Stellar Consensus Protocol Implementation

Jeremy Rubin
jr@mit.edu

John Holliman
holliman@mit.edu

May 6, 2015

1 Introduction

Cryptocurrencies are systems which facilitate the execution of payments, contracts, and other types of transactions over the internet in secure and robust ways. There is a rich early history of cryptocurrencies, ranging from Digi-Cash[**?**] to Peppercoin[**?**]. These systems require trusted third parties, however, and ultimately were not successful. Bitcoin solved the problem of reliance on trusted third parties through its Byzantine Fault Tolerant consensus mechanism – a proof of work based blockchain.

Proof of work is a very costly and energy inefficient means to reaching consensus as it requires solving dificult problems which wastes computational resources. Furthermore, transactions take a long time to confirm and security is in question if a single group at any point accounts for more than 50% of the computing power.

The Stellar Consensus Protocol[**?**] is a design of a Federated Byzantine Agreement System, or a consensus protocol which relies on Federated Voting for security rather than proof-of-work. This is a much more efficient means of reaching consensus compared to proof-of-work, although the lower expense might mean that the incentives to keep it are lower.

2 Stellar Consensus Protocol

The Stellar Consensus Protocol is a four-phase Paxos-like consensus protocol. Nodes exchange a series of ballots to vote to confirm, then accept values. The protocol can be considered in a single instance (i.e., determines one value) context, but it can also be easily extended to multiple instance log replication. The Stellar Consensus protocol is rather complex. As a result, a

large part of the effort for this project was in understanding the subtlies of how it works. Here we give a best-effort summary on how it works, but for more understanding the full paper[**?**] is necessary. For simplicity, we will talk about how it works in the context of a single slot.

Stellar Consensus Protocol hinges on a property called quorum intersection. Quorum intersection is the notion that all properly behaving nodes have connections. This property is, more formally, the notion that you can remove some set of misbehaving nodes and nodes which are dependent on them, and if quorum intersection holds the nodes will not be partitioned. TODO....

3 Design Overview

We implemented the Stellar Consensus Protocol from a clean slate, not referencing the existing implementation.

3.1 C++

We decided to implement our project in C++. Although neither of us knew C++ well a priori, we considered it an important goal for out project given that almost all cryptocurrency/consensus systems code is written in C++. There were several stumbling blocks to get over, but we are both know proficient in C++, which we consider to be a very large reward of this project as we are now more comfortable contributing code to existing implementations.

3.2 Implementation Details

Network We implemented a mock RPC interface. The reason we did this was so that we could extend it to easily show certain byzantine conditions, such as packet loss and reordering. We didn't implement a networked RPC interface although our RPC abstract base class could be subclassed to communicate over network.

Even though our network was not real, key functionality was not mocked out. For instance, we serialize and deserialize all messages to and from JSON using the Cereal library. Node threads only communicate with these mock network queues, there is no direct memory sharing.

Node Each node maintains a set of slots. Upon receipt of a message, the node looks up the slot, creating it if it doesn't exist. The node then processes

the message in the context of the slot. Slots do not have an effect on one another.

Quorum The node maintains a quorum set of size n. It also chooses a parameter $m < n$ of which any set of $m + 1$ nodes constitutes a quorum slice. Quorum selection is an open problem in Stellar Consensus Protocol, it is unclear how to get nodes to select quorums such that quorum intersection holds.

3.3 Proof Of Timeout

One open problem in Stellar Consensus is determining the mechanism by which Nodes are allowed to propose arguments for the log. Stellar consensus can be extremely inefficient in terms of number of messages sent, especially with multiple proposers.

One solution which we added was adding a proof-of-work packet filter for ballots. By requesting a solution to: $hash(value||slot||nonce) < bound$ with every ballot, two different values will take different amounts of time to find solutions to which serves as a randomized timeout which is valid in a byzantine context. This can help the network make progress. The bound can be tuned based on network activity. This also helps achieve anti-spam properties as well.

Unlike in bitcoin, this proof-of-work is not directly incentivized, therefore hopefully less prone to the development of ASIC hardware to compute them. The only incentive is to spend funds more quickly.

3.4 Applications

We implemented two simple key value stores on top of the Stellar Consensus Protocol: Asteroid and Comet.

Asteroid

The Asteroid KV store has the following semantics:

Get(Key) returns a pair of Version and Value. Gets are not put in the log, they are served at whatever slot the server has read up to. Entries are individually versioned in any case.

Put(Key, Value) reads the log like Get, and puts in an entry of Value under Key with Version + 1. Versions less than what is stored in the log will be ignored.

In a more full implementation, Key can be a public key, and value can be a signed message by the key. This allows for the StellarKV to be used a configuration updating consensus protocol. Other invariants could be added per key perhaps as well. This could also serve as the start of storing a balance as well for a transaction system. Keys could also be modified to be hierarchal for better name spacing, e.g. $PublicKey||Pictures||10$.

Versioning allows for a performant way of de-duplicating entries which get in the log at multiple slots. If a Version is younger than what is in the log, the updates are not applied. Of course, a user should keep track of the highest version they have sent and be sure that a server reflects that change eventually.

Comet

The Comet KV store was designed to mirror the 6.824 labs. Comet consists of comet/client.cpp, comet/common.hpp, and comet/server.cpp.

Comet replicas stay identical unless some lag behind, in which case they are able to catch back up.

The Comet clients tries different replicas until one responds. Calls to Client.Get() and Client.Put() appear to have affected all replicas in the same order and have at-most-once semantics.

3.5 Consensus Overview