Project Lamancha

The Whys

This all started with the simple fact that implementing a board game on the computer is really... really hard.

It is especially hard with a traditional web stack since it requires a game developer's bag of techniques. However, it is still a complicated endeavor since modern board games can be exceptional beasts to implement simply due to the massive size of the state machine to satisfy all the rules over an unreliable network. The core goal of Lamancha is to lower the engineering and design bar of implementing a board game.

The secondary goal of Lamancha is to build a new web. The web feels very unhealthy right now. Ads are everywhere. So many exploits on the browser. You are being tracked. You are one zero day away from exposing your machine to malicious code. This is not a good thing.

CORE VALUES OF THE NEW WEB

- All communication is done with latest transport security, and the security aspect will be evergreen within a year.
- All communication is point to point between a single user and a single server. The “new browser” will not share resources across “sessions”.
- Compute becomes a common commodity similar to the power grid. The end game is to drive ISP to adopt a compute standard. (Rally the edge to tackle the cloud)
- All or nothing approach to access. Either you have a fully secure line or you don't. This eliminates pesky corporate firefalls, and this probably eliminates China access.
- The standard interface will allow minimal client code which is stateless in nature and optional. The client code will resemble Excel without VBA.
- Experiences should be durable without the need for a database; databases should be optional.

Ideas that Power the What of Lamancha

The goal is to make it easy to implement a board game, so let's start by considering an “all open” game like Chess or Checkers where there is no hidden state per player. Without secrets, then players could share an open computer, so we ask:

Is it easier to implement a board game that runs on a single desktop (with players swapping a chair) or runs on multiple clients with a server? Clearly, there should be less code with a single desktop as there is no synchronization of clients and all the code is in one place. The state management is in one place, and all decisions are made by the software. The software is then responsible for *asking which player some question*.

Since we made an assumption around the game being “all open”, we then ask: what is the minimal next step (from a code perspective) to eliminate that assumption (without locking down the current monitor). That is, how do we introduce hidden state per player.

Well, what if all the players had their own monitor and input devices and the software knew which player was bound to which. The software is still running on a single desktop, so that is still a positive. However, since the software knows where the player is, then much of the code to inform the players to rotate chairs has gone away, so this should simplify the engineering process.

Since multiple input devices are not typical, we then ask a slightly different question. Instead of having multiple monitors with separate input devices, what if each player had their own machine and then used software like remote desktop to use the game. This keeps the board game software simple, but creates a debt to make the “remote desktop client” work well.

This is the first core idea of Lamancha, there is a browser that communicates to a server with a protocol that is not game specific. Instead, the protocol focuses on two things.

First, the server is able to stream rendering primitives to the client. The easiest model is to stream a video feed, but this can be torn down with a more efficient scene graph representation.

Second, the server is able to ask the client for events. That is, the server could say “if the client clicks in this region, then send me this event”. Alternatively, a more extreme version is for the client to stream the mouse events (down, move, up) to the server.

These two aspects define the game that Lamancha is playing on the client. Namely:

- How does the server stream rendering primitives to the client?
- How does the server collect events from the client?

Answering these questions will produce a platform that allows servers to behave like a single desktop game (which is much easier to implement).

So far, the networking aspect has been removed from the engineering planning for the board game implementer. The next step is then understanding the state machine of a game. There are two elements of the state machine: explicit state and implicit transaction state, so we ask.

Is it easy to build a formal state machine for every aspect, or is it easier to just write code? Clearly, code is easier to write than modelling a state machine for every single aspect of a game. This is the entire purpose of good programming languages since we could just do everything with a turing machine or machine language. We ideally want to use good programming language primitives.

So, let's consider a deck builder game. Each player has a deck and a current hand where players play a card if they have an action available; players start their turn with one available action. The core complexity of such a game is in the diversity in behavior of the cards.

With this deck builder, explicit state is both easy and unavoidable. For instance, the explicit state would be: each player's deck, each player's hand, and the current player. You can't get away from this, but what about the behavior for each card?

Each card, to some degree, is a transaction across the entire explicit state. As a transaction, it has implicit state that defines who and what actions are needed to happen, and many games put those decisions in the hands of the other players. Some cards are easy, and some are … complex.

Let's consider an easy card. An easy card would be “Draw a card into your hand, then play another card”. This is easy because it only manipulates the player's hand and requires zero additional decisions. The code for this would be something like:

```
DeckOperation.Draw(CurrentPlayer.deck, CurrentPlayer.hand);
CurrentPlayer.Actions ++;
```

What makes this easy is that it simply changes the state and requires zero decisions from any other humans.

Now imagine a card like this “All other player reveals three cards from their deck. You pick a card to discard for each player. All other players then discards one and places the other on top of their deck”. This is a very complex card, but it illustrates that a typical “do it yourself state machine” is going to be tricky to do in a sustainable way. The reason is that all players are engaged to make a decision and the current player has to make decisions proportional to the number of other players. This is complex.

What if we treat the player as a server that can be asked questions? This is where co-routines (something any decent language should have) come into play, and the core mechanism is that the server can ask each player a question and may block the server for a result.

```
var others = ForEachOtherPlayer(async function(otherPlayer) {
  // Reveal 3 cards
  for (var k = 0; k < 3; k++) {
    // draw a card [from deck] [to deck]
    DeckOperation.Draw(otherPlayer.deck, otherPlayer.reveal);
  }

  // ask the current player to decide on a card
  var card= await CurrentPlayer.DecideCard(otherPlayer.reveal);

  // move a [card] [from deck] [to deck]
  DeckOperation.MoveCard(card, otherPlayer.reveal, otherPlayer.discard);

  // ask the other player to make a decision on the remaining cards
  var otherPromise = otherPlayer.DecideCard(otherPlayer.reveal);
  return otherPromise.then((nextCard) => {
    DeckOperation.MoveCard(nextCard, otherPlayer.reveal, otherPlayer.discard);
    // move all cards [from deck] [to deck]
    DeckOperation.MoveDeck(otherPlayer.reveal, otherPlayer.discard);
  });
});

await vasync(others); // wait for all players to finish
```

This code is fairly simple, and it requires a corpus of explicit state machine mechanisms (like deck operations) which may also be asynchronous. As a side bonus, this also means that the server can control time and define the animation for each card.

This vastly simplifies the execution of the diversity of cards, but it creates a new problem.

With state now held in memory, how do we to operate the server in a reliable way. Since the goal is to make it easy, then we need to talk about the developer and operational experience and how these impact game play experience.

It is easy to save the explicit state. But, if you throw away the transaction state, then players have to repeat questions. This gives an opportunity to cheat based on a server failure. It also wastes time and creates confusion.

It should be possible to serialize the state of the game to maintain the explicit state and preserve the implicit transaction state. This means we should be able to survive crashes and deployments without affecting the playing experience.

This has been traditionally handled by having a stateless logic tier with a database, but we broke a rule with co-routines and we now have state held by the compute environment. This is a challenge, but not an impossible one. The key to cracking this is that we need any state transition which is asynchronous and relies on players' decisions to be deterministic with the

events it will ask for. This determinism then allows us to wield database techniques such that the game server becomes a highly specialized database itself.

Non-determinism (like a dice roll) can be introduced in any transaction that does not rely on player input. For instance, if you know that dice need to be rolled for some behavior, then you can have a transaction prior to the player's turn roll a bunch of dice such that the sequence of dice is then deterministic for the player. Alternatively, care can be taken to set a random seed for the next transaction. This does, however, require us to be mindful of determinism.

This borders on the boundary of madness, but it is possible to solve with chain replication, and this protocol will be called “Octoprotocol” which is defined in another quip. The key idea is that chain replication will go both ways where implicit state (i.e. events.) are replicated from clients to server and explicit state is replicated from server to clients,. This will provide an environment such that any node may fail and the system can recover from any failure.

These core ideas then define Lamancha

- Separate the client and server via a protocol that puts all code on the server
- Use a good programming language that has co-routines and write code rather than model state
- The game server will be treated like a database and chain replication will sit between the clients and server to provide a reliable experience.

The engineering discipline then to use Lamancha is

- Work within a single server
- Use a server language with co-routines
- Be mindful of non-deterministic behaviors

What actually is Lamancha

Lamancha is about rethinking the core of what powers the web from both a bottom up perspective and top down perspective.

TOP DOWN

From the top, an experience on Lamancha will feel very familiar with the “Splashy” browser. The “Splashy” browser is a very simple client which will connect to a server, and the server will take over. Core differences between “Splashy” and Chrome/Firefox:

- New Rendering and Layout engine designed for:
    - precision so designs transfer well from toolin
    - open and easy specification to implement a browser
    - performance so it's fast
    - efficiency so battery life is preserved for long games
- A streaming protocol between the client and server such that server can
    - control what the screen is showing
    - control what events the client is able to emit

A key element of the plan is to separate the rendering engine into two parts. First, we need an expressive language for design such that designers are a first class citizen. Second, this expressive language then has a transition to the rendering engine such that the rendering engine is easy to implement and predictable. This changes the game a bit, and what this means is that the rendering engine is very generic and portable, but it will be hard to use directly. This requires us to develop tooling to build products for the rendering engine, but this actually a win.

Most modern web development requires some form of a build phase due to the complexities of javascript, css, and html. In the Lamancha world, this is being codified such that the “browser” has only one input. The complexity is then pushed to the tooling side. This then makes it possible for multiple browsers to be created for a variety of platforms quicker since that platform only needs to address a smaller and easier specification which has less magic in it.

This requires the rendering specification to be very generic for a wide variety of scenarios, and there becomes an arms race for how to build design tools to make it easy.

BOTTOM UP

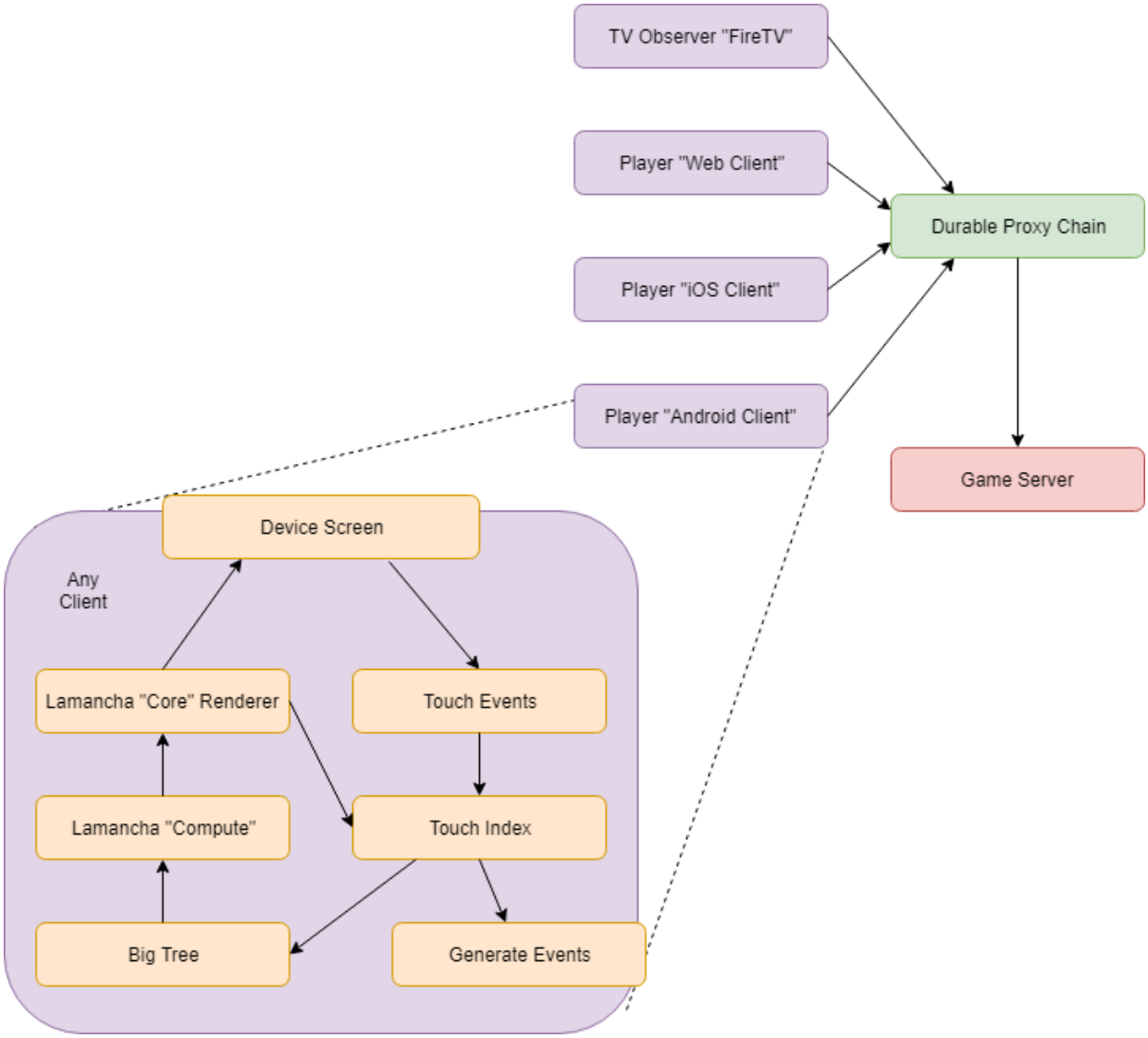
The key idea is that the server is in control of the client experience, so the client will provide the server events and the server will tell the client what to render and what events are possible.

The concept of a database is a troubling one and a great pain point. Instead, the entire application is going to be treated like a database with chain replication.

The goal from the bottom up is to make it easy, very easy, to write

Execution Architecture

The below diagram outlines the entire picture of Lamancha.



CLIENTS

CLIENT ENGINE

DURABLE PROXY CHAIN

GAME SERVER

Engineering Tools