Oort: Stellar Consensus Protocol Implementation

Jeremy Rubin
jr@mit.edu

John Holliman
holliman@mit.edu

May 8, 2015

https://github.com/OortConsensus/SCP

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[2] to Peppercoin[7]. These systems require trusted third parties, however, and ultimately were not successful. Bitcoin[6] solved the problem of reliance on trusted third parties through its Byzantine Fault Tolerant consensus mechanism — a proof of work[1] based blockchain.

Proof of work is a very costly and energy inefficient means to reaching consensus. It requires solving difficult problems which wastes computational resources. Furthermore, transactions take a long time to confirm and the consistency of the transactional history is vulnerable if a single group at any point accounts for more than 50% of the computing power.

The Stellar Consensus Protocol[5] is a design of a Federated Byzantine Agreement System, a consensus protocol which relies on reputation-based majority agreement for security rather than proof-of-work. If Stellar's assumptions hold it is a much more efficient means of reaching consensus than Bitcoin.

For our 6.824 MIT Distributed Systems class final project, we wrote our own implementation of the Stellar Consensus Protocol, working from the whitepaper[5] and not the existing implementation. Our implementation of the Stellar protocol is modularized. As a result, it was simple for us to build two distinct key-value store applications on top of it.

2 Stellar Consensus Protocol

The Stellar Consensus Protocol (SCP) solves the problem of agreement in the presence of Byzantine actors without resorting to expensive means such as proof-of-work. The number of misbehaved nodes SCP can handle varies with respect to certain parameters and peer selection.

SCP is a four-phase Paxos-like[4] consensus protocol. Nodes exchange a series of ballots to vote to confirm, then accept values. Much like Paxos, the protocol can be considered in a single instance (i.e., determines one value) context to prove correctness, but it can also be easily extended to multiple instance log replication. SCP is rather complex. As a result, a large part of the effort for this project was in understanding the subtleties of its operation.

SCP hinges on a property called quorum intersection. Quorum intersection is the notion that all properly behaving nodes have strong enough network topology such that they are able to reach consensus. More formally, the property that you can remove some set of misbehaving nodes and nodes which are dependent on them, and if quorum intersection holds, the network topology is strong.

The entire protocol is involved, and in the interest of space we will refrain from giving a more complete summary.

Two open problems in Stellar are the mechanism by which quorums are chosen (peer selection) and how new arguments may be proposed such that contention is low (i.e. avoid dueling proposers).

3 Design Overview

In this section, we outline our choice of programming language, the key components of our implementation, and our solution to the issue of when to propose. 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 a requirement for our 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 now 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.

It will be useful in our future research to understand:

1. Memory management best practices
2. Templating
3. Common gotchas (e.g. forward declarations)

3.2 Implementation Details

Network We implemented a mock RPC interface. The reason we did this was so that we could extend it to be able to easily simulate 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[3]. Furthermore, there is no direct memory sharing nodes. Nodes only communicate with mock non-blocking network interfaces.

Node As with many distributed systems, the basis of SCP is log replication. Each node maintains its own copy of the log. Upon receipt of a message, the receiving node looks up the appropriate slot in the log, 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. Each is a standalone Stellar consensus instance. As such, applications implemented on top of our layer may need a hole filling protocol. Nodes also serve views of externalized slots.

Quorum Nodes maintain a quorum set of size n. They also chooses a parameter $m \leq n$ of which any set of m nodes constitutes a quorum slice. If any quorum slice agrees on a given slot that slot will be externalized. In general, a node can use other criteria to select their quorum slices. 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. In general, large quorum slices are safer but reduce system liveness. In the wild, one could imagine selecting quorum sets on the basis of reputation (e.g. MIT over our personal node). It's unclear how an unestablished node can participate in consensus.

Consensus Overview Each node maintains the state detailed in Stellar paper [5, p. 27-28]. Nodes exchange two types of messages: PrepareMessages and FinishMessages. When a node receives the proper sequence of messages with respect to its quorum slices it transitions state. The best entry point into our code to observe the underlying consensus mechanism is src/scp/slot.cpp:Slot::handle.

See the code source map in the appendix for more details.

3.3 Byzantine randomized timeouts

Another 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 dueling proposers.

To combat this issue, we added 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, and therefore hopefully less prone to the development of ASIC hardware to compute them. The only incentive is to be able to transact more quickly. Given the high throughput of Stellar, however, someone is unlikely to need to send messages that much 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 try 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.

4 Conclusion

After implementing the Stellar Consensus Protocol, one of our biggest concerns is how hard it is to understand. Even after implementing SCP, we still feel our understanding is rudimentary. History has been shown to favor simpler solutions over more complicated but possibly superior ones.

References

[1] A. Back. Hashcash - A Denial of Service Counter-Measure, 2002.

[2] D. Chaum. Blind signatures for untraceable payments. 1998.

[3] S. Grant. cereal - A C++ library for serialization.

[4] L. Lamport. Paxos made simple, 2001.

[5] D. Mazieres. The stellar consensus protocol: A federated model for internet-level consensus.

[6] S. Nakamoto. Bitcoin: A peer-to-peer electronic cash system, 2008.

[7] R. Rivest and S. Micali. Peppercoin micropayments. http://people.csail.mit.edu/rivest/pubs/Riv04c.slides.slides.pdf, 2003.

5 Appendix

Source map

```
src
├─ Makefile
├─ apps
│  └─ kv-stores
│     ├─ asteroid
│     │  ├─ main.cpp
│     │  ├─ stellarkv.cpp
│     │  └─ stellarkv.hpp
│     └─ comet
│        ├─ client.cpp
│        ├─ client.hpp
│        ├─ main.cpp
│        ├─ misc.hpp
│        ├─ server.cpp
│        └─ server.hpp
├─ rpc-layer -- network interfaces
│  ├─ RPC.hpp -- abstract interface
│  ├─ fakeRPC.cpp -- mock cross-thread implementation
│  └─ fakeRPC.hpp
├─ scp -- consensus core
│  ├─ ballot.hpp
│  ├─ message.cpp
│  ├─ message.hpp
│  ├─ node.cpp -- maintains a log of slots and responds to user-level applications
│  ├─ node.hpp
│  ├─ quorum.hpp -- set of nodes being tracked by a node
│  ├─ slot.cpp -- single instance of consensus
│  └─ slot.hpp
└─ util
   ├─ common.hpp -- typedefs (eg NodeID)
   ├─ hash.hpp -- proof-of-work code
   └─ queue.hpp -- thread-safe queue for mock network
```