Lamancha's Octoprotocol: Symmetric Multi-Client Chain Replicated Durable Compute State

This document outlines Lamancha's “Octopus Protocol” called Octoprotocol. The story of the protocol is like this; you have between one to a thousand people connected to a shared server to play or observe a game. With potentially many players in an intense game, a server failure would be noticeable in some form. It is then considered a requirement that players will not notice a server failure during the experience beyond lag. We should only expect players to have to repeat actions if they were disconnected and experienced a crash.

It's worth noting that the observation of a game can scale beyond a thousand, but there is an implicit bottleneck to the people playing a game. In the case a larger scale game, this bottleneck then suggests some of kind of sharding requirement by either different tables or geographical sharding.

With the Lamancha Browser “Splashy”, players in a game will interact with the UI such that interactions will generate events. (and maybe change local state which are a specialized form of an event). These events needs to flow from the device to the infrastructure. The events coming from devices will be of the form:

```
struct DeviceEvent {
  int DeviceSequencer;
  string Name;
  string Payload;
}
```

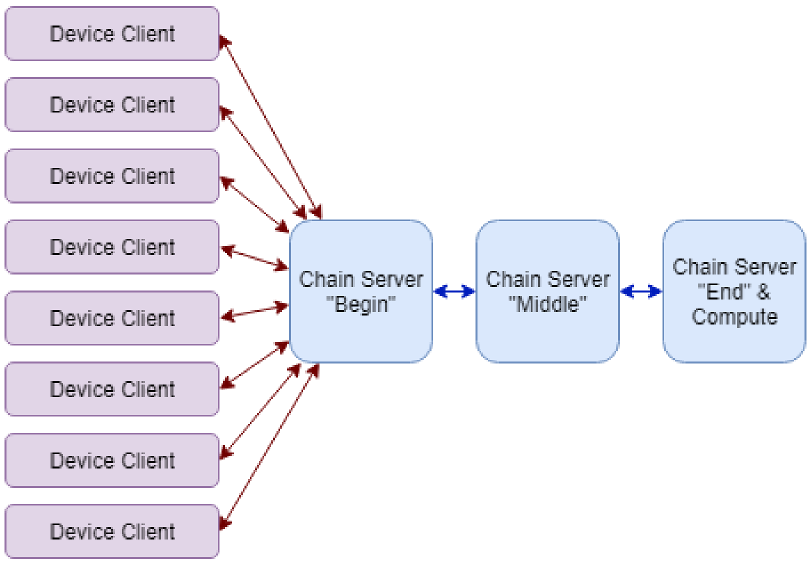
These **DeviceEvent**s are stored locally in the device in a state machine with a schema of:

```
struct DeviceStateMachine {
  OrderedMap<int, DeviceEvent> Events;
  int NextDeviceSequencer;
  JSONTree State;
}
```

This state machine, if persisted to disk, must be grouped together atomically. That is, the **Events**, **NextDeviceSequencer**, and **State** must be atomically stored together. During a crash or device change, it is entirely appropriate for the state machine to reset to an initial or prior state. This will imply that the user will have to repeat an action which is expected in a crash scenario.

A **DeviceEvent** happens when the UI kicks out a (**Name**, **Payload**) pair. The **DeviceSequencer** then comes from the state machine via **NextDeviceSequencer**, and **NextDeviceSequencer** is incremented atomically with inserting the new **DeviceEvent** into the Events map.

The namesake of the protocol is found in that multiple devices connect to the same server, and these converge into a joined state machine. Let's visualize this.



It looks like an octopus. The body of the octopus is series of chain links that are each are a replica of the state. Each link in the chain on the infrastructure then has a copy of each **DeviceEvent** embedded in an **Event** with a newly created per chain **ChainSequencer** and a **Tombstone** flag initialized to false.

```
struct Event {
  string Device;
  int DeviceSequencer;
  int ChainSequencer;
  string Name;
  string Payload;
  bool Tombstone;
}
```

And there is a state machine per replica link to keep track of events.

```
struct ChainStateMachine {
  // state from clients
  OrderedMap<int, Event> Events;
  Map<string, int> DeviceTrims;
  int NextChainSequencer = 1;

  // state from downstream
  int Trim;
  JSONTree State;

  // chain peers
  Proxy<ChainStateMachine> UpstreamPeer;
  Proxy<ChainStateMachine> DownstreamPeer;
}

void IngestDeviceEvent(ChainStateMachine sm, string Device, DeviceEvent de) {
  foreach (Event e in Events.Values) {
    if (e.DeviceSequencer == de.DeviceSequencer && e.Device == Device) {
      return true;
    }
  }

  // we have already processed the event and memorized that action
  if (sm.DeviceTrims[Device] >= de.DeviceSequencer) {
    return false;
  }

  Event event = new Event();
  event.Device = Device;
  event.DeviceSequencer = de.DeviceSequencer;
  event.ChainSequencer = sm.NextChainSequencer;
  event.Tombstone = false;
  event.Name = de.Name;
  event.Payload = de.Payload;
  sm.Events.add(event.ChainSequencer, event);
  sm.NextChainSequencer++;

  if (DownstreamPeer) {
    DownstreamPeer->Ingest(event);
  }
  return true;
}

void Ingest(ChainStateMachine sm, ChainEvent event) {
  sm.Events.Add(event.ChainSequencer, event);
  sm.NextChainSequencer = event.ChainSequencer + 1;

  if (DownstreamPeer) {
    DownstreamPeer->Ingest(event);
  }
}
```

This state machine is then replicated multiple times across different failure domains in a linear fashion using chain replication. One of the replicas, such as the end, is then designated as the “compute note”. The core of the idea is that the computation is then limited to the current state machine in the node and the incoming events. Once computation finishes, then the required changes are propagated.

The computation then has the duty to drain the Events and update the state. This process will emit a StateChange up the chain.

```
struct StateChange {
  bool Replace;
  JSONTree Change;
  List<int> EventKill;
}
```

This event mirrors the consumption of Events and a JSON Merge style change. In the case that State change is too severe for a merge, then it can be simply be a **Replace**ment state tree.

The key idea is that the chain replication has the job of persisting both the aggregated events from all clients and the current state of the computation. Events flow from devices to compute through the chain, and state change flows up the chain from compute to the devices. In the event of a failure, the durability of the compute depends on how quickly the chain can be reconstructed. However, it should be clear that as long as at least one node remains then the experience is smooth beyond a the lag of failure detection.

Modes of Operations

WHAT HAPPENS WHEN A DEVICE CONNECTS

The device may have an empty state machine (device crash + data loss), a previously stored state machine (device crash), an accurate state machine (socket loss). For all these situations, the device client must perform a negotiation with the first link in the chain. When connecting, the client sends the a **BootstrapChainRequest** with a copy of the current sequencer and the events.

```
struct BootstrapChainRequest {
  int NextDeviceSequencer;
  OrderedMap<int, DeviceEvent> Events;
}
```

The client will be expecting a **BootstrapComplete** message to return from this. However, until the **BootstrapComplete** is returned, the client state machine is locked and all events must be queued. The server will get **BootstrapChainRequest** and return a **BootstrapComplete.**

```
struct BootstrapComplete {
  int NextDeviceSequencer;
  List<int> Kill;
  OrderedMap<int, DeviceEvent> NewEvents;
  JSONTree Change;
}

BootstrapComplete Bootstrap(ChainStateMachine sm, string Device, BootstrapChainRequest
  BootstrapComplete bc;
  bc.NextDeviceSequencer = request.NextDeviceSequencer;
  foreach (Event e in Events.Values) {
    if (e.Device == Device) {
      if (!request.Events.Contains(e.DeviceSequencer)) {
        // the server has detected an event that the device doesn't have.
        DeviceEvent nde = new DeviceEvent();
        nde.DeviceSequencer = e.DeviceSequencer;
        nde.Name = e.Name;
        nde.Payload = e.Payload;
        if (bc.NextDeviceSequencer < e.DeviceSequencer) {
          bc.NextDeviceSequencer = e.DeviceSequencer + 1;
        }
        bc.NewEvents.add(nde.DeviceSequencer, nde);
      }
    }
  }
  for (DeviceEvent de in request.Events.Values) {
    if (!Ingest(sm, Device, de)) {
      bc.Kill.add(de.DeviceSequencer);
    }
  }
  bc.State = sm.State;
  return bc;
}
```

When the clients gets a BootstrapComplete, it will

- merge the NewEvents into Events map
- remove all Events from the kill list
- assume the State

This process will:

- Recover Events and ensure no sequencer conflicts during an application crash or client side data loss.
- Recover Events that were sent while the socket was broken.

WHAT HAPPENS WHEN THE FIRST LINK BREAKS (CRASH OR DEPLOY)?

Assume the most brutal scenario and the process of the first link just straight up crashes. The next link in the chain gets promoted with the state that it has. This scenario mirrors the behavior of a socket disconnect between the devices and first node. The devices will catch the next node up.

WHAT HAPPENS WHEN A MIDDLE LINK BREAKS (CRASH OR DEPLOY)? TODO: CODE

When a downstream peer changes, the downstream peer will be bootstrapped by getting parent's state machine. A key idea here is the the server closest to the device is more authoritative for events, and server closest to compute is more authoritative for state.

```
void Ingest(ChainStateMachine mine, ChainStateMachine prior) {
  // foreach event in prior, pump into mine
  // as Trim applies, kill prior events and generate a StateChange
}
```

What happens when the end link breaks? (Crash, Deploy, Code Update) TODO: Code

The code itself is kind of stateless.

Now, with this in mind, there are a bunch of stories about how this will work which need to be addressed.

- [x] ~~What happens when a device connects~~
- [x] ~~Device's Application Crashed~~
- [x] ~~Device's Application Was Up, Connection was Lost~~
- [x] ~~How Devices Send Data to First Node in Chain~~
- [x] ~~How Nodes Communicate Down~~
- [ ] How State Flows Up
- [x] ~~**What Happens when First Node Fails and a Device Resets**~~
    - [x] ~~If the first node failed prior to updating the secondary node, then that device is fucked.~~

    - [x] ~~Otherwise, the device will recover everyt~~
- [x] ~~What Happens when a Middle Node Fails~~
- [ ] What Happens when Final Node Fails
- [ ] Why are events consumed potentially out of order

Prior Design Notes

```
struct Event {
 int DeviceSequencer;
 int ChainSequencer;
 string Name;
 string Payload;
 string StateChange;
 bool Tombstone;
}

struct StateChange {
  int Sequencer;
  bool is_delta;
  String change;
  List<int> Kill;
}

struct ChainStateMachine {
  // state from clients
  OrderedMap<int, Event> Events;
  int NextChainSequencer;

  // state from downstream
  int Trim;
  string State;
  OrderedMap<int, String> StateChanges;
  int NextStateSequencer;
  Proxy<ChainStateMachine> UpstreamPeer;
  Proxy<ChainStateMachine> DownstreamPeer;
}

OnEventFromDevice(ChainStateMachine sm, string Device, DeviceEvent event) {
  var found = FindEvents(
    sm,
    (e) => { e.Device==Device && e.DeviceSequencer ==event.Sequencer);
  if (found) {
    // ignore it, we already have it
    return;
  }

  // incorporate
  ClientEvent ce = new ClientEvent();
  ce.Device = Device;
  ce.DeviceSequencer = event.Sequencer;
  ce.ChainSequencer = NextChainSequencer;
  ce.Name= event.Name;
  ce.Payload = event.Payload;
  ce.StateSequencer = event.StateChange;

  sm.Events.Add(ce.ChainSequencer, ce);
  sm.NextStateSequencer ++;
  sm.NextChainSequencer ++;
  DownstreamPeer->Send(ce);
}

OnPeerConnected(ChainStateMachine sm) {
  // TODO: think about negotiation
  DownstreamPeer->Replicate(sm);
}

OnClientEventFromPeer(ChainStateMachine sm, ChainEvent ce) {
  if (ce.ChainSequencer < sm.Trim) {
    // return; we have already processed it
  }

  if (!sm.Events.Contains(ce.ChainSequencer)) {
    if (DownstreamPeer == null) {
      StateChange sc = new StateChange();
      sc.is_delta = true;
      sc.change = ce.StateChange;
      sc.Kill = [];
      sm.StateChanges.Add(sm.NextStateSequencer, ce.StateChange);
      sm.NextStateSequencer++;
      UpstreamPeer->Send(sc);
    }
    sm.Events.Add(ce.ChainSequencer, ce);
    if (sm.NextChainSequencer <= ce.ChainSequencer) {
      sm.NextChainSequencer = ce.ChainSequencer + 1;
    }
    if (DownstreamPeer != null) {
      DownstreamPeer->Send(ce);
    }
  }
}

OnStateChangeFromPeer(ChainStateMachine sm, StateChange sc) {
  if (sm.StateChanges.Contains(sc.Sequencer) {
    return;
  }
  sm.StateChanges.Add(sc.Sequencer, sc.StateChange);
  if (sm.NextStateSequencer <= sm.Seqeuncer) {
    sm.NextStateSequencer = sm.Seqeuncer + 1;
  }
  foreach (int kill in sc.Kill) {
    sm.Events[kill].Tombstone = true;
  }
  while (sm.Events[sm.Trim].Tombstone) {
    sm.Events.Remove(sm.Trim);
    sm.Trim++;
  }
  if (UpstreamPeer != null) {
    UpstreamPeer->Send(sc);
  } else {
    // Broker with devices to trim
    DownstreamPeer->TrimStateDeltas(...)
  }
}

struct ChainStateMachine {
  // state from clients
  OrderedMap<int, Event> Events;
  int NextChainSequencer;

  // state from downstream
  int Trim;
  string State;
  OrderedMap<int, String> StateChanges;
  int NextStateSequencer;
  Proxy<ChainStateMachine> UpstreamPeer;
  Proxy<ChainStateMachine> DownstreamPeer;
}
```