Realtime HTML5 Multiplayer Games with Node.js

http://holiday2010.ogilvy.com

For Ogilvy's 2011 Holiday Card, we decided to make an HTML5 game

We decided that the game should be created in HTML5, and having heard of Node.JS and Socket.IO slightly we decided that we should create a multiplayer game

The game is located at http://holiday2010.ogilvy.com,

To say something about Node.js, the game has been running without a restart since around December 22nd and thousands of games have been played

Benefits of writing your game in JavaScript with Node.js

- JavaScript is a great language
- Node.JS and V8 are very fast
- Interoperability between the client and the server
- Shared code base means no translation layer or violating DRY principal
- Many JavaScript libraries already exist to help your game do cool stuff
- When writing your game no mental model shift required when switching back and fourth

* Javascript is a really great language for writing games, it has all the tools you could ask for due to it's dynamic nature

* Node.JS and the V8 engine is extreme fast, on our game the node.js process is using only 1 thread and runs at about 3-4% CPU usage while running 8 simultaneous games each with their own collision engine instance

* The client and the server are both running javascript. This means that we can share logic between both when needed.

For example having a Vector math utility class. Another example in our case was the NetworkMessage class.

Even though the server is the authoritative source, and is the one that is actually running the game. It can be beneficial to share some piece of logic between the two

* Many javascript libraries exist, relating to cool game stuff. With Node.js using one of those libraries to help you create your game is relatively easy to do.

Example: Box2D js implementation!

* Finally because the client and the server are both running in javascript. It makes it a lot easier to write a cohesive game. Switching between different mental models be it java or python or a C++ backed server can make you lose focus on writing the game

Interoperability between Client and Server

Example - Shared NetworkMessage class

```
RealtimeMultiplayerGame.namespace("RealtimeMultiplayerGame.model");
RealtimeMultiplayerGame.model.NetChannelMessage = function(aSequenceNumber, aClientid, isReliable, aCommandType, aPayload)
{
this.seq = aSequenceNumber;
this.id = aClientid;        // Server gives us one when we first  connect to it
this.cmd = aCommandType;
this.payload = aPayload;
this.messageTime = -1;
this.isReliable = isReliable;
};

RealtimeMultiplayerGame.model.NetChannelMessage.prototype = {
isReliable : false,
cmd : 0,
aPayload : null,
seq : -1,
id : -1,
messageTime : -1,
encodeSelf: function() { ... }
      decodeSelf: function() { ... }
}
```

Key Take Away: The fancier you make things, the more you benefit from this.
No middle man layer to introduce bugs

Well that's cool, let's write a game now!

...problems arose

The initial (naive) approach

Clients send information about their player to the server:

```
var info = {};
info.position = {x:10, y:15};
info.velocity = {x:0.5,y: 0.2};
info.currentWeapon = 1;
info.radius = 10;
this.netChannel.send( info );
```

Server re-broadcast that info to all other clients:

```
function broadcastInfo( sendingClient, messageInfo )
{
    for(var aClient in allClients) {
        if(aClient != sendingClient) {
            aClient.connection.send( messageInfo );
        }
    }
}
```

Problems with this approach

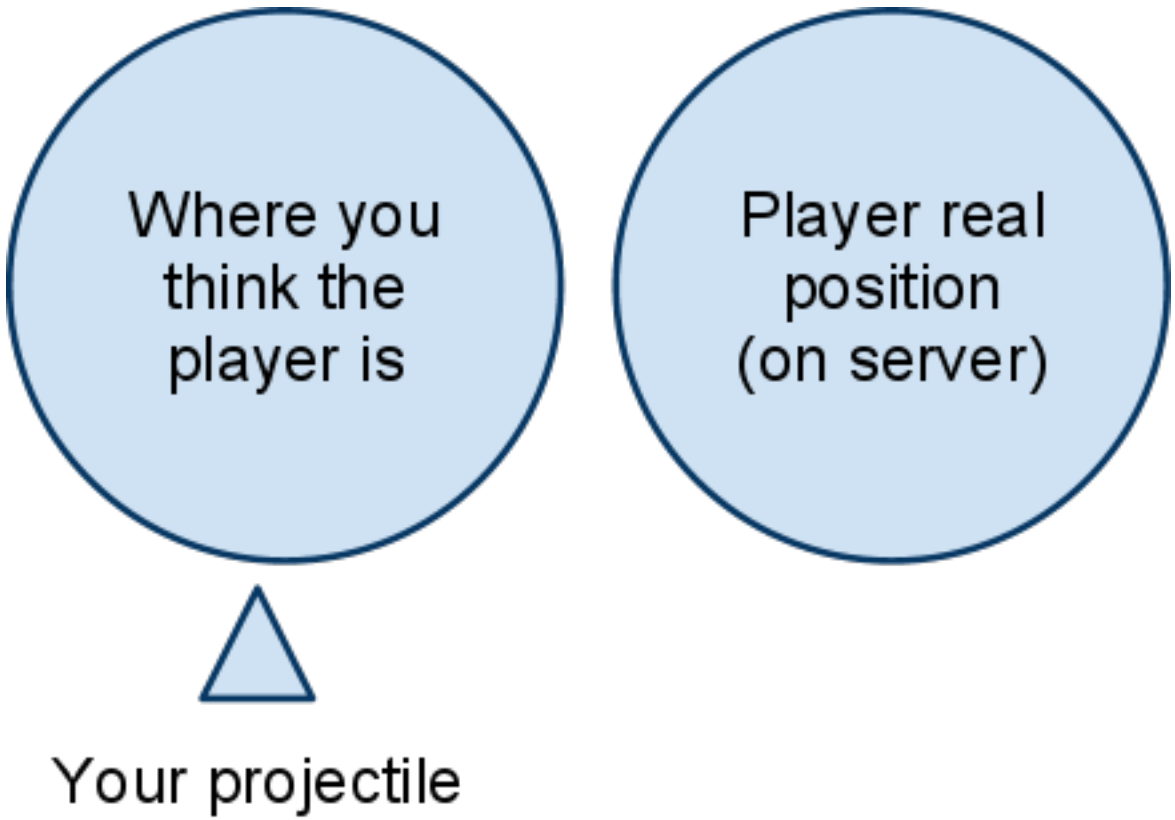
- We cannot trust users to give us correct info
  - Position?
    - Everywhere
  - Velocity?
    - Infinite
  - Weapon fire hit other player?
    - Yes! Twice!
    - (and both were headshots)
  - Health?
    - Number.MAX_VALUE

The main problem with this approach is that we cannot trust the users to provide us with correct information about their current state.

A nefarious can enter the game, and report that other plays projectiles never collide with them, and that their health is always Number.MAX_VALUE

Problems with this approach

When you draw based on last received network information, you can never represent a the true motion path (the "bouncing ball" problem)



Another problem with this approach is determining where the collision takes place.

If you're locally playing the game, and placing users at their last reported position -

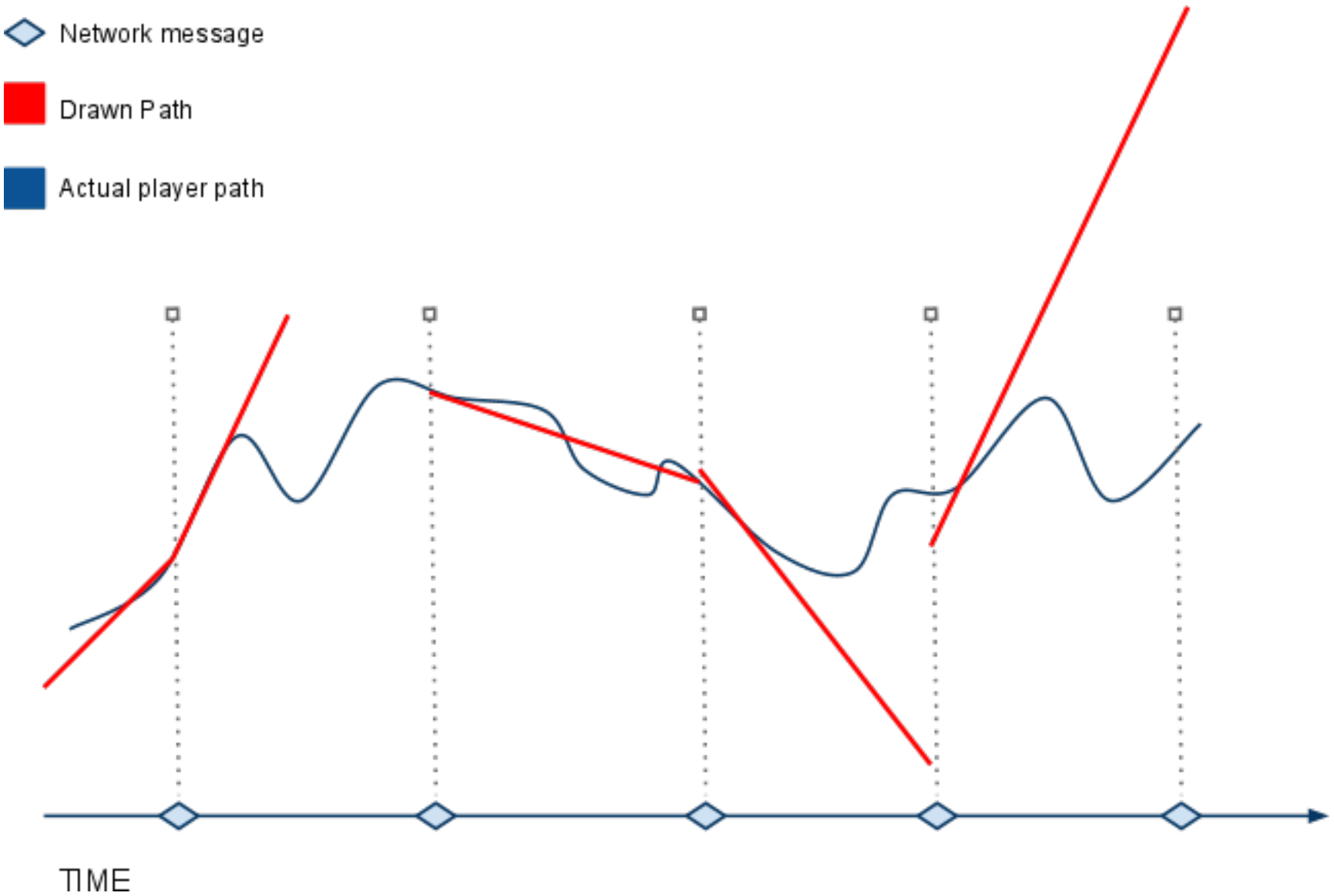
it might appear that you have hit the other player with your projectile, however when the server checks this information will report that your projectile was and that player in fact did not collide.

This is obviously a very large problem, because it also happens in reverse.

You can locally dodge a projectile narrowly however, when that data gets to the server, you have already been considered hit.

Problems with this approach

When you draw based on last received network information, you can never represent a the true motion path (the "bouncing ball" problem)



A problem with this approach is that as we receive new position and velocity information regarding an entity from the server, when we draw the entity using this information it causes an abrupt shift in motion, and we will continue along this motion path until new more update information is received from the server.

Players and entities tend to be very jumpy, so as they move around they are rarely traveling along a straight path and velocity is usually damped.

This is sometimes referred to as the bouncing ball problem, because if you visualize the parabola of a ball thrown in the air as gravity accumulates and eventually sends it back down smoothly. It is very difficult to achieve that motion using only the last instantaneous velocity information
---

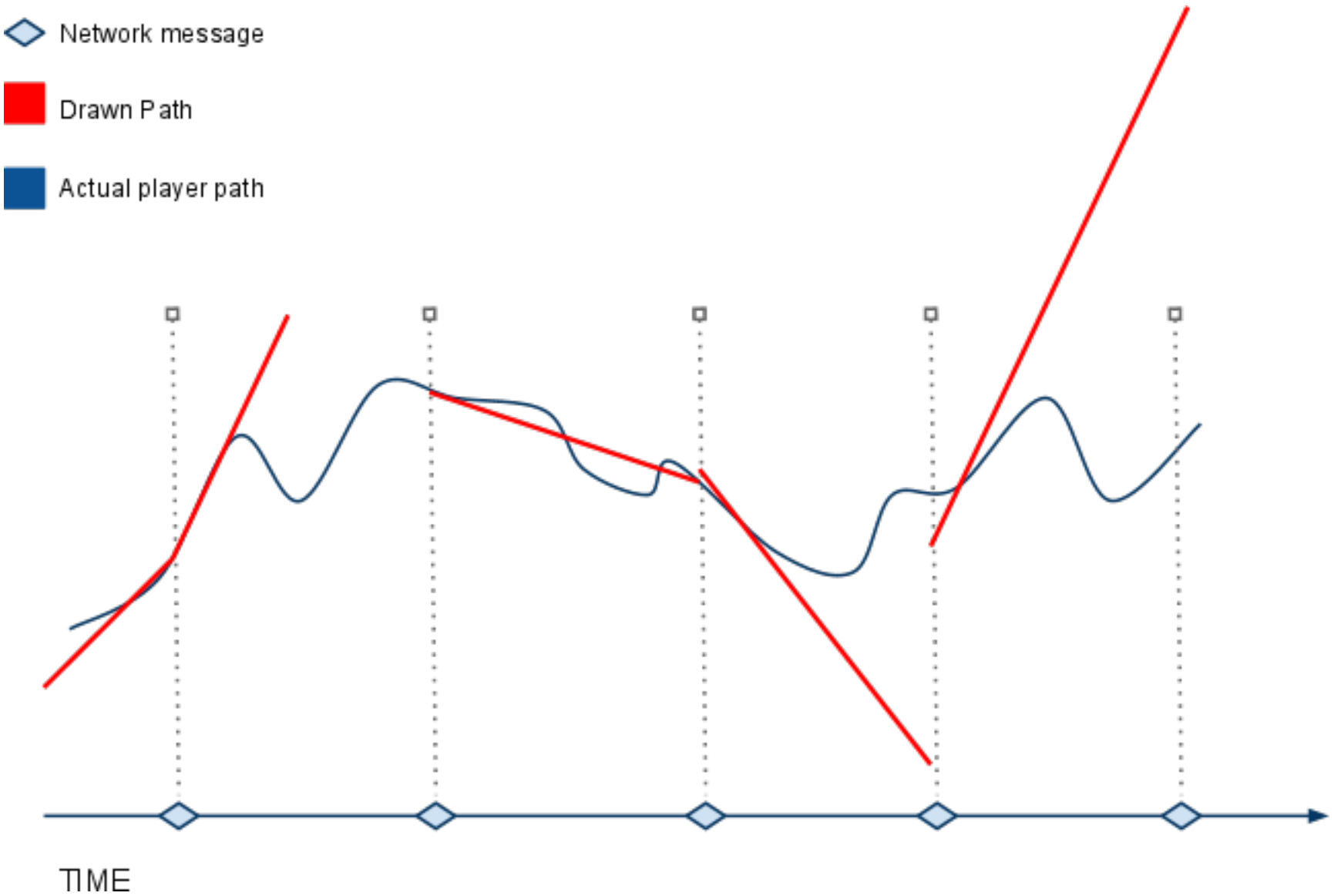
The dotted lines represent times when we receive new information from the server.

The red line represents the drawn path that uses only last instantaneous velocity will move along the incorrect path continuously and when it receives newer updated information will move using the velocity at that time

This problem can be lessoned by receiving more world state samples from the server.

Problems with this approach

When you draw based on last received network information, you can never represent a the true motion path (the "bouncing ball" problem)



Another problem with this approach is packet loss.

If one of the packets as shown by the dotted lines is dropped, the user will travel along an incorrect path for a longer period of time.

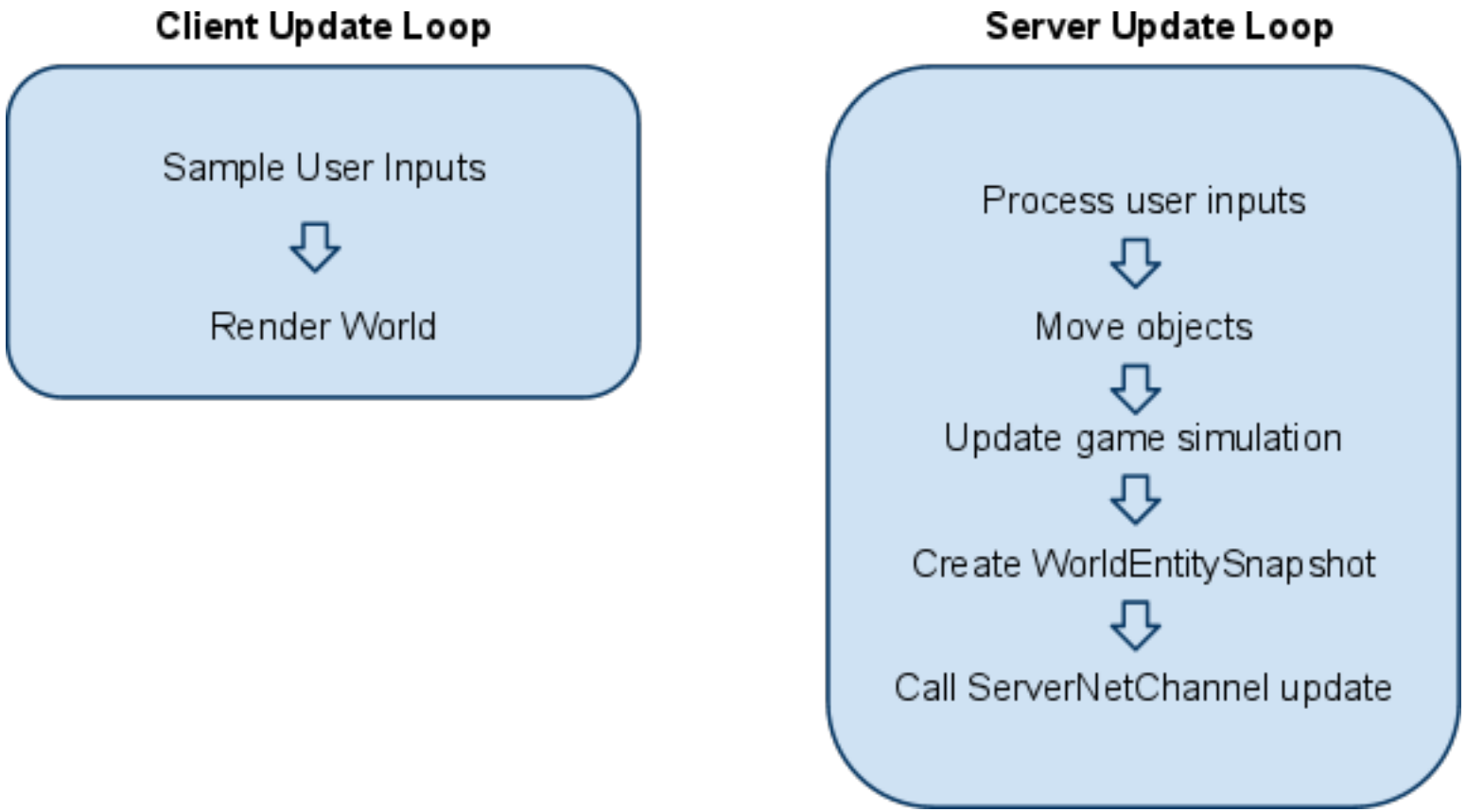
Of course this time is measured in milliseconds so it's not the end of the world.

However in action, it feels very wrong visually because things don't change position instantnously in real life

A Different Approach

Server / Client Model

Server / Client Model



So I decided to look around, and see how this was generally done. After all people have been making realtime multiplayer games for a while now.

I found some interesting sources of information, specifically the QuakeWorld source code as well as some Valve white papers on networking

In this approach there is a single authoritative server which is playing the game, and clients only send their sampled input data

For example, i send only the fact that my spacebar key is being pressed down the server decides what that means in terms of the game and when it decided i can -
--

With this model we completely sidestep many problems of previous implementation

Cheating is also made more difficult slightly, because at most a player can tell us 'hey all my keys are being pressed' which does not really gain them anything anyway.

Rendering the world

- Our nervous system is actually designed to work around latency
- If we can apply a constant smooth latency to an object it is easy for a human being to adjust

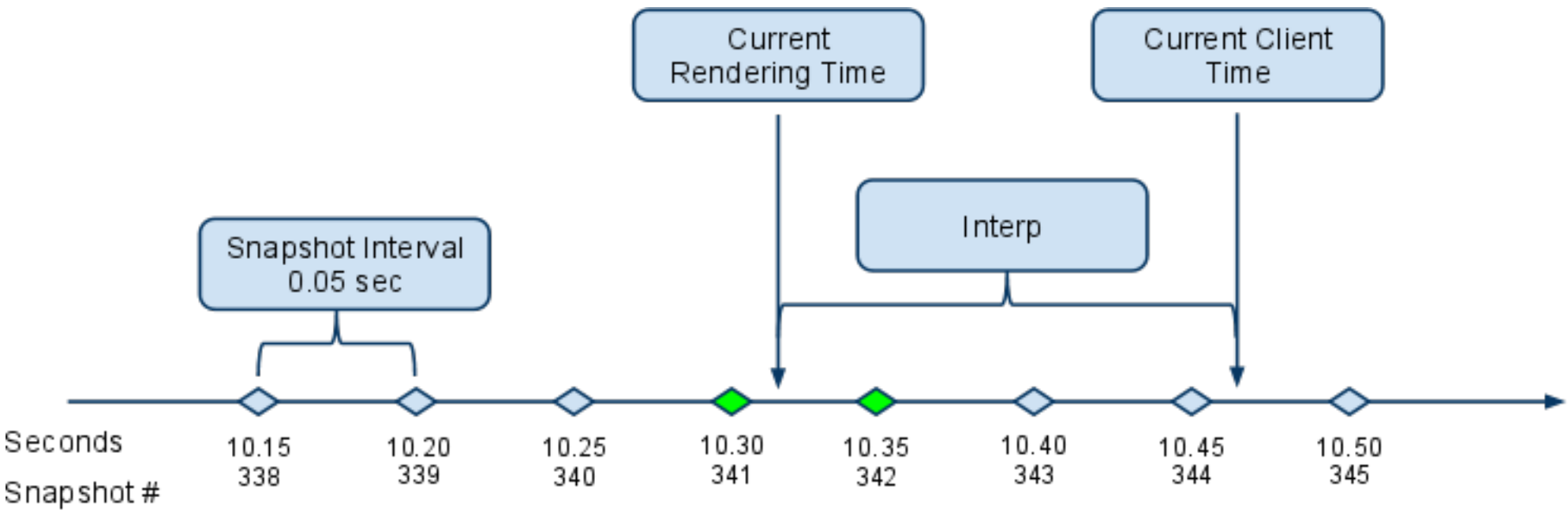
Human beings are very good at adapting to a latency, and predicting what will happen on the short term scale.

This is when i had the eureka moment going through some of valves Source engine wiki.

The key piece of the puzzle is that we render all clients N milliseconds backwards in time at all times. This number is an arbitrary number, so i'm going to go with 75 for the rest of the presentation

Entity Interpolation

Smoothly moving between two known world updates



So we set up a system in place, where the client receives a high fidelity updates from the server at a discrete interval

We store those WorldUpdates in an array.

When it comes time to draw the world on the players screen we simply draw them at currentTime - interpAmount

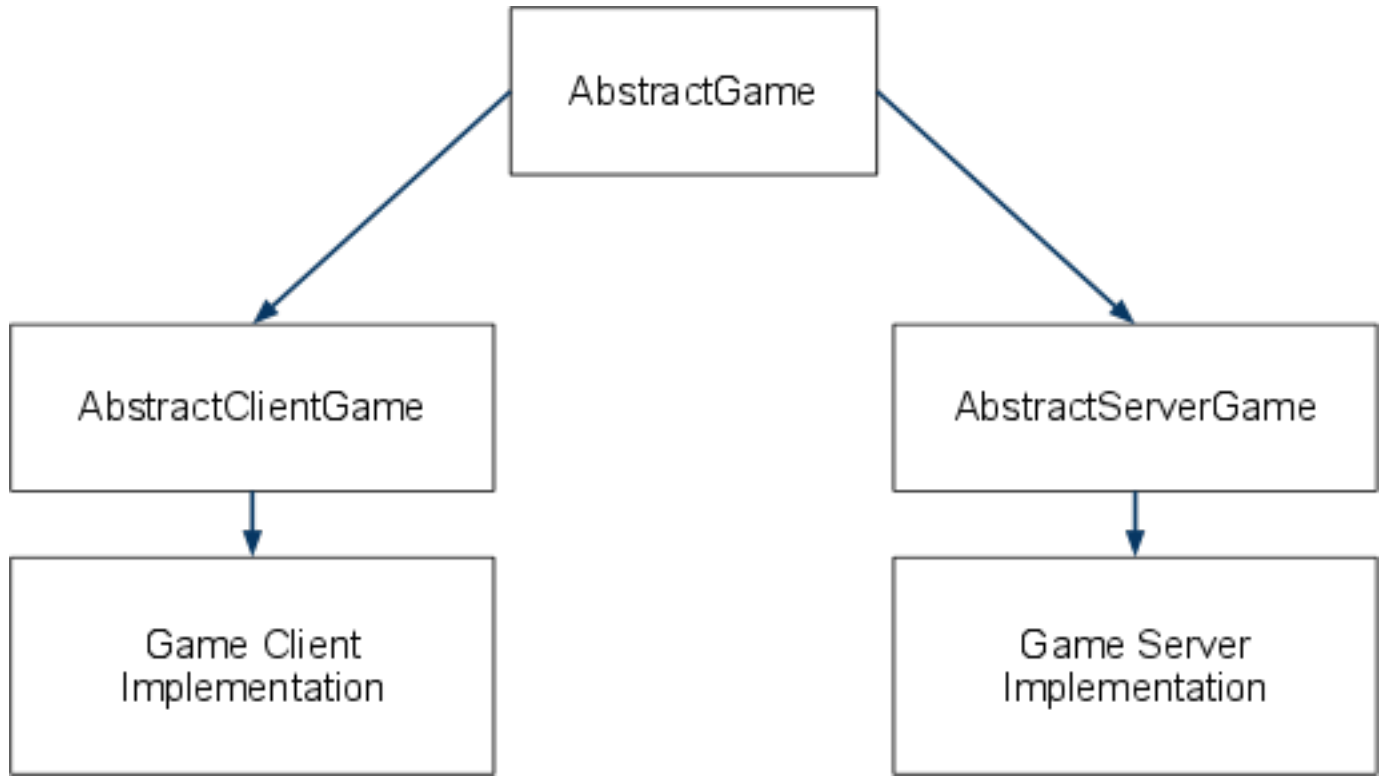
We'll call this value renderTime, which again is simply the currentTime minus 75 milliseconds

To do every render, we first find the two updates that our renderTime falls between with a simple loop

Once we find those two updates we use a Linear interpolation function to position our object precisely along it's path.

If a packet is dropped, for example if we never receive packet 343 on this slide we can still render between two known packets 342 and 344

Using RealtimeMultiplayerNodeJS



RealtimeMultiplayerNodeJS

Demos / Repo

http://bit.ly/RealtimeMultiplayerNodeJs

Questions?

Mario Gonzalez
Senior Applications Developer
Ogilvy & Mather, New York

mariogonzalez [at] gmail [dot] com
https://www.github.com/onedayitwillmake
http://www.onedayitwillmake.com
@1dayitwillmake