CS 5410

Intro to Multiplayer

This presentation sponsored by...

Relevant Note

- The subject of multiplayer and networking in games could take up a full semester course
- We are doing it in a single presentation!

Terminology

- **Server**
  - Runs **THE** authoritative model/simulation of the game
  - Collects and process 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
  - Renders a local client view of the game state

Terminology – Continued

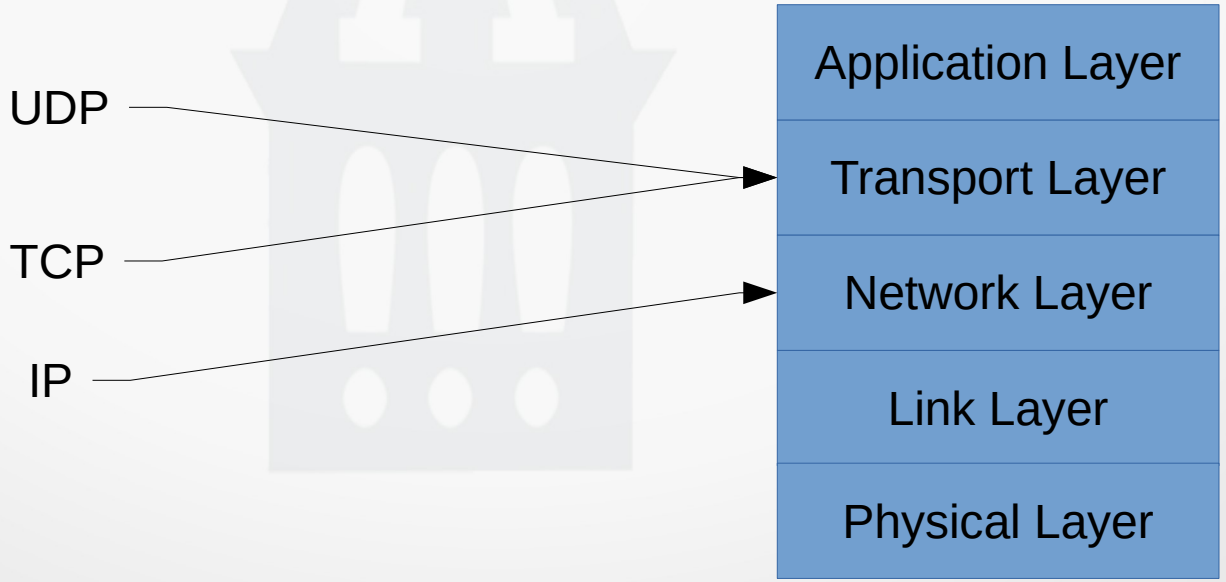
- **Bandwidth** : Rate of data transfer over a given connection. Often measured in bits-per-second (bps) or bytes-per-second (Bps).
- **Latency** : Time it takes to send and receive a packet between two networked computers
- **(latency) Jitter** : Variation in latency
- **Connectionless** : Sending of messages without a prior arrangement/channel; may result in loss of data. e.g., UDP
- **Connection-oriented** : Sending of messages over an established channel; usually a no data loss protocol. e.g., TCP

Networking Terminology – Transport

- **UDP** (datagram)
  - Unreliable, no guarantee of delivery, no guarantee of order of delivery
  - Packet oriented, limited (max) size per packet
  - Low overhead; around 28 bytes per packet
  - Fastest possible delivery mechanism
- **TCP/IP**
  - Reliable, guaranteed delivery; if connection stays alive
  - Stream oriented, no size limit on data
  - Connection oriented, state is preserved, requires some overhead
  - Relative to UDP, much slower
  - Implemented as a protocol in the “stack” that guarantees packet delivery and order

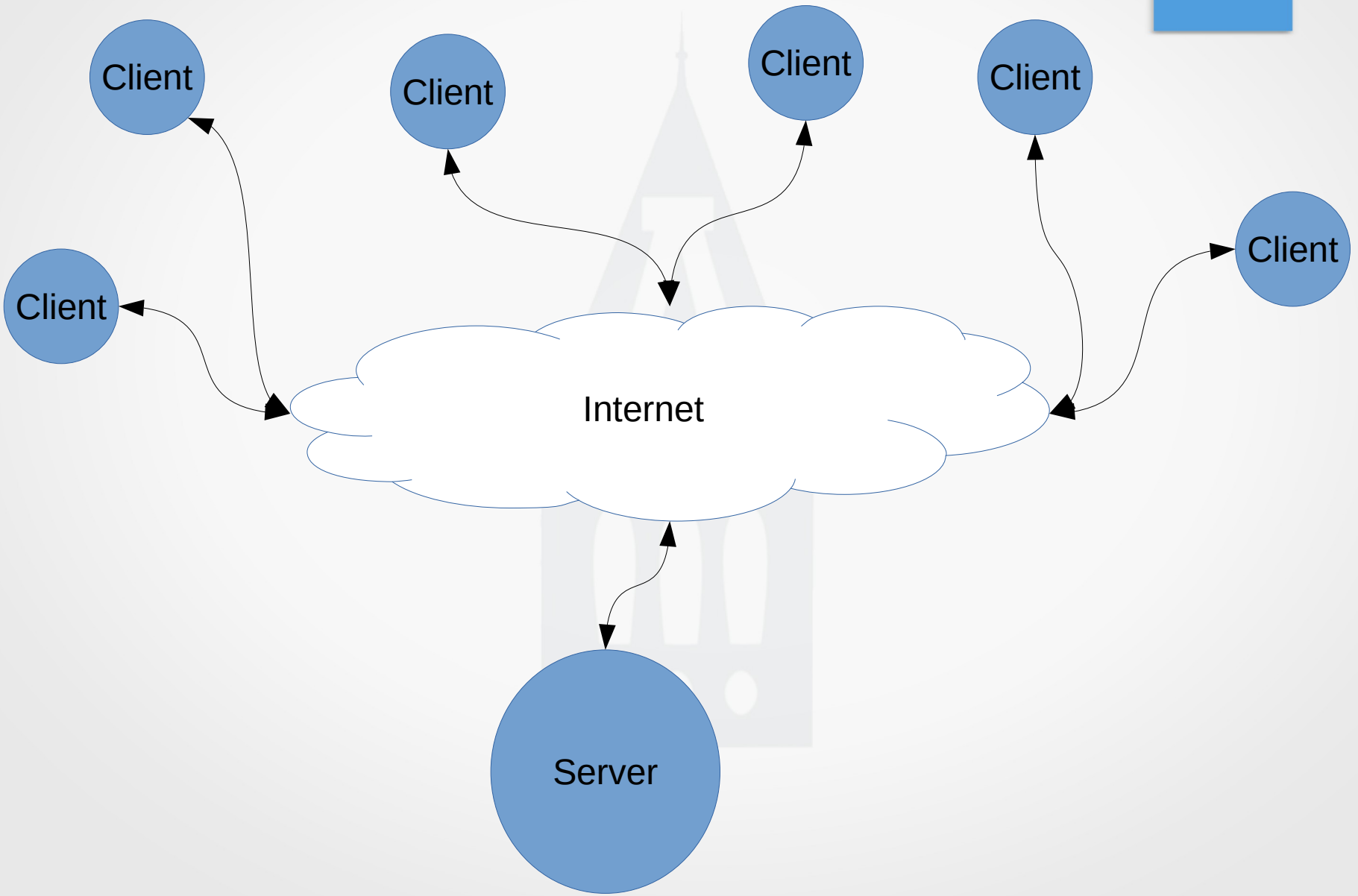
Networking Terminology – Packet Types

- **IP** : Header and data section
- **UDP** : Header and data section; packet fits inside an IP packet (usually)
- **TCP/IP** : Header and data section; fits inside an IP packet)



* Not the full 7 layer OSI model

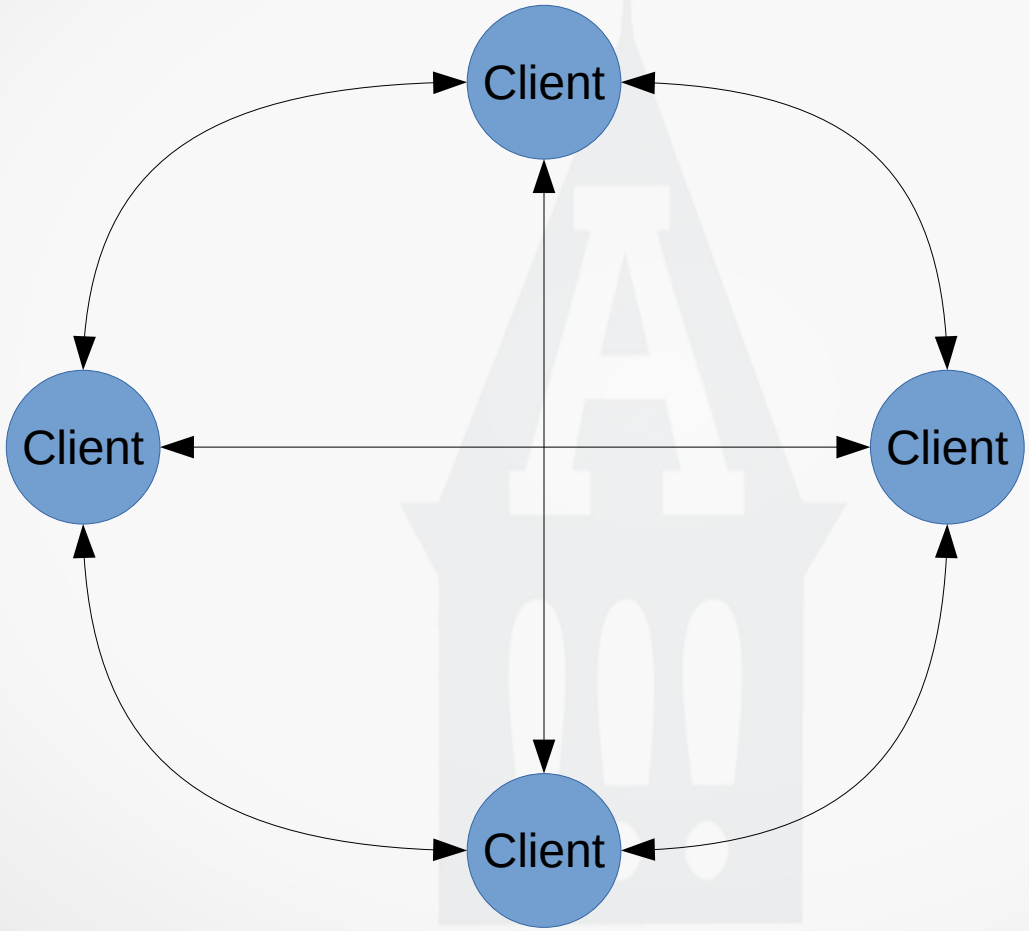
Networking Models : Client-Server



Client – Server : Topics

- Server must be able to handle computational requirements.
  - Is it a turn-based game, slow, twitch?
- Server must have networking bandwidth data rates
  - incoming
  - outgoing
- Game State
  - (authoritative) Server updates based on inputs from clients
  - Clients receive updated game state from server

Networking Models : Peer-to-Peer



Peer-to-Peer : Topics

- No use of authoritative server
  - But might use a server for lobby and matchmaking
- Potentially reduces network latency and bandwidth
  - No server in the middle, half the latency
  - Only inputs sent to each peer vs inputs & game state
- Synchronization among peers is more complex. May have to work in lockstep.
  - Each peer waits to simulate game state until inputs from all other peers are received
  - If random numbers used, have to share the seed
- Some games that seem peer-to-peer are actually client-server
  - One of the peers is chosen to run the server

Data Protocol Considerations

- **Data Rate**
  - Turn-Based, RTS, FPS are all different
- **Data Volume**
  - Tic-tac-toe has different data rate than Call of Duty
- **Data Delivery Guarantee**
  - Some games might survive data loss, others might not
- **Backwards Compatibility**
  - Minimum: identify the protocol version
  - Might need to support multiple versions simultaneously...maybe for an MMO

Data Protocol Considerations - Continued

- **Independent of…**
  - Client frame-rate
  - Server simulation rate
- **Object Serialization/Replication**
- **Data Reduction**
  - On-the-fly compression: trade CPU for bandwidth
  - Numeric ranges: value [0, 180] only needs a byte
  - floats instead of doubles, maybe 16 bit float
  - (delta compress) Only send changes in state/values, rather than entire new state.

Two Major Issues

- **Lag (latency, input) Sources**
  - Speed of light is a constraint
    - e.g., (time) P_1 to P_2 is different than P_2 to P_3
  - So is the speed of the game loop (input)
- **Trust – Authority, Cheating**
  - Need a source for the authoritative model of the game, regardless of cheating
  - Can’t trust to report position, inventory, health, etc.
  - Trust client only for inputs
    - Even inputs need to be validated for reasonableness

Game Models

- Two models of the game
  - Authoritative **server** model – full simulation
  - Lightweight **client** model – local state

Both models run a game loop, but are informed from different inputs/sources.

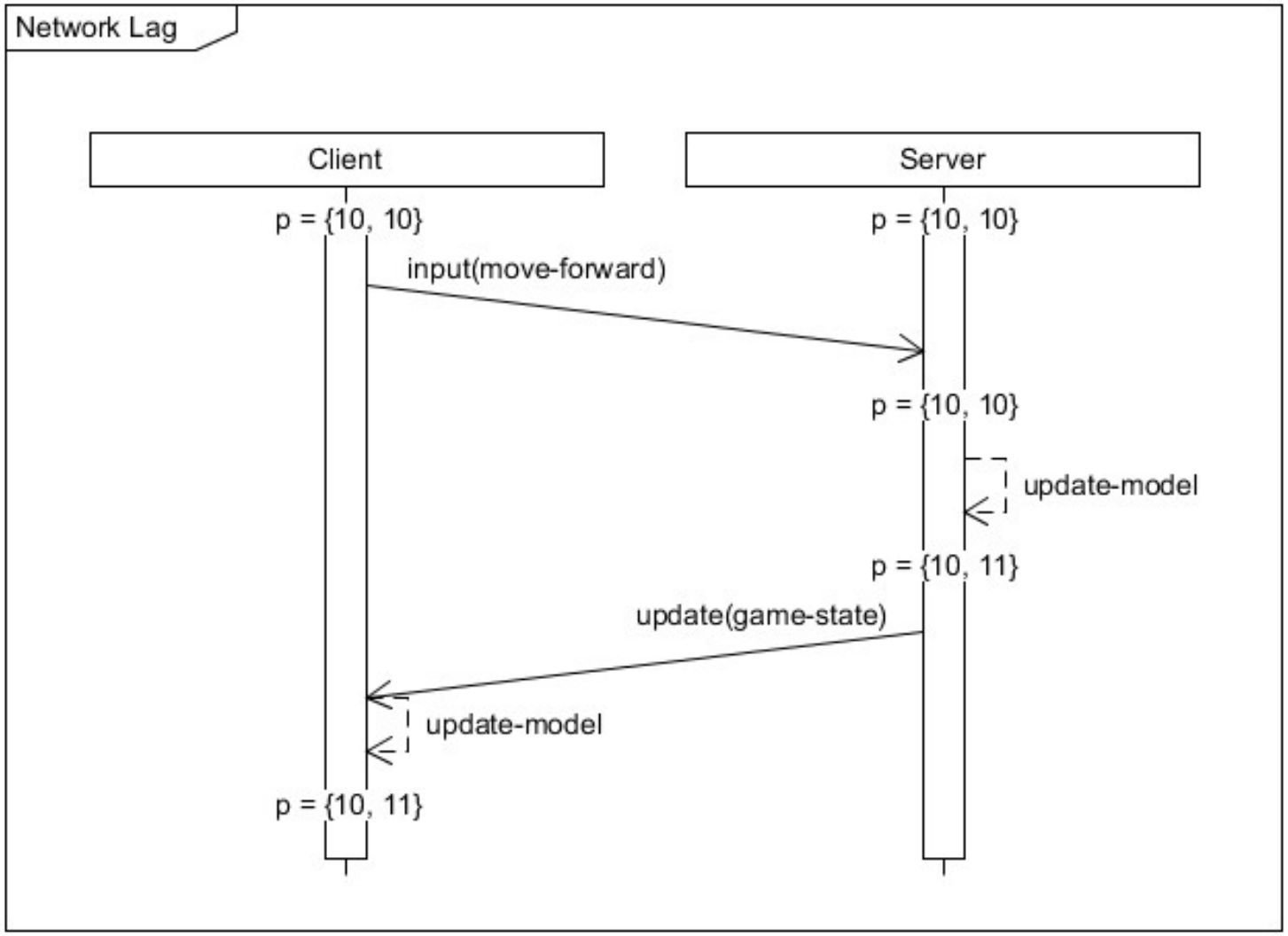
Lag

Lag

- Ping to www.mit.edu is about 25ms (round trip)
- Client game model runs at 30 fps (33.33 ms per frame)
- Server game model runs at 30 Hz (33.33 ms per loop)

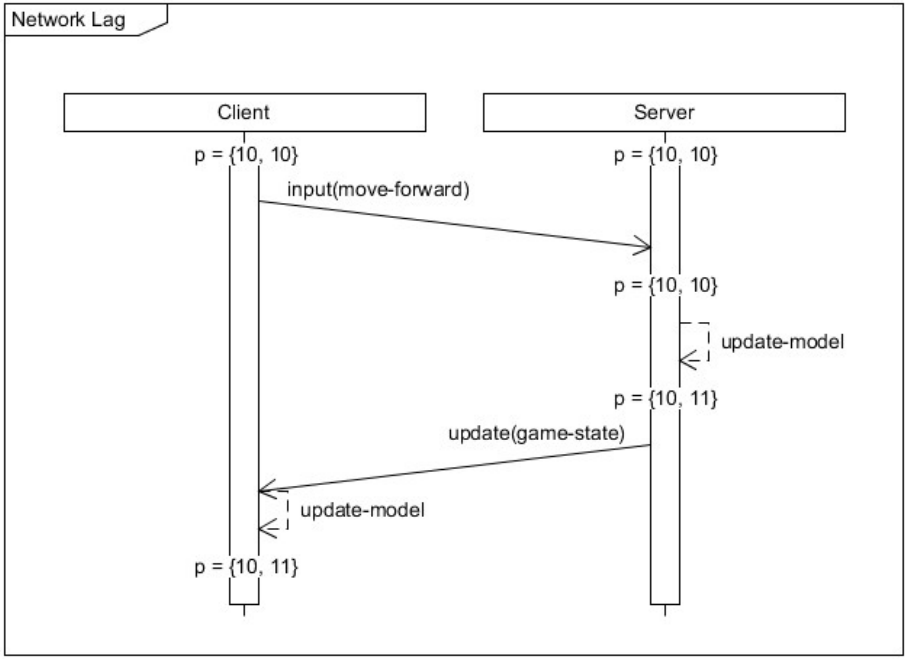
Lets take a more detailed look...

Lag – Ideal Scenario



Lag – Ideal Scenario

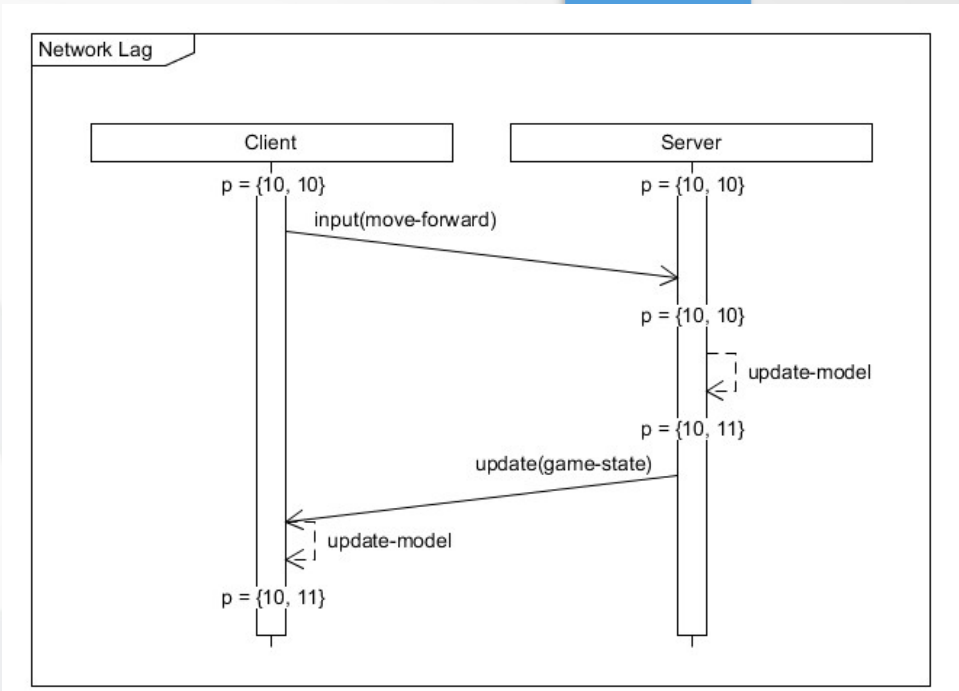
- 12.5 ms to send an input to the server.
- 33 ms to update game at server; assuming packet arrives just before start of the game model update.
- 12.5 ms to send an updated state
- 33 ms to update & render game at the client

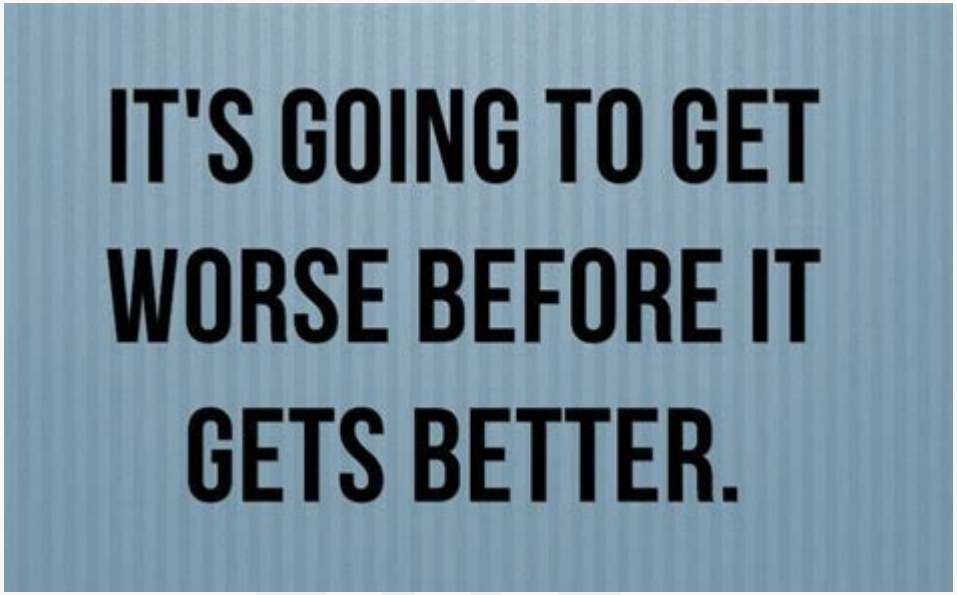


- Total time from input to seeing a change on the screen: 91 ms, or 11 times per second (um yeah, not so great)

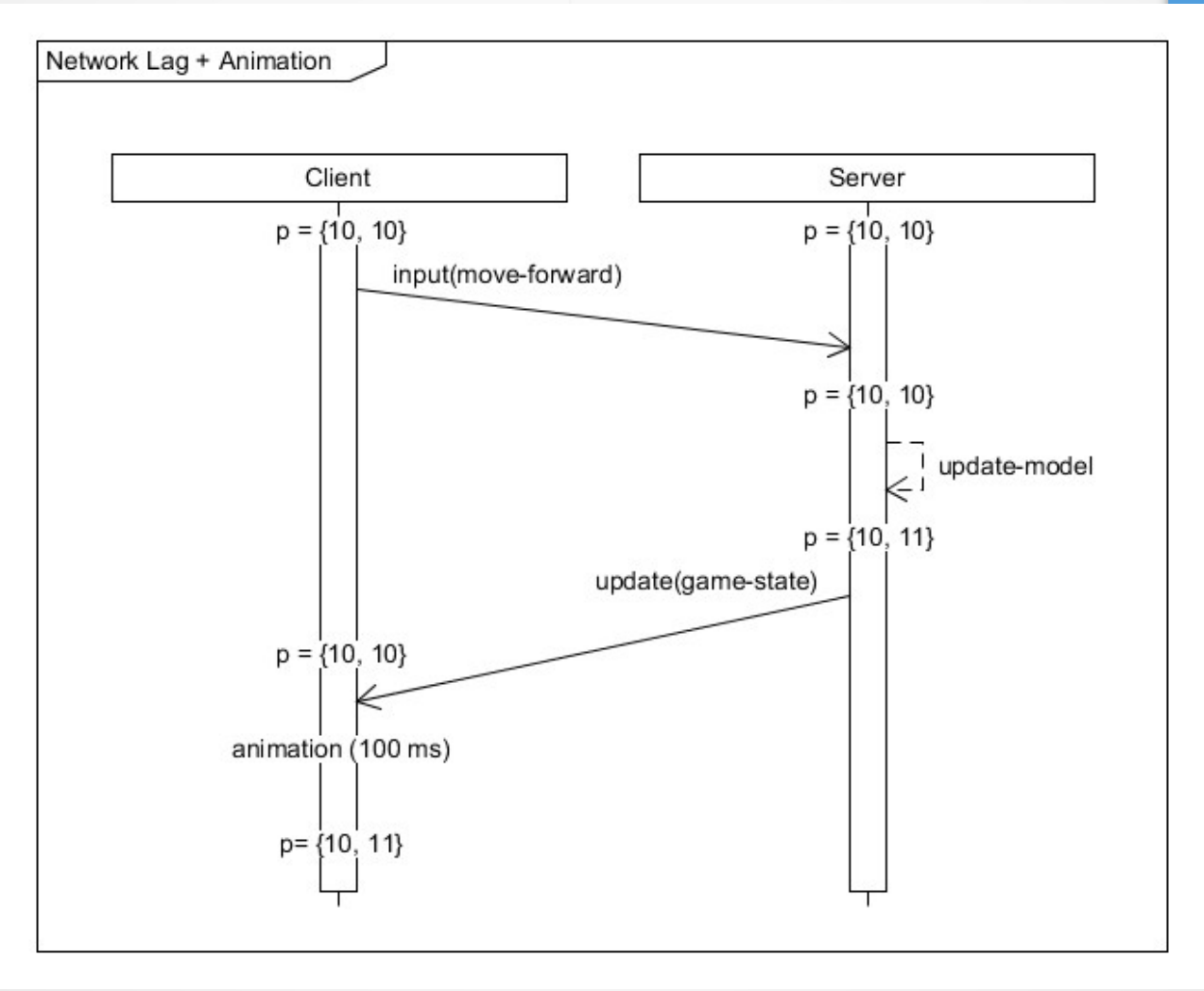
Lag – Worst Case Scenario

- Input occurs right after client sends inputs to server. Wait until next game loop to send: 33 ms
- 12.5 ms to send an input to the server.
  - Packet arrives right after start of sever updating game model. Wait until next game loop to process: 33 ms
- 33 ms to update game at server
- 12.5 ms to send an updated state
  - Update shows up just as client begins local update and render. Wait until next game loop to process: 33 ms
- 33 ms to update & render game at the client
- **Total time from input to seeing a change on the screen: 190 ms, or 5 times per second**




IT'S GOING TO GET
WORSE BEFORE IT
GETS BETTER.

Lag – It's Actually Worse!



190 ms is when the animation begins; now add in time for the animation to complete: 290 ms
(what!?!)

...Demo Basic...

Basic - Server

```
function gameLoop(currentTime, elapsedTime) {
    processInput();
    update(elapsedTime, currentTime);
    updateClients();

    if (!quit) {
        setTimeout(() => {
            let now = present();
            gameLoop(now, now - currentTime);
        }, UPDATE_RATE_MS);
    }
}
```

```
io.on('connection', function(socket) {
   ...
   socket.on('input', function(data) {
      inputQueue.push({
         clientId: socket.id,
         message: data,
      });
   });
   ...
}
```

Basic - Server

```
function processInput() {
    // Double buffering on the queue so we don't asynchronously
    // receive inputs while processing; which can’t actually happen
    // given the JS execution model, but stil...
    let processMe = inputQueue;
    inputQueue = [];

    for (let inputIndex in processMe) {
        let input = processMe[inputIndex];
        let client = activeClients[input.clientId];

        switch (input.message.type) {
            case 'move':
                client.player.move(input.message.elapsedTime);
                break;
            case 'rotate-left':
                client.player.rotateLeft(input.message.elapsedTime);
                break;
            case 'rotate-right':
                client.player.rotateRight(input.message.elapsedTime);
                break;
        }
    }
}
```

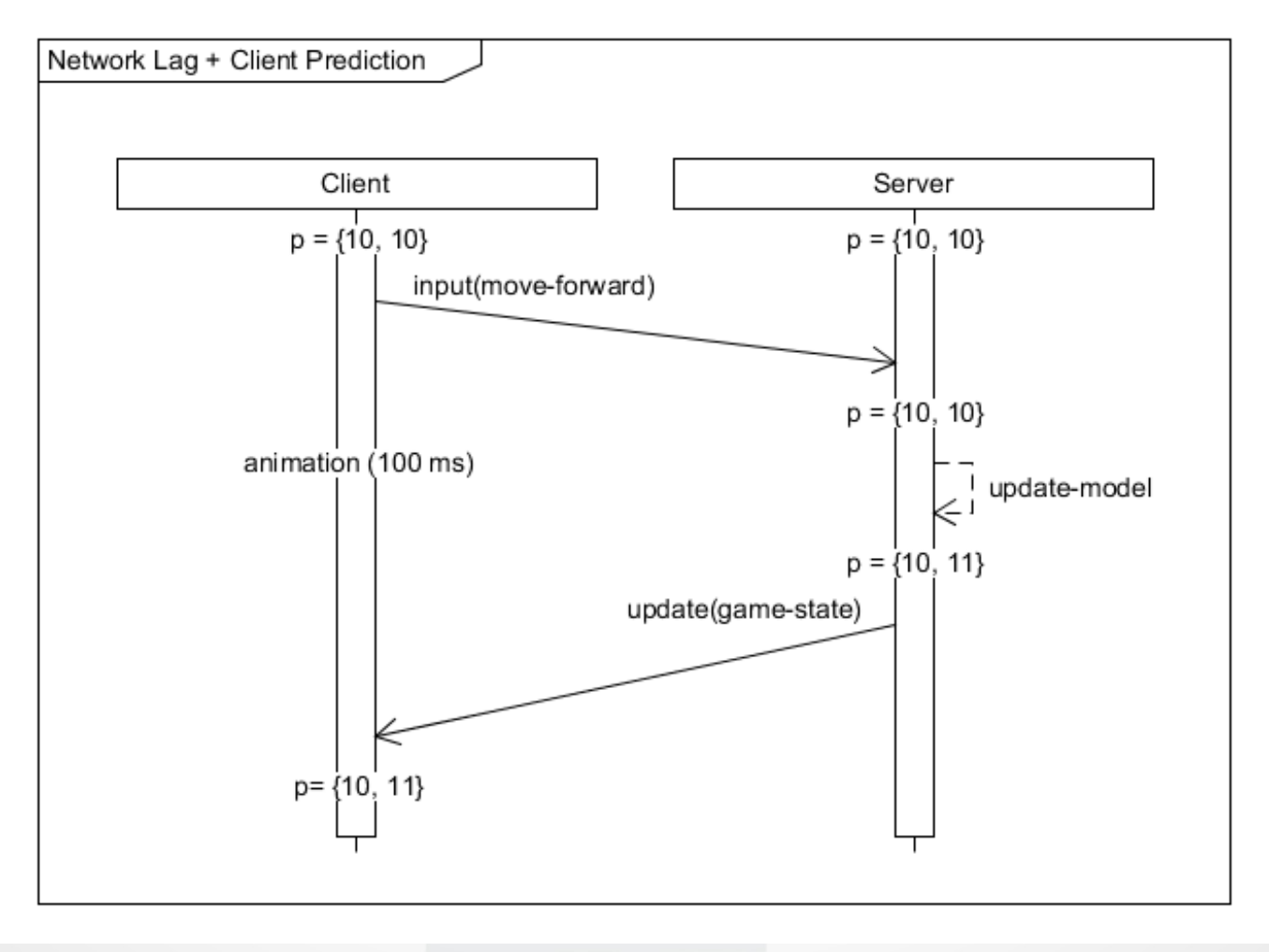
Basic - Client

```
socket.on('update-self', function(data) {
    playerSelf.model.position.x = data.position.x;
    playerSelf.model.position.y = data.position.y;

    playerSelf.model.direction = data.direction;
});
```

What to do?

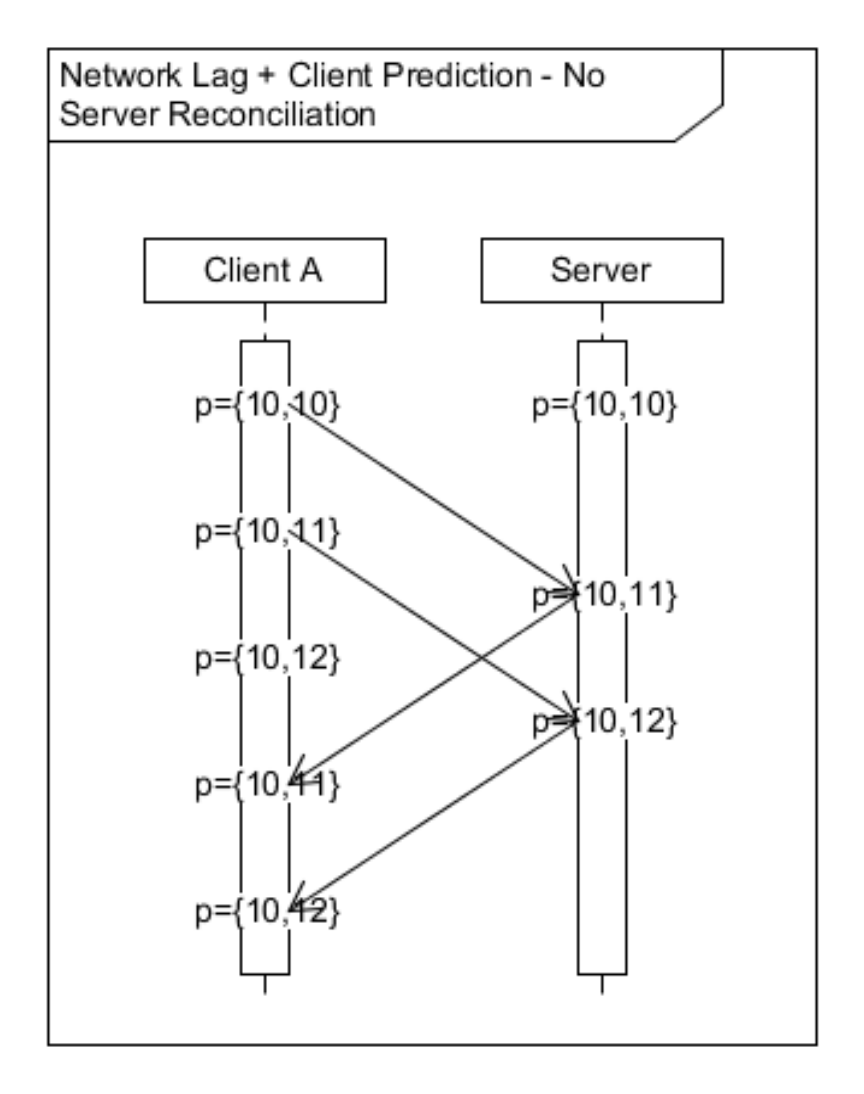
- Core Elements (for this class)
  - Client Prediction
  - Server Reconciliation
  - Entity Interpolation
  - Entity Prediction
- Additional Layers (exercises for the reader)
  - Lag Compensation
  - More Client Prediction
  - Server Prediction

Client-Side Prediction

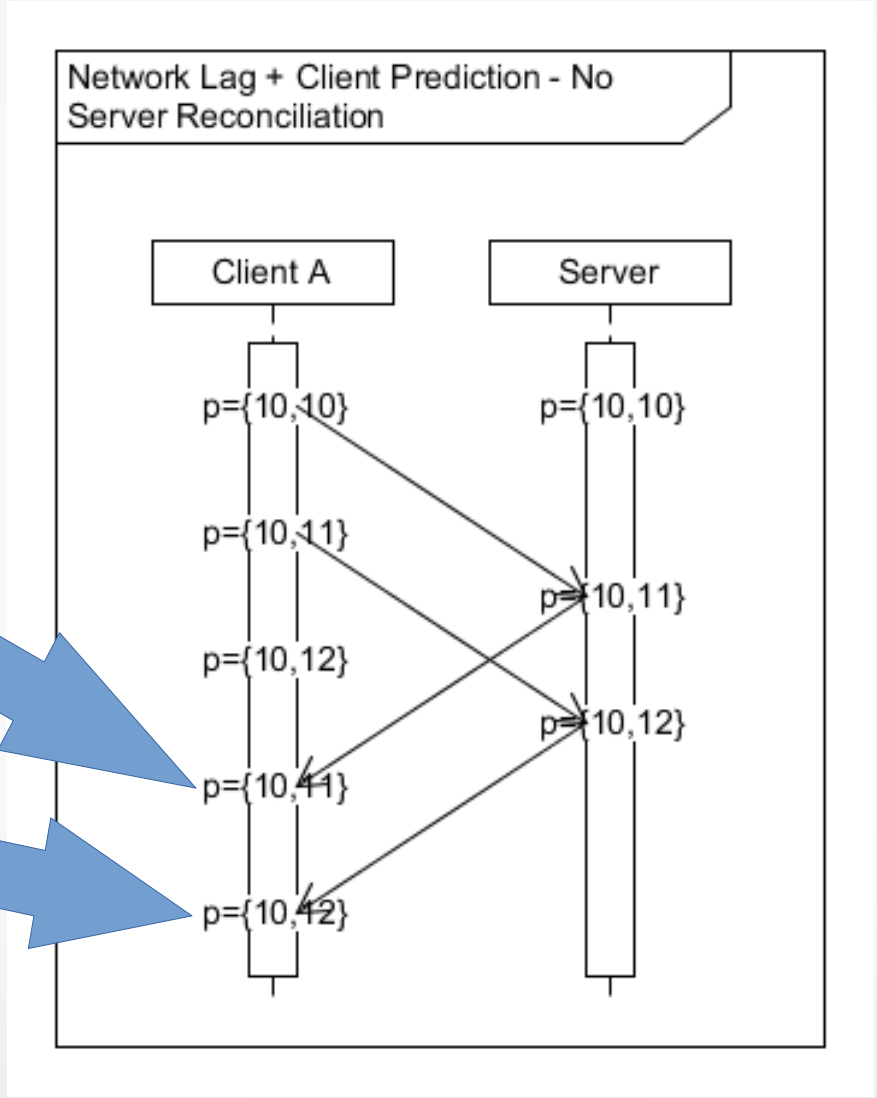


A client can trust itself, even if the server can't. Therefore the client can show action on an input before receiving a game state update from the server confirming the input.

Client Prediction – Not Enough



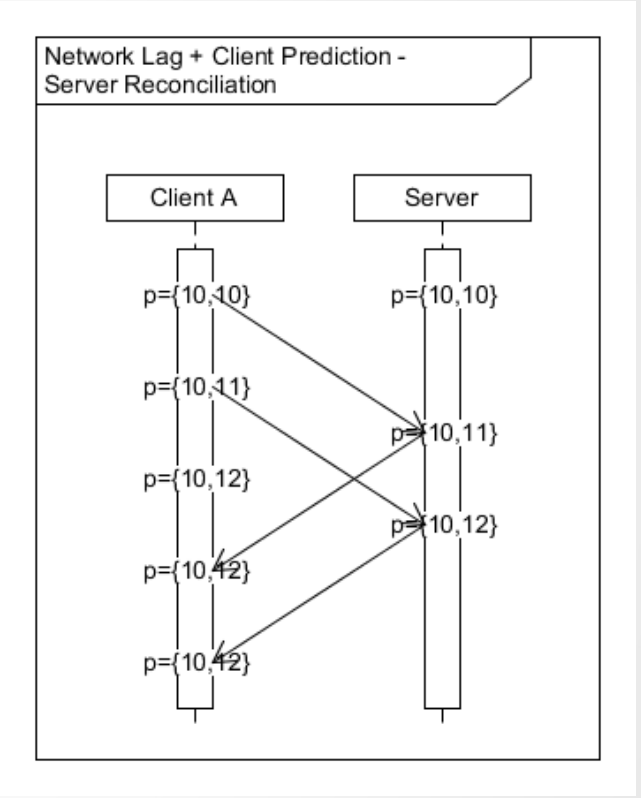
Client Prediction – Not Enough



Server Reconciliation

The client & server have to cooperate, just a bit, to allow the client prediction to not be adversely impacted by server updates. Here is how…

- Client adds a sequence number to each message.
- When server sends an update, it includes last message sequence number it has processed.
- When client receives update, it updates the game model to that state.
- THEN...the client replays all local inputs the server hasn't yet acknowledged.



Client Prediction - Server

```
function processInput() {
    // Double buffering on the queue so we don't asynchronously
    // receive inputs while processing.
    let processMe = inputQueue;
    inputQueue = [];

    for (let inputIndex in processMe) {
        let input = processMe[inputIndex];
        let client = activeClients[input.clientId];
        client.lastMessageId = input.message.id;

        switch (input.message.type) {
            case 'move':
                client.player.move(input.message.elapsedTime);
                break;
            case 'rotate-left':
                client.player.rotateLeft(input.message.elapsedTime);
                break;
            case 'rotate-right':
                client.player.rotateRight(input.message.elapsedTime);
                break;
        }
    }
}
```

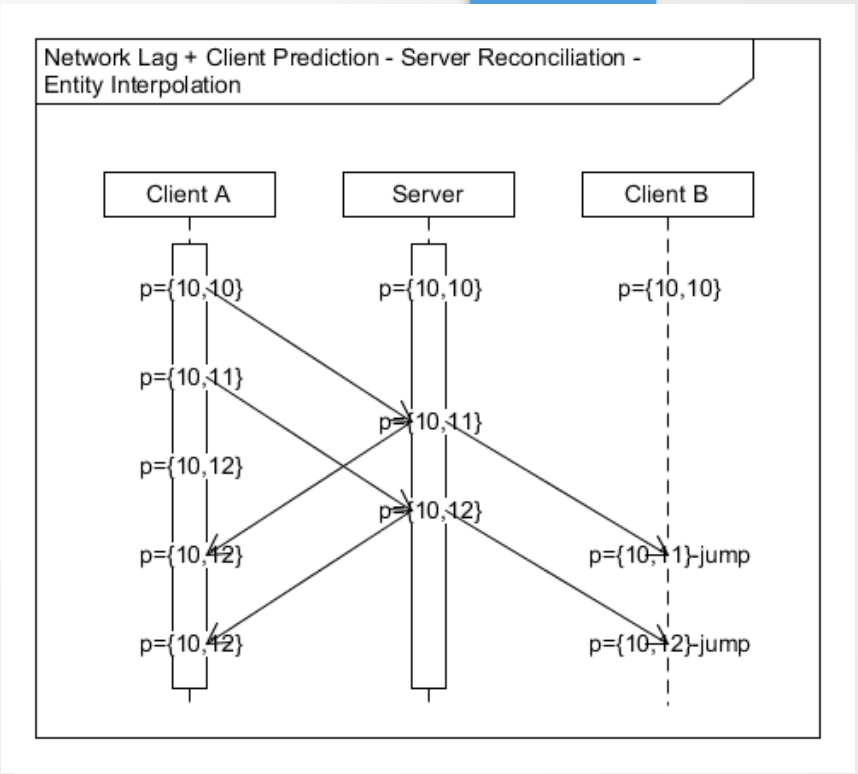
Client Prediction - Client

```
socket.on('update-self', function(data) {
      playerSelf.model.position.x = data.position.x;
      playerSelf.model.position.y = data.position.y;
      playerSelf.model.direction = data.direction;
      // Remove messages from the queue up through the last one identified
      // by the server as having been processed.
      let done = false;
      while (!done && !messageHistory.empty) {
            if (messageHistory.front.id === data.lastMessageId) {
                  done = true;
            }
            messageHistory.dequeue();
      }
      // Update the client simulation since this last server update, by
      // replaying the remaining inputs.
      let memory = MyGame.utilities.Queue();
      while (!messageHistory.empty) {
            let message = messageHistory.dequeue();
            switch (message.type) {
                  case 'move':
                        playerSelf.model.move(message.elapsedTime);
                        break;
                  case 'rotate-right':
                        playerSelf.model.rotateRight(message.elapsedTime);
                        break;
                  case 'rotate-left':
                        playerSelf.model.rotateLeft(message.elapsedTime);
                        break;
            }
            memory.enqueue(message);
      }
      messageHistory = memory;
});
```

Demo Client Prediction

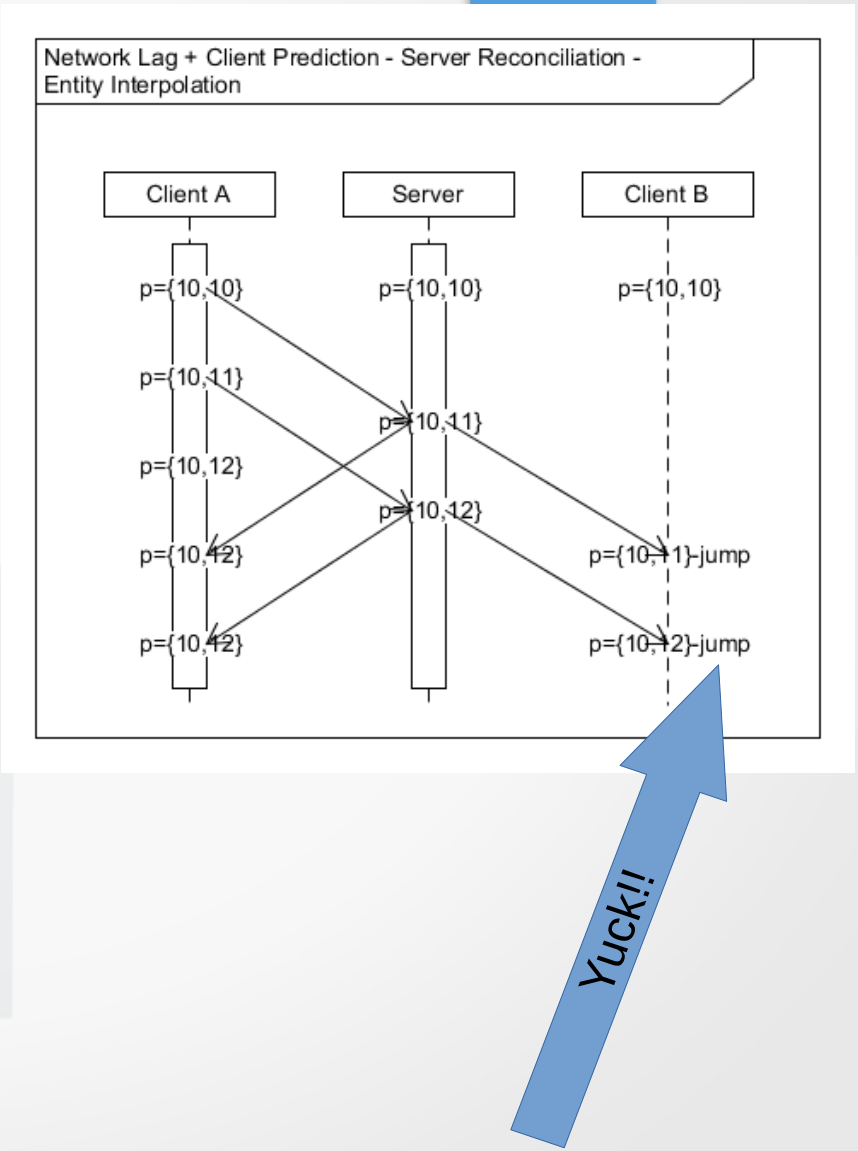
Another Problem – Motivation

T_0 : A sends input to Server
T_1 : A finishes client prediction
A sends input to Server
B sends input to Server (not illustrated)
T_2 : Server updates model
Server sends updated state to A and B
T_3 : A finishes client prediction
T_4 : Server updates model
Server sends updated state to A and B
T_5 : A reconciles self
B moves A (sudden jump)
T_6 : A reconciles self
B moves A (sudden jump)



Another Problem – Motivation

T_0 : A sends input to Server
T_1 : A finishes client prediction
A sends input to Server
B sends input to Server (not illustrated)
T_2 : Server updates model
Server sends updated state to A and B
T_3 : A finishes client prediction
T_4 : Server updates model
Server sends updated state to A and B
T_5 : A reconciles self
B moves A (sudden jump)
T_6 : A reconciles self
B moves A (sudden jump)



How do we deal with this?

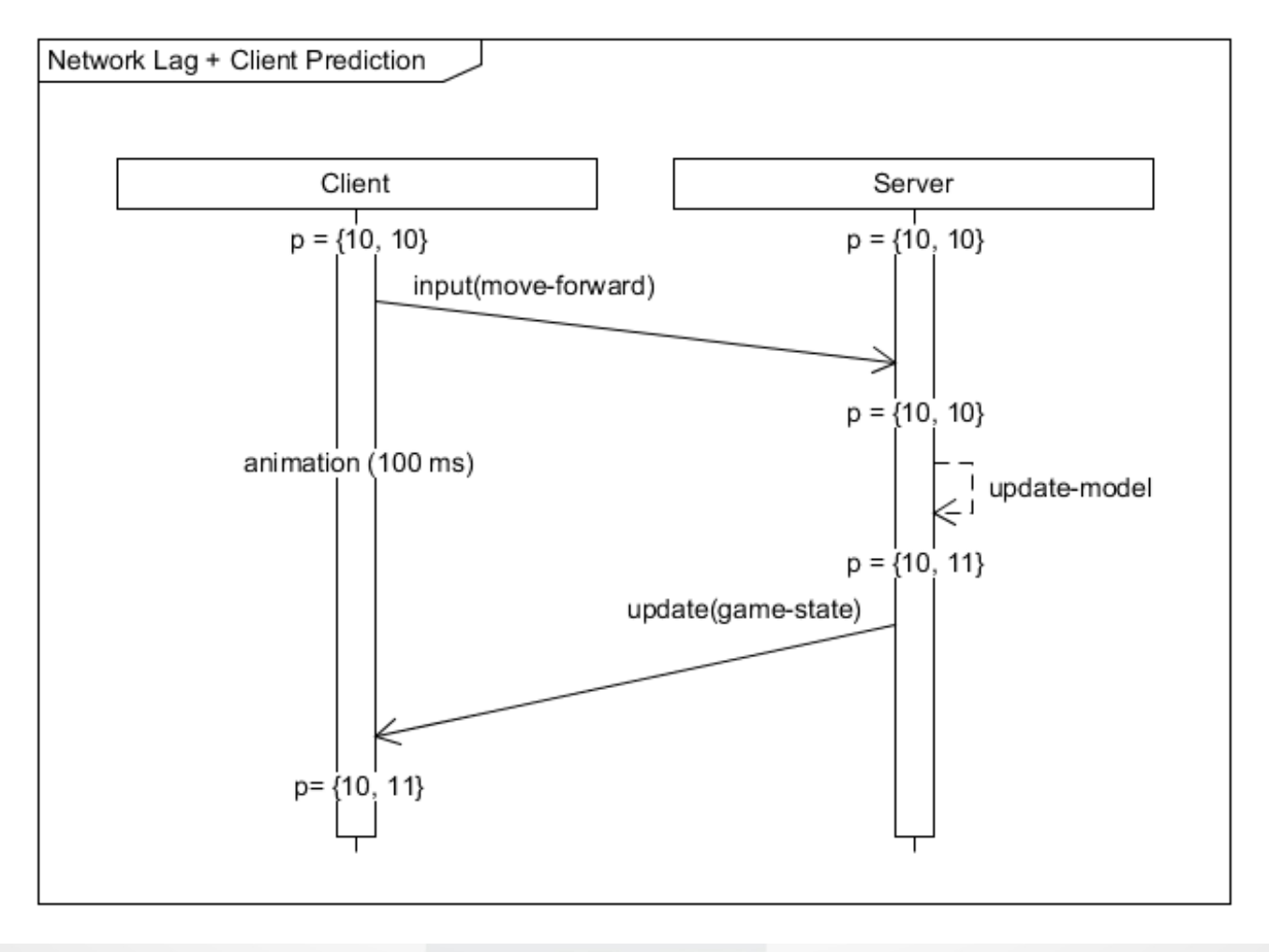
How do we deal with this?

You aren’t going to like it...

How do we deal with this?

You aren’t going to like it...

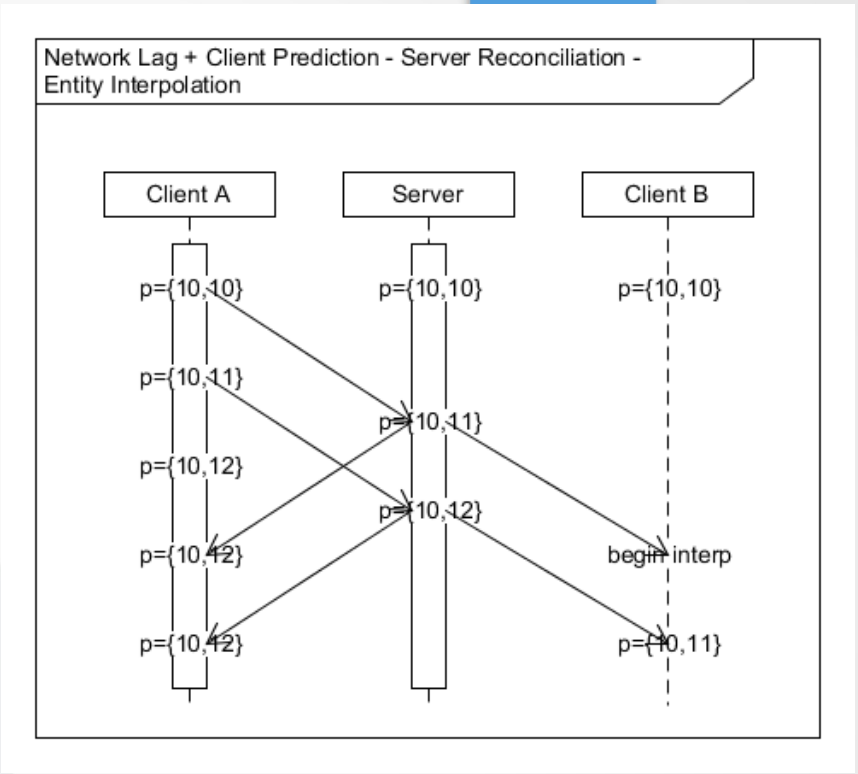
Remember: Client-Side Prediction



A client can trust itself, even if the server can't. Therefore the client can show action on an input before receiving a game state update from the server confirming the input.

Solution – Entity Interpolation

T_0 : A sends input to Server
T_1 : A finishes client prediction
A sends input to Server
B sends input to Server (not illustrated)
T_2 : Server updates model
Server sends updated state to A and B
T_3 : A finishes client prediction
T_4 : Server updates model
Server sends updated state to A and B
T_5 : A reconciles self
B moves A (sudden jump)
B begins interpolating position of A
T_6 : A reconciles self
B finishes previous interpolation
B begins interpolating next position of A



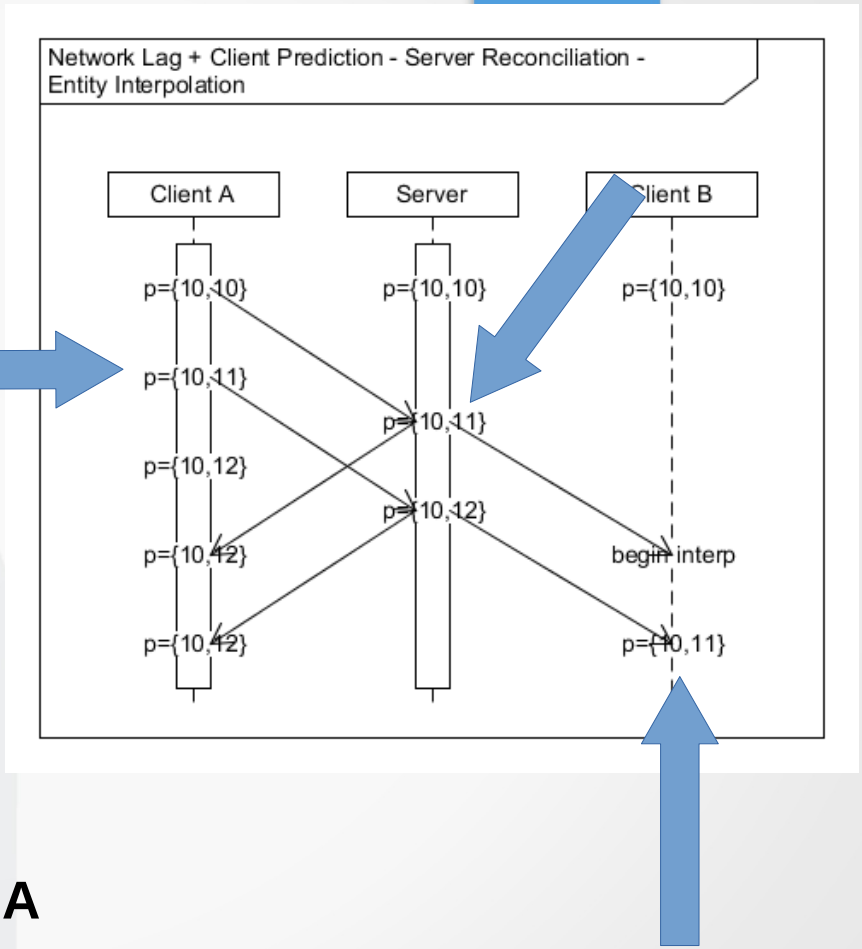
Entity Interpolation - Client

```
socket.on('update-other', function(data) {
    if (playerOthers.hasOwnProperty(data.clientId)) {
        let model = playerOthers[data.clientId].model;
        model.goal.updateWindow = data.updateWindow;

        model.goal.position.x = data.position.x;
        model.goal.position.y = data.position.y
        model.goal.direction = data.direction;
    }
});
```

```
MyGame.components.PlayerRemote = function() {
...
    function update(elapsedTime) {
        if (goal.updateWindow === 0) return;

        let updateFraction = elapsedTime / goal.updateWindow;
        if (updateFraction > 0) {
            // Turn first, then move.
            state.direction -= (state.direction - goal.direction) * updateFraction;
            state.position.x -= (state.position.x - goal.position.x) * updateFraction;
            state.position.y -= (state.position.y - goal.position.y) * updateFraction;
        }
    }
...
}
```

Entity Interpolation – What is Reality?

T_0 : A sends input to Server
T_1 : A finishes client prediction
A sends input to Server
B sends input to Server (not illustrated)
T_2 : Server updates model
Server sends updated state to A and B
T_3 : A finishes client prediction
T_4 : Server updates model
Server sends updated state to A and B
T_5 : A reconciles self
B moves A (sudden jump)
B begins interpolating position of A
T_6 : A reconciles self
B finishes previous interpolation
B begins interpolating next position of A



...Demo Entity Interpolation...

Another Issue

Game object update without player input

Spaceships have momentum

Changes To Implementation – Server

- Player Ship
  - Remove movement ‘speed’, replace with ‘momentum’
  - Add ‘thrustRate’
  - Remove ‘move’ function, replace with ‘thrust’
    - **Key element**: In game.js track ‘lastUpdateTime’
      - First simulate ship momentum, until time when thrust input was received.
      - Then simulate the thrust time.
- Network Messages
  - Remove ‘move’, replace with ‘thrust’
  - Also need to remember when ‘thrust’ message received

Changes To Implementation – Client

- (self) Player
  - Remove ‘speed’
  - Add ‘momentum’ state
  - Add ‘thrustRate’ setting
  - Remove ‘move’ function
  - Add ‘thrust’ function
  - Modify ‘update’ function to change position based on momentum

Changes To Implementation – Client

- (remote) Player
  - Add ‘momentum’ state
  - Modify ‘update’ function to change position based on…
    - Time left to reach goal
    - Momentum over elapsed time

Changes To Implementation – Client

- Game Model
  - Add momentum to connect-ack message
  - Add momentum to connect-other message
  - update-self message
    - Add momentum
    - Replace ‘move’ with ‘thrust’
  - update-other message
    - Add momentum
  - In keyboard handler replace ‘move’ with ‘thrust’

Server : game.js

```
socket.on('input', data => {
     inputQueue.push({
          clientId: socket.id,
          message: data,
          receiveTime: present()
     });
});

case 'thrust':
    client.player.thrust(input.message.elapsedTime, input.receiveTime - lastUpdateTime);
    lastUpdateTime = input.receiveTime;
```

Server : player.js

```
that.thrust = function(elapsedTime, updateDiff) {
    lastUpdateDiff += updateDiff;
    that.update(updateDiff, true);
    reportUpdate = true;
    let vectorX = Math.cos(direction);
    let vectorY = Math.sin(direction);

    momentum.x += (vectorX * thrustRate * elapsedTime);
    momentum.y += (vectorY * thrustRate * elapsedTime);
};

that.update = function(elapsedTime, intraUpdate) {
    if (intraUpdate) {
        reportUpdate = true;
    } else {
        elapsedTime -= lastUpdateDiff;
        lastUpdateDiff = 0;
    }

    position.x += (momentum.x * elapsedTime);
    position.y += (momentum.y * elapsedTime);
};
```

Client : player-remote.js

```
that.update = function(elapsedTime) {
     let goalTime = Math.min(elapsedTime, goal.updateWindow);
     if (goalTime > 0) {
          let updateFraction = goalTime / goal.updateWindow;
          // Turn first, then move.
          state.direction -= (state.direction - goal.direction) * updateFraction;

          state.position.x -= ((state.position.x - goal.position.x) * updateFraction);
          state.position.y -= ((state.position.y - goal.position.y) * updateFraction);

          goal.updateWindow -= goalTime;
     } else {
          // Ship is only floating along, only need to update its position
          state.position.x += (state.momentum.x * elapsedTime);
          state.position.y += (state.momentum.y * elapsedTime);
     }
};
```

Not Done Yet...

- Lag Compensation
- More Client Prediction
- Server Prediction

- Other resources
  - Papers & Web Articles
    - Value Software: https://developer.valvesoftware.com/wiki/Source_Multiplayer_Networking
    - Gabriel Gambetta: https://www.gabrielgambetta.com/client-server-game-architecture.html
    - Tribes : https://www.gamedevs.org/uploads/tribes-networking-model.pdf
    - Valve :
      - https://developer.valvesoftware.com/wiki/Source_Multiplayer_Networking
      - https://www.gamedevs.org/uploads/latency-compensation-in-client-server-protocols.pdf
  - GDC Videos
    - Overwatch : https://www.youtube.com/watch?v=W3aieHjyNvw
    - Halo : https://www.youtube.com/watch?v=h47zZrqjgLc
      - From the comments, “i’m only 2 minutes in and i already feel like this guy would have been terrible to work with, but probably actually knows his stuff.” Don’t be like that.
    - Mortal Kombat : https://www.youtube.com/watch?v=7jb0FOcImdg