Lamancha, the Core Idea

Why

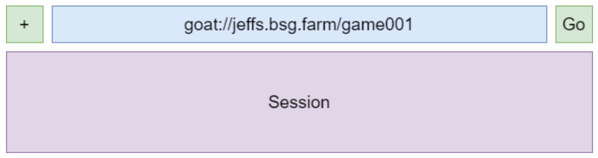
Let's start with what is most important: **Why**. Building build games that enforce rules is a real engineering challenge. It's hard to do because the rules in modern games are vast and complex, and it would be great if implementing these games was easier. Coordinating the actions of multiple people during a single turn is so hard that most products in this space today simply give up and provide players with a sandbox to trust the players to exercise the rules. This makes a ton of sense and is an excellent product hack to make progress. The good news that people are willing to do that, but this shutdowns interesting avenues of product development like artificial intelligence and helping players learn rules.

So, back to **Why**. The Lamancha project aims at greatly simplifying the engineering problem such that tools could eventually be built to enable children to design complex games. Now, board games are the product in question because a burning passion to implement Battle Star Galactica exists. **But!** the problem extends beyond that domain and into the entire space of web programming, and the real goal of Lamancha is to build a new web around security, ease of development, decentralized authority with multi-level compute. Now, that's a whole lot of craziness which may not make sense, but now it is time to describe the **What** and transition into **How.**

What

STEP 1: NEW “BROWSER”

So, what is this madness. Let's start with the web model and then see what we would need to build out a “new web”. Well, the most impactful element of the web is a standardized client called the browser. With a browser, people can put in an address and then get a complete experience that is new and fresh. This is a great model to build upon, and this is broken down to the right. This then sets a stage for good questions like “what does the client connect to?”

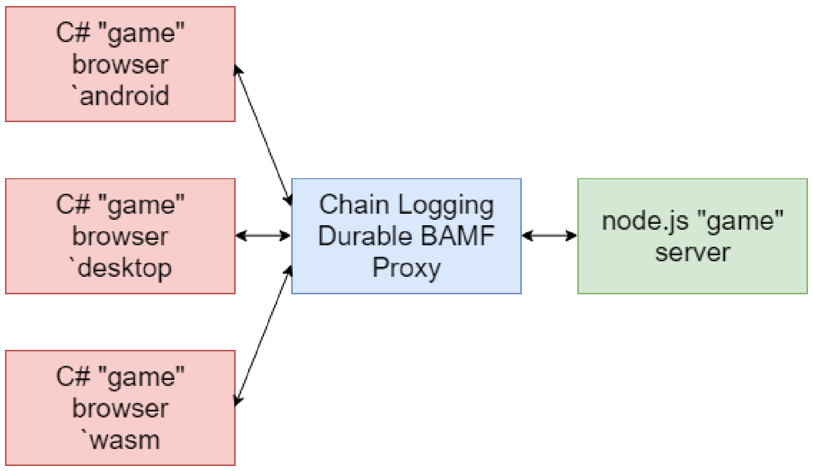


This means the basic user interface looks like a 90s era internet browser with a URL bar, some buttons, and a window that renders the game “session”.

STEP 2: THINKY THOUGHTS ON SERVERS

With the browser model in mind, we have to figure out what went wrong with the way we build things today that make these efforts hard.

First, the networking stack is primarily request response due to its simplicity and perceived goodness. HTTP provides a nice world for most people, but Erlang message passing tends to be better. However, this breaks how we think and communicate as people.


C# "game" browser `android
C# "game" browser `desktop
C# "game" browser `wasm
Chain Logging Durable BAMF Proxy
node.js "game" server

HTTP provides a nice transaction style which makes simple things simple, but people engage in conversations and conversations are complex. Good products mirror human behavior. I believe that the right thing to do is provide a simpler abstraction on top of Sockets, so we need a streaming API between the client and server.

Secondly, the database makes things hard. As someone that enjoys writing databases, I have found that the split between the user interface, application logic, and the database to be problematic. A big part of the problem is that databases tend to take on the role of “doing the hard stuff”, but using a database to make a great experience is actually really hard. The hard problem of a database is really just “don't lose shit” (However, it doesn't protect against user error, so even then it isn't that good...). **Now!** I believe that we have to rethink the core essence of the database problem being one primarily of persistence and durability. Maybe, just maybe, we don't need a traditional database or any webscale nonsense NOSQL thing. Maybe, just maybe, we just need some simpler.

STEP 3: JUMP BACK TO CLIENT

A good client from back in the day with just HTML and was dumb as nails. The old days had documents and forms, and it was good enough for business. Fast forward to today, the browser business of today is so complex and Chrome has over 6.7 million lines of code. Effectively, it now has entrenched players with Google and Mozilla as the main players forming little room for competition. With Microsoft jumping ship, the open web is basically dead. Building a new browser based on existing specifications is a lost cause.

With a notion of rethinking the database and networking stack, let's think about what properties we would want in an idealized client to challenge the status quo. The client should

- be super dumb
- ultra efficient for smooth rendering during high volume and good battery performance during low performance
- have adjustable behavior such that the experience automatically adjust to customer's bias between quality, performance, battery, and network usage.
- be immune to many security problems that make launching new web properties expensive today (cross site)
- be simple to build without a over-burdened specification; it should be so easy to build that a college student could build a reasonable client in a matter of two months.
- provide accessibility transformations for blindness, deafness, visual impairments

The solution to achieving the above requires pulling back and understanding what the actual goal is with the machine. It's about providing a great experience. Achieving this requires seeing that you need a visual surface, maybe an audio channel, and a way of collecting signals. It is worth also considering how to provide taste, feel, and smell; these are considered as exercises for the read. So, no tall order, we just need:

- a way of showing any kind of information
- a way of rendering audio
- a way of getting input

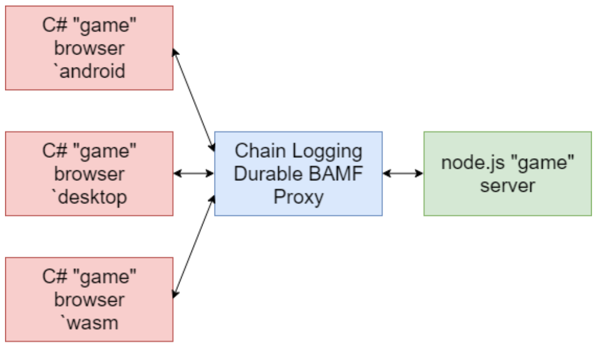
The extreme version of this is actually Google' Stadia which is extreme because the first two goals are achieved with streaming video, and the third goal is simply streaming all keyboard and mouse events to the server. However, Stadia fails many of the state client goals for us like battery and network, but this points to a nice escape hatch: provide video streams.

Due to the escape hatch, the plan is to iterate on the goals to achieve great experiences. This is the Lamancha Client called “Splashy”.

STEP 4: ROUTE TO A SERVER

With a super dumb client, the next step is to solve the distributed system, durability, and persistence problem. Since load balancers are important for scale and safety, we want this to be relative simple. However, the stream protocol will be a first class citizen and it will be built from the ground up. Because it is stream, we have some useful concepts that work well within data centers. For instance, we can use DNS to provide an initial IP, but then have the client and server negotiate the right link. Ideally, the actual server connected to the person's device should be geographically close.

Since load balancing is a fair simple load and primarily networking only, we will raise the bar and turn the load balancer into a symmetric chain logger for durability. This is where we can go into crazy town with distributed system ideas, but the core idea is that the state between the device's client and server is durable to the number of links between them. The server is also part of the chain.


C# "game" browser `android
C# "game" browser `desktop
C# "game" browser `wasm
Chain Logging Durable BAMF Proxy
node.js "game" server

For failure modes, it would require both the proxy and the server to simultaneously fail for data loss to happen. With a loss of a proxy, the client and server could re-establish a chain and regain durability. However, it is also possible to introduce additional proxies between the client and server to achieve extra durability. Additionally, you can have a proxy tail the server to provide a warm back-up for failure modes.

This requires the programming model of the server to be transactional with a hybrid two phase commit. The core challenge is writing code such that it can “resume” a process restart. With techniques like Redux and making the server's state deterministic, this is actually not hard and becoming popular. It does require discipline, and it is worth designing a programming language to make this easy. However, the challenge at hand is to do it with node.js.

STEP 5: GREAT DEVELOPER EXPERIENCE

So, setting up the chain logger proxy is going to be a pain in the ass. HOWEVER, it must be the cast that a person could run the node.js game and then use it locally at home.