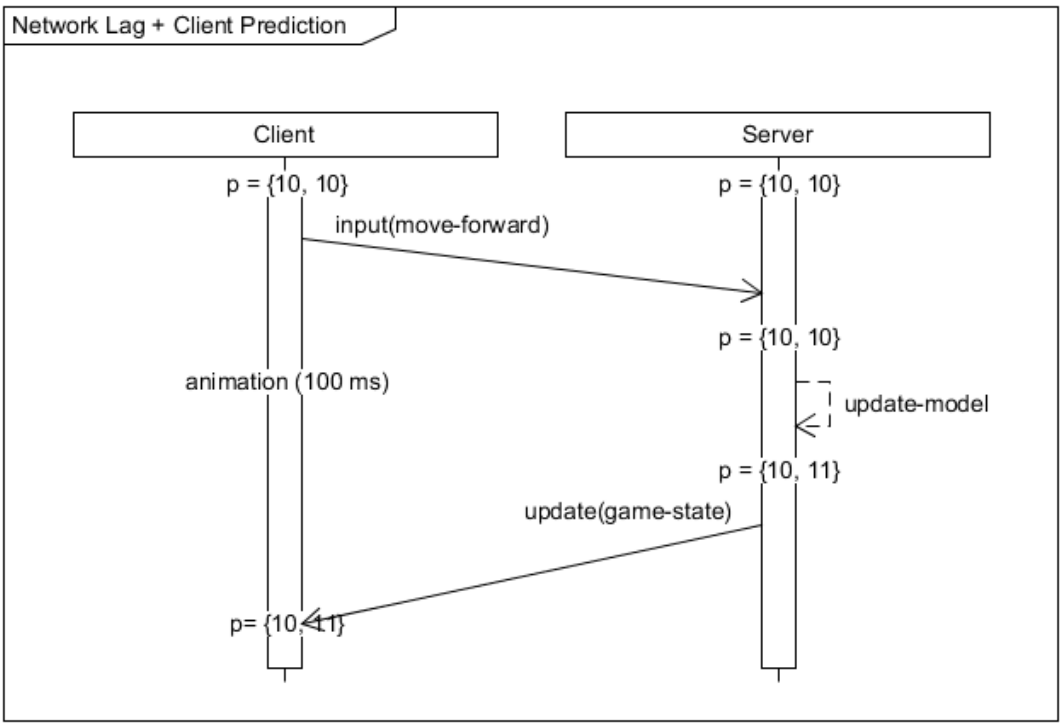


Because of the character animation, we are now at 292.65 ms before the human sees the game character in the next spot, where the expectation is that it should have only taken 100 ms (the animation time).

Client-Side Prediction

A client can trust itself, even if the server doesn't. Therefore, the client can go ahead and take action based on the user's input and update the game model based on that. As updated states are received from the server, the game model is reconciled between the two.

Client prediction looks like the following…



We let the client go ahead and start the animation based on the input, even before it has received an updated state from the server; with the expectation the server update will match the client prediction of state. From the player's perspective, significant improvement.

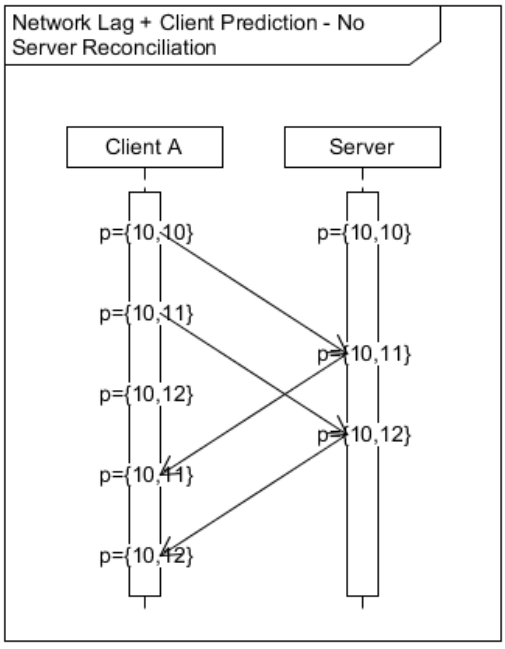
Everyone is happy, right?

Wait, what about this…ugh!

- Notice the server update caused the client prediction to be overridden, taking the player about to 10, 11, then a later update puts the player at 10, 12.

The problem is the client is seeing the game model in the present, whereas the server updates are from the past.

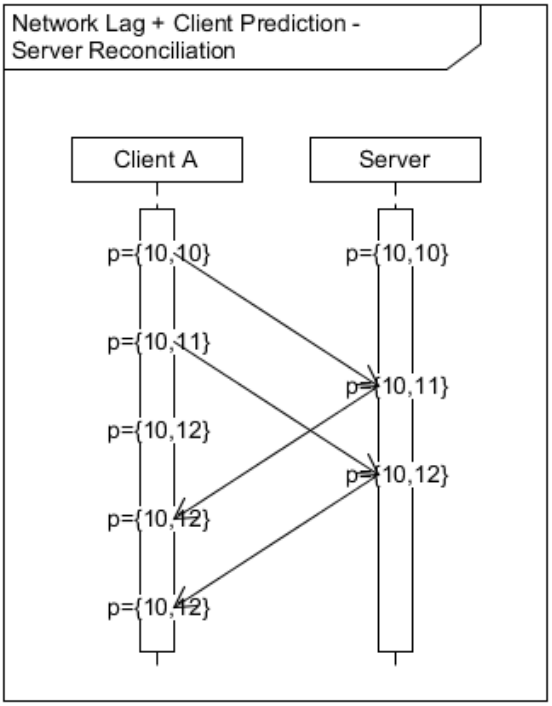
What do we do...server reconciliation to the rescue!



Server Reconciliation

The client and server have to cooperate (only a little) to allow the client prediction to not be adversely impacted by server updates. Here is how to do it…

- Client adds a sequence number to each message sent to the server.
- When the server sends an update to a client, it includes that last message sequence number it processed.
- When the client receives an update from the server, it updates the game model based on that.
- THEN, and this is the key, the client reapplies all local inputs the server hasn't yet acknowledged.
    - In other words, the game state is updated to represent the historical state known at the server, then the client performs its prediction by updating the model to the present by reapplying any inputs not yet processed by the server.

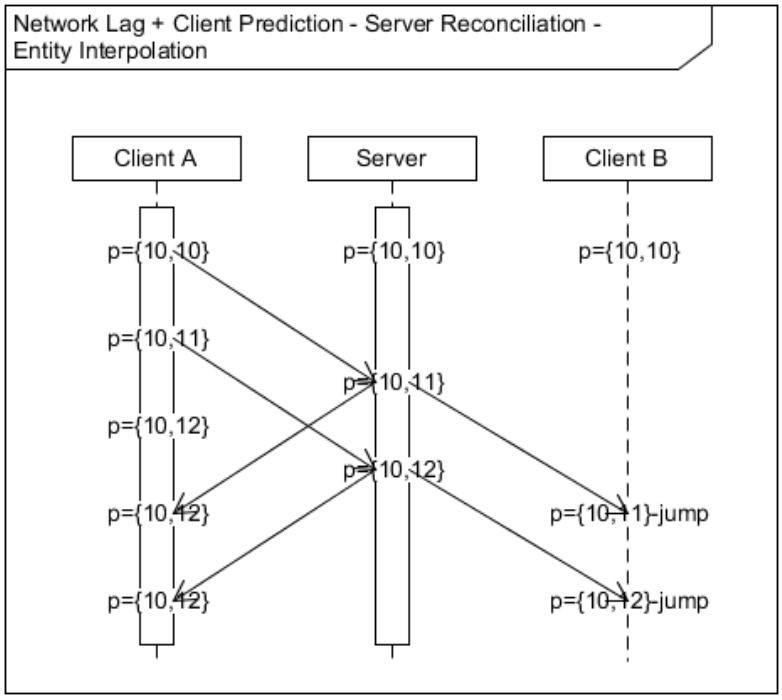


Additional Notes on Client Prediction and Server Reconciliation

- Client might predict another player's death, but may want to wait until the server actually says so, just in case there is a mis-prediction (lost packets, for example). Other player may have applied a health pack that you don't yet know about, resulting in an incorrect prediction.
- Server doesn't tell client when to do things like particles or explosions, those are automatically decided on at the client.

Entity Interpolation

To this point, we have tackled the issue of trust by moving to an authoritative server game model with updates, then dealt with network lag through client prediction and server reconciliation. What about other players and their updates? Won't they still stutter as the server sends discrete updates about them? Well, yes, they will.



Consider the sequence diagram…

- T_0 : Client A sends input to Server
- T_1 : Client A finishes client prediction for self; Client A sends input to Server
- T_2 : Server updates model, sends updated state to Client A & Client B
- T_3 : Client A finishes client prediction for self
- T_4 : Server updates model, sends updated state to Client A & Client B
- T_5 : Client A reconciles self; Client B moves Client A (sudden jump)
- T_6 : Client A reconciles self; Client B moves Client A (sudden jump)

We'd like to eliminate the sudden jumps at Client B for Client A's character. You won't like this, but this is what to do…

- A T_5 Client B begins animation (interpolating) the position of Client A. It knows how long to perform the interpolation because the server passes along a length of time since the last time Client A was updated, an "update window".

Yes, from the point of view of each player, all the other players are in the past. This is why you can't have super slow network lag and game updates, it still needs to be updated fairly frequently, like 10 times per second (at least).

** This is "a" way of doing entity interpolation, not the only way. Client might predict where the entity is going along a curve/path and then reconcile, over time, the authoritative position the server sends.

Lag Compensation

Ugh, too much for what we are going to do.

I definitely don't agree with the web site author's approach to lag compensation, on several accounts.

- You can not trust a timestamp sent from a client!!! Don't do it! It isn't just about trust, it is also that clocks in a distributed system aren't guaranteed to be synced, especially as they are not in a controlled environment.
- You can not trust a client to tell you anything about the state of the player (like the aim direction). You can ONLY accept inputs, validate the inputs, and let the server specify the state.

The appropriate clock for a distributed system in a Lamport Clock, or Vector Clock, or similar approach.

The server has to remember state for X updates, then when an event occurs (such as testing for a weapon hit), the logical clock has to be inspected to find the historical state the client of interest was in at the time the local event occurred and use that state to decide what happened.