National Kapodistrian University of Athens

Graduate Program of Logic, Algorithms and Computations
Department of Mathematics

Federated Consensus Protocols

Galenianou Myrto

201401

Thesis advisor: Aggelos Kiayias

Submitted in part fulfilment of the requirements for
the Master Degree in Theoretical Computer Science
at the University of Athens, 2017

Η παρούσα Διπλωματική Εργασία
εκπονήθηκε στα πλαίσια των σπουδών
για την απόκτηση του
Μεταπτυχιακού Διπλώματος Ειδίκευσης
στη
Λογική και Θεωρία Αλγορίθμων και Υπολογισμού
που απονέμει το
Τμήμα Μαθηματικών
του
Εθνικού και Καποδιστριακού Πανεπιστημίου Αθηνών

Εγκρίθηκε την 28/7/2017 **από Εξεταστική Επιτροπή**
αποτελούμενη από τους:

Ονοματεπώνυμο	**Βαθμίδα**	**Υπογραφή**
1. Κιαγιας Αγγελος	Αν. Καθηγητής	
2. Παγουρτζης Αρ.	Αν. Καθηγητής	
3. Φωτακης Δημ.	Επικ. Καθηγητής	

Abstract

This dissertation studies consensus protocols and specifically Raft and the Stellar Consensus protocol. We first define the execution model under which we study the protocols as well as the notion of a robust transaction ledger that we want the protocols to maintain and its properties. We proceed by presenting Raft as concrete algorithm and we prove that indeed Raft maintains a robust transaction ledger. We then move to the Stellar Consensus protocol and analyse federated voting, Stellar's mean to reach consensus. Subsequently, we present the two protocols that constitute the Stellar Consensus protocol, the Nomination and Ballot protocol, as concrete algorithms and further explore their properties. Finally, we show that the Ballot protocol has both persistence and liveness, the two necessary properties a protocol need to have to maintain a robust transaction ledger.

Περίληψη

Αυτή η διπλωματική εργασία μελετά πρωτόκολλα συναίνεσης και συγκεκριμένα το πρωτόκολλο Raft και το πρωτόκολλο Stellar Consensus. Αρχικά ορίζουμε το μοντέλο εκτέλεσης υπό το οποίο μελετάμε τα πρωτόκολλα όπως και την έννοια του ισχυρού κατάστιχου συναλλαγών και τις ιδιότητές του. Στη συνέχεια παρουσιάζουμε το πρωτόκολλο Raft σε αλγοριθμική μορφή και δείχνουμε ότι πράγματι το Raft υλοποιεί ένα ισχυρό κατάστιχο συναλλαγών. Συνεχίζουμε με το πρωτόκολλο Stellar Consensus και αναλύουμε την ομόσπονδη ψηφοφορία, το βασικό μέσο που χρησιμοποιεί το Stellar Consensus για να πετύχει συνέναιση. Έπειτα παρουσιάζουμε τα δύο πρωτόκολλα που αποτελούν το Stellar Consensus, το πρωτόκολλο Nomination και το πρωτόκολλο Ballot, σε αλγοριθμική μορφή. Τέλος δείχνουμε ότι το πρωτόκολλο Ballot έχει και τις δύο απαραίτητες ιδιότητες που οφείλει να έχει ένα πρωτόκολλο για να διατηρεί ένα ισχυρό κατάστιχο συναλλαγών.

Acknowledgements

I would like to express my appreciation to my supervisor, A.Kiayias, for his constant help, encouragement and guidance. Moreover, I would like to express my gratitude to my colleagues, Zeta Avarikioti and Isidoros Tziotis for their help and support through out the preparation of this master thesis. Our years in this master program I hope will be the beginning of a long friendship. Lastly, I would like to express my gratitude to my family and especially my sister. Without her support this thesis would never have been completed.

A path is made by walking on it

Zhuang Zhou, chinese philosopher.

Contents

1 Introduction **2**

1.1 The problem . . . 2

1.2 Previous work . . . 3

1.3 Contirbutions and thesis organisation . . . 4

2 Model **5**

2.1 Robust Transaction Ledger . . . 5

2.2 Execution Model . . . 6

3 Raft Consensus Protocol **7**

3.1 Leader election . . . 8

3.2 Log replication . . . 9

3.3 Persistence and Liveness . . . 11

4 Stellar Consensus Protocol **13**

4.1 Federated Voting . . . 15

4.2 Nomination Protocol . . . 16

4.3 Ballot Protocol . . . 19

4.4 Persistence and Liveness for the SCP . . . 24

4.4.1 Persistence for the Ballot protocol . . . 25

4.4.2 Liveness for the Nomination and the Ballot protocol . . . 26

5 Conclusion **30**

5.1 Summing up . . . 30

5.2 Future work . . . 31

Chapter 1

Introduction

1.1 The problem

In everyday life people with different backgrounds and opinions that work for a common goal are asked to make decisions jointly. For this to be achieved members of the group need first to meet in the same place, then all opinions need to be expressed and finally people turn to a previously agreed voting procedure to make a final decision. Anyone who has found himself in such a situation can imagine the difficulties and problems that arise before that final decision is made. Computer systems must also deal with a version of the same problem in order to communicate and agree on different tasks.

In distributed computing, the consensus problem is one of the central topics which has attracted intensively the research community for many years. The consensus problem is the problem of getting a set of nodes in a distributed system to agree on a value. A node which can have arbitrary behaviour is called byzantine. This includes not sending any messages at all, sending different and wrong messages to different neighbours, collusion, or lying about the input value. A consensus protocol that tolerates byzantine failures must be resilient to every possible error that can occur.

Consensus allows a computer network to communicate and work as one group that can survive the failures of some of its members. All computers -or nodes- in the network follow a protocol that guides them to agree on a task or value. Nodes often need to elect a leader to coordinate their actions, since consistency is an important property the protocol they are following needs to maintain. Failures can include nodes coming and going during normal operations, unresponsive nodes but also malicious ones. Nodes need to react quickly to changes in a well defined manner in order for the system to continue to operate properly. Distributed consensus can help with all these challenges.

Distributed consensus is also a computer science term often associated with cryptocurrencies. A consensus algorithm ensures that the next block in a blockchain is the one and only version of the truth, and it keeps malicious entities from successfully forking the chain. A blockchain is a public ledger that records transactions and it is distributed to all nodes in the network. In the blockchain transactions are permanently recorded in blocks and blocks are linked and from the beginning of the chain to the most current block giving the public

ledger the name blockchain. The blockchain acts as a single source of truth, since the history of all exchanges that take place between the peers in the network is recorded there. Bitcoin [6] uses *proof of work* to maintain its blockchain, while other digital currencies may use *proof of stake* or other ways to maintain their blockchain.

1.2 Previous work

There are two main timing models under which we study consensus. The synchronous and the asynchronous model. In the synchronous model, all message exchanges happen in rounds. In one round a node may send all the messages it requires while receiving all messages from other nodes. No message from one round may influence any messages sent in the same round. In the asynchronous model, algorithms are event based A message sent from one node to another will arrive in a finite but unbounded time. There is no deterministic algorithm which always achieves consensus in the asynchronous model. This result, known as the *FLP impossibility proof* [13] does not state that consensus can never be reached, but that under the model's assumptions, no algorithm can always reach consensus in bounded time.

One of the most famous distributed consensus protocols is Paxos [2]. Although no deterministic fault tolerant consensus protocol can guarantee progress in an asynchronous network, Paxos guarantees *safety* and the conditions that could prevent it from making progress are considered difficult to happen. However, Paxos is a protocol with significant difficulty to understand and moreover with considerable difficulty in implementation.
Another well-known consensus algorithm is Viewstamped Replication [10]. Viewstamped Replication was created around the same time as Paxos and it is a replication protocol rather than a consensus protocol. It uses consensus as part of supporting a replicated state machine.
Solving the problem of distributed consensus was one of the reasons that attracted researchers to Bitcoin and enabled the explosion in the number of "alt coins" (alternative cryptocurrencies). Nodes in the Bitcoin network try to solve a cryptographic problem, where the probability of finding the solution is proportional to the computational effort, and the node that solves the problem has their version of the block of transactions added to the blockchain and accepted by all of the other nodes. The proof of work scheme can be used to solve Byzantine Agreement. In their paper *The Bitcoin Backbone Protocol: Analysis and Applications* [7], Garray, Kiayias and Leonardos showed that the Backbone protocol satisfies the Byzantine Agreement properties, assuming that the adversary's hashing power is less than $1/3$, with an error that decreases exponentially in the length of the chain.
In the last few years, the byzantine fault tolerance theory has become more popular, due to the success of the blockchain technology of Bitcoin and the protocols behind the cryptocurrencies, like CryptoNote [12] in Ethereum. Every year more and more people are trying to explore and expand the world of cryptocurrencies and as a result, new protocols and ideas often arise and open new discussions. An example to that is the Tendermint [11] protocol. Tendermint consists of a blockchain consensus engine (Tendermint core) and a generic application interface and solves Byzantine consensus using proof of stake.

1.3 Contirbutions and thesis organisation

Motivated by the analysis of the Bitcoin protocol in*The Bitcoin Backbone Protocol: Analysis and Applications*, we express both Raft and the Stellar Consensus Protocol in the form of an algorithm and subsequently we examine them under the model presented in chapter 2, a model greatly inspired by the paper *Ouroboros: A Provably Secure Proof-of-Stake Blockchain Protocol* [9] as well. Specifically, in chapter 2, we define the notion of *robust transaction ledger* and the properties a protocol must have to implement one, as well as the execution model under which we study the protocols.

We consider the protocols as federated consensus protocols, since in both cases the nodes running the protocol in order to reach consensus they need to join together, in a way dictated by the protocol and form a unit, within which consensus is reached. Chapter 3 is devoted to the first consensus algorithm we study *The Raft consensus algorithm.* Raft was first introduced in 2013 by *Diego Ongaro and John Ousterhout* in an effort to create an understandable consensus algorithm, that would allow more accurate implementations and it has many similarities to Oki and Liskov's Viewstamped Replication. Under the model presented in chapter 2, we show that Raft implements a robust transaction ledger.

The second consensus algorithm we study, *The Stellar consensus protocol (SCP)* is presented and analysed in chapter 4. SCP is different to previous consensus algorithms, since the main tool used to reach consensus is *federated voting*. Federated voting gives the protocol the advantage of creating a network open to anyone, in contrast for example to proof of work where a miner needs enough computational power to keep up with the rest of the network. Since not everyone can have the necessary equipment to be a miner, not everyone can participate in ensuring the integrity of the system.

The core of SCP is a consensus algorithm that consists of two protocols, the Nomination and the Ballot protocol. Both the Nomination and the Ballot protocol implement federated voting. In short, to agree on a set values, nodes executing the Nomination protocol must vote, accept and confirm statements concerning those values. Subsequently, the set of values is combined (deterministically) to a single value and a ballot consisting of a counter and the combined value is created. In the Ballot protocol to agree on a ballot, nodes must accept and confirm statements concerning that ballot.

Chapter 2

Model

In order to describe the protocols better and argue about their properties, we first need to define the model under which they are examined. Specifically, since the protocols main job is to maintain a public ledger we need to define what a robust transaction ledger is. Furthermore, we need to specify the notion of protocol execution.

2.1 Robust Transaction Ledger

The protocols we study maintain a ledger of transactions. The ledger is divided in slots, indexed by an integer $i \in \{1, 2, ...\}$ that increases monotonically overtime. Every slot stores a block of transactions, those that the nodes agreed on while executing the protocol for that specific slot.

Nodes are equipped with ”almost synchronized” clocks to help them during the execution of the protocols, with collectively assigning a block of transactions to the current slot. The clocks are implemented in manner that best fits the protocol.

The time window that corresponds to a slot is sufficient to ensure that all messages send by honest nodes at the beginning of the time window will be received by an other honest node by the end of that window. This means that even though time delays may occur, they will not exceed the slot's time window.

Robust Transaction Ledger A protocol Π implements a robust transaction ledger provided that the ledger maintained by Π is divided into blocks of transactions that determine the order in which those transactions were included in the ledger. A robust transaction ledger satisfies the following properties :

- **Persistence:** Once an honest node claims a transaction tx is *stable*, every other honest node will either report tx in the same position in the ledger or will not report any transaction conflicting tx as *stable*.
  A transaction is considered *stable* if and only if it is in a block that is more than k blocks deep in the ledger
- **Liveness:** If all honest nodes in the system attempt to include a certain transaction, then after the passing of time corresponding to l slots, all nodes, if queried and responding honestly, will report that transaction as stable.

2.2 Execution Model

We consider an environment Z that generates all the nodes that run a protocol Π and a controller (a control program) C. Together (Z, C) are responsible for the execution of the protocol.In the protocols we examine, nodes are aware of the number of nodes running the protocol.
Specifically, Z generates a set of nodes V that run protocol Π and an adversary A. All nodes have an input tape, denoted $Input()$ and Z is responsible for providing nodes' inputs in the round indicated by Π and also receives all the outputs. In our model nodes have access to two functionalities, a "Diffuse" and a "Key registration" functionality. Both of them are described below:

- **Diffuse functionality** To proceed with the protocol nodes need to exchange messages. They do that by using the diffuse functionality. The functionality keeps a $Receive()$ tape for every node and nodes have access to their communication tape in every round. The functionality starts with round 1 and when a node gets the next message in its communication tape, the functionality marks him finished for that round. That way nodes have access to their communication tape once in every round. When all nodes are finished, the functionality includes any mew messages sent by the nodes to the necessary $Receive()$ tapes and increments the number of round.

- **Key registration functionality** During the execution of the protocol, nodes need to know the public keys of the nodes they trust in order to communicate. To achieve that they enlist the help of the key registration functionality. The functionality takes as input n nodes $v_1, ..., v_n$, consults with the adversary A to mark the corrupted nodes and subsequently samples public and private key pairs for all the honest nodes. Public keys for the corrupted nodes are set by the adversary.
  Nodes can request their public and private key as well as the public keys of all or specific nodes and the functionality will provide the information.

In every round, Z activates each node one by one and then the adversary. The controller's role is to ensure that Z activates all the necessary nodes, with respect to the protocol, the adversary and the functionalities.
When A is activated, he can corrupt new nodes which he declares to C, so that the node will be considered corrupted after that point, provided that A was already given the necessary input by Z to use for the corrupt node. In all rounds, every time a corrupted node is activated by Z, the adversary A is activated instead. The adversary also has access to nodes messages and can try to confuse nodes by sending inconsistent messages and delaying message delivery. However, A can not change message contents. A message can be "pushed back" in the communication tape Δ times. After that, the delayed message will be delivered. Given that, the timing model for the protocols is semi-synchronous. Moreover, honest nodes are given enough time by Z to process all messages delivered to them in a round. The way the communication between nodes is described, simulates communication over the Internet, where messages can be delayed but will eventually be delivered and malicious parties can sent messages using the identity of someone else.

Chapter 3

Raft Consensus Protocol

Raft [4] is a consensus protocol, created as an alternative to Paxos, one that would cover certain weak spots in Paxos. Specifically, Paxos is a protocol with significant difficulty to understand and moreover with considerable difficulty in implementation. Raft is designed, with a primary goal to be understandable. That way, it provides a solid foundation to build practical systems.

Every node in Raft, during the execution of the protocol maintains a state, that of *a leader, a follower or a candidate*. To reach consensus, nodes first elect a leader and subsequently that leader gets complete responsibility for managing client requests and maintaining the replicated log. Leaders send periodic *heartbeats* (empty $AppendEntries()$ requests) as a way to maintain authority. If at some point the leader finds out that he is out of date he returns to follower state.

A node in follower state is passive, meaning that it simply responds to requests from leaders or candidates. Candidate state is used to elect a new leader.

The protocol is divided in rounds that increase monotonically. Every round begins with an election and the candidate that wins the election becomes the leader for that round. Nodes store the current round and they include it in every message they exchange. If a node finds out that his current round number is smaller than that of other nodes it updates the round number to a larger value. If a candidate or a leader learns that his round number is out of date it returns to follower state. Nodes ignore messages with round number smaller than their current one.

We can consider every round or leader term as a ”stand-alone” execution of the protocol and all the requests handled by that leader as a block of transactions that occupy one slot in the ledger.

Given all the above, we can separate Raft in two parts, that can will be examined separately in the following sections :

1. Leader election
2. Log replication

3.1 Leader election

Every node in Raft begins as a follower and remains a follower as long as he receives valid messages. If a long period of time passes with no communication, the node increments its round number and turns its state to candidate. He then sends a *RequestVote* to all other nodes. The time period that passes before a node turns to candidate state is called *election time-out* and it is different for every node.
The node will remain a candidate until one of the following happens :

- He wins the election, by receiving votes from a majority of nodes. Nodes vote at most for one candidate, the one whose *RequestVote* they receive first.
- Another node becomes leader. While waiting for the votes to become leader, the candidate may receive an *AppendEntries* message from another node, that claims to be leader. If the round number of that node is is at least as large a the candidate's round number, he recognizes the leader and returns to follower state.
- A period of time passes with no winner. If many nodes turn to candidate state at the same time, there is the possibility of a split vote. In that case, each candidate will time-out, increment it's round number and start a new election. To avoid split votes and resolve them quickly when they happen, Raft uses randomized election time-outs.

Election time-outs are chosen at random for every node from a fixed interval. That way, in most cases only one node will time-out, win the election and establish leadership. When a split vote happens, all candidates restart their randomized election time-out at the beginning of the new election. That reduces the likelihood of a new split vote.

The $LeaderElection()$ function, given bellow, is called at the beginning of the protocol, after the election time-outs are set for every node.
The $VoteCount()$ function called by the $LeaderElection$ does exactly what it's name defines. Specifically, the function returns the number of messages containing positive votes for the node requesting the vote.

We also define the predicate $leaderMsg(M) = True \Leftrightarrow$ the node running $LeadereElection()$ receives an $AppendEntries()$ request from another node, declaring that he is the leader.

Algorithm 1

1: **function** LEADERELECTION($state, r$)
2: $state \leftarrow candidate$
3: $r \leftarrow r + 1$
4: $electionTimer()$
5: $RequestVote()$
6: $M \leftarrow Receive()$
7: $votes \leftarrow VoteCount(M)$
8: **if** $votes \geq \frac{n}{2}$ **then**
9: $state \leftarrow leader$
10: **if** $leaderMsg(M)$ **then**
11: $state \leftarrow follower$
12: **if** $electionTimer()$ **then**
13: go to step 1

3.2 Log replication

When a node becomes leader, he starts answering client requests. Client requests contain commands that need to be executed or transactions that need to be included in the ledger. The leader adds the command to his log and then sends *AppendEntries* messages to all other nodes. Once the request is replicated to a majority of nodes, the request is considered **committed** and the leader can execute the command and return the result to the client. Leaders also periodically send empty *AppendEntries* requests (heartbeats) to maintain authority. The time that passes between sending two different *AppendEntries* messages has to be smaller than that of election time-outs to keep followers from starting a new election.

Every log entry contains the command, the round number and an index number identifying the entry's position in the log. At most one entry is created by the leader for a given index number and round and entries never change position in the log.

The leader has to keep track of the highest index he knows is committed and include it in the *AppendEntries* message he sends. When a follower finds out a log entry is committed, he applies it to its log. Once a log entry is committed, all previous entries are considered committed as well. Raft also ensures that when a new leader is elected, all previous committed entries are included in his log, by preventing a candidate from being elected in case his log isn't up to date. This is done by including information about the candidate's log in the $RequestVote$ message. The vote is only granted if the candidate's log is up to date. If a leader crashes before he commits an entry, then it is up to the new leader to finish the job. A leader cannot conclude that an entry from a previous round is committed only because the entry is replicated in a majority of nodes. However, once a new entry, from the current round is committed, all prior entries are indirectly committed, due to the Log Matching Property (explained below).

When comparing two logs, we examine the index and round number of the last entry. If the logs have different round numbers, the one with the larger round number is more up to date. If the logs have the same round number, then the longer one is considered more up to date.

The leader also includes the index and round number of the entry that comes exactly before the new entries. This works as a consistency check. A follower will refuse the new entries if it doesn't have an entry with that index and round number. This allows leader and follower logs to stay consistent during a normal execution of the protocol.

The leader keeps a *nextIndex* for every follower, the index of the next entry he will sent to that specific follower. If a follower's log becomes inconsistent, the *AppendEntries* request will fail, due to the consistency check. The leader needs to handle the inconsistencies. To do that the leader first finds the latest entry that his and the follower's log agree on, by decreasing the *nextIndex* for that follower and resending the *AppendEntries* request repeatedly until it succeeds. Once the point of agreement is found the leader deletes all other entries in the follower's log after that point and sends him his entries after that point.

Given all the above, we can deduce that there are two properties satisfied by the *Log replication* procedure:

1. If two entries in different logs have the same index and round number, then they store the same command
2. If two entries in different logs have the same index and round number, then the logs are identical in all preceding entries.

   We can use the consistency check as an inductive step. Initially, all logs are empty and as a result identical. The consistency check ensures that logs will remain identical while they are extended.

Log replication is the main job of the Raft protocol. Every node starts in follower state, sets the $electionTimer$ and calls the $LeaderElection()$ function. Once the leader is elected, the protocol continues with the log replication.

The protocol also calls an $AddEntries()$ function that once again does exactly what it's name defines. If the node is the leader the function adds the node's input, t to the log and if the node is a follower the function adds the commands from the $AppendEntries()$ requests the node has received.

Algorithm 2 Raft, run by node v, where $Input()$ is the input tape and $Receive()$ the communication tape

1: $state \leftarrow follower$
2: $r \leftarrow 1$
3: $t \leftarrow NULL$
4: $electionTimer()$
5: $LeaderElection(state, r)$
6: **while** $state \neg candidate$ **do**
7: **if** $state = follower$ **then**
8: $M \leftarrow Receive()$
9: $AddEntries(state, M, t)$
10: **if** $state = leader$ **then**
11: $t \leftarrow Input()$
12: $AddEntries(state, M, t)$
13: $AppendEntries()$
14: **if** $electionTimer$ **then**
15: $state \leftarrow candidate$

3.3 Persistence and Liveness

Raft ensures that the following properties are true at all times during the execution of the protocol:

- **Election Safety**: At most one leader can be elected in every round.
  As explained in section 3.1, in case of a split vote, nodes restart the $LeaderElection$ function, with a higher round number. This results in a ”barren” round, where no leader is elected and no transactions are included in the ledger.

- **Leader Append Only**: A leader never overwrites or deletes entries in its log, he only appends new entries. Even when inconsistencies occur, the leader will only overwrite entries in the follower's log.

- **Log Matching**: If two logs contain an entry with the same index round, then the logs are identical in all entries up through that index.
  The Log Matching property follows directly from the two properties satisfied by the Log Replication procedure.

- **Leader Completeness**: If a log entry is committed in a given round, then that entry will be present in the logs of the leaders for all higher-numbered rounds.
  We assume that the property does not hold. Let i be a the slot where leader v_i commits an entry tx that is not stored by a future leader. Let v_j, the leader of slot $j > i$ be the first leader that doesn't store tx.
  Since v_i committed tx there is a majority of nodes that copied tx and there is also a majority of nodes that voted for v_j to be elected. It follows that there is a node v that both voted for v_j and has tx in his log. The

entry tx could only have been copied in v's log before v_j was elected, otherwise v would have rejected the $AppendEntries$ containing tx. When v_j was elected his log had to be up to date. This leads to two contradictions:

1. v and v_j had the same last round number and v_j log was at least as long as v's, so it must have contained every entry in v's log. But we assumed that v_j's log does not contain tx
2. v_j last round number is larger than v's. But since v_j is the first leader whose log does not contain tx, any previous leader, that created v_j's last entry, would also have tx in his log. By the Log Matching property, v_j would also have tx in his log.

- **State Machine Safety**: If a node has applied an entry to its log at a given index, no other node will ever apply a different entry for the same index.
  When a node applies an entry to his log, his log must be identical to the leader's up until that entry and the entry must be committed. Let i be the smallest round in which any node applies an entry. By the Leader Completeness property all the leaders for of rounds higher than i will also have the same entry.

Using the above properties it is easy to conclude that Raft implements a *robust transaction ledger*.

Theorem 1. Raft implements a *robust transaction ledger*, as long as there is a majority of honest nodes.

Proof. We need to show that the protocol satisfies both persistence and liveness *Persistence* follows directly from the State Machine Safety property. If a node claims a transaction tx is *stable* that means that tx is committed and the node has applied tx to his log.
Furthermore, if there is a majority of honest nodes *liveness* also holds. If all honest nodes want to include tx in the ledger then the first time an honest node is elected leader tx will be committed. If that leader crashes before tx is committed then future leaders will try to finish his job. □

Chapter 4

Stellar Consensus Protocol

The Stellar Consensus Protocol [1] consists of two protocols, the Nomination and the Ballot Protocol. The Nomination Protocol produces candidate values for a slot and subsequently, using the Ballot Protocol nodes reach consensus. A set of nodes V runs the Stellar Consensus Protocol(SCP) to agree on updating a slot. The protocol implements federated voting.
To thoroughly explain how federated voting works, we first need to define the basic components of a Federated Byzantine system :

Definition 1. Federated Byzantine system (FBAS) : A Federated Byzantine Agreement System is a pair $< V, Q >$, where V is a set of nodes and $Q : V \rightarrow 2^{2^V}$ specifies the quorum slices for all $v \in V$.

In a FBAS malicious parties can join many times and even outnumber honest parties. Because of that, the usual majority quorums used by other consensus protocols don't work in a FBAS and a different way to define a quorum is needed.

Definition 2. Quorum Slice : A quorum slice q is a set of nodes sufficient to convince a node of a value or statement.

A node u can have more than one quorum slice and u belongs to all of its quorum slices.

Definition 3. Quorum : A quorum U is a set of nodes sufficient to reach agreement s.t
$\forall v \in U \; \exists q \in Q(v) : q \subseteq Q$.

We call *well-behaved* nodes, those who choose sensible quorum slices, follow the protocol and answer to all requests from other nodes and *ill-behaved* nodes those who behave arbitrarily. The goal of the protocol is to make sure that all well-behaved nodes externalize the same value, despite the existence of ill-behaved nodes. Well-behaved nodes that satisfy both safety and liveness are called correct. If a node is not correct, the node has failed.

For a FBAS to succeed, nodes should have the following properties :

1. A set of nodes in an FBAS $\langle V, Q \rangle$ has ***safety*** if every two of those nodes externalize the same value for the same slot.

2. A node in an FBAS $\langle V, Q \rangle$ has ***liveness*** if it can externalize new values without the participation of failed nodes.
   To avoid confusion with the Liveness property defined in chapter 2, we shall call this property *node-liveness*

All ill-behaved nodes are failed, but a well-behaved node can also be failed if he lacks either safety of liveness. Well-behaved nodes that lack safety are called divergent and those who lack node-liveness are called blocked.

Definition 4. A FBAS $\langle V, Q \rangle$ enjoys **Quorum Intersection** if and only if for any two quorums U_i, U_j in $\langle V, Q \rangle$, $U_i \cap U_j \neq \emptyset$.

A FBAS $\langle V, Q \rangle$ cannot guarantee *safety* in the absence of *Quorum Intersection*, since without that property the system can split and operate as two independent systems. In fact to guarantee safety, we need the set of well-behaved nodes to have *Quorum Intersection*.

Definition 5. If $\langle V, Q \rangle$ is a FBAS and $B \subseteq V$, then $\langle V, Q \rangle^B$ is the FBAS that arises if we *delete* B from $\langle V, Q \rangle$ and we compute the modified FBAS $\langle V \setminus B, Q^B \rangle$, where $Q^B(v) = \{q \setminus B \mid q \in Q(v)\}$

Definition 6. DSet (Dispensable Set) : Let $\langle V, Q \rangle$ be a FBAS and $B \subseteq V$. We call B a DSet,iff :

1. $\langle V, Q \rangle^B$ enjoys Quorum Intersection (quorum intersection despite B)

2. Either $V \setminus B$ is a quorum in $\langle V, Q \rangle$ or $B = V$ (quorum availability despite B)

Quorum intersection despite B ensures that after a set of nodes is deleted the system will not split, while quorum availability despite B ensures that there is at least one quorum in the system unaffected by the nodes we delete.

Definition 7. A node v in a FBAS $\langle V, Q \rangle$ is *intact* if there exists a DSet B that contains all the ill-behaved nodes and $v \notin B$. A node v is *befouled* iff it is not intact

A well-behaved node can be befouled if he is surrounded by enough ill-behaved nodes to affect his state or his progress is blocked. A Dset that contains all ill-behaved nodes may also include some well-behaved, but befouled nodes.

Theorem 2. In a FBAS with quorum intersection, the set of befouled nodes is a DSet.

Proof. Let B^* be the intersection of every Dset that contains all ill-behanved nodes. By the definition of intact nodes, a node v is intact if and only if $v \notin B^*$. It follows that B^* exactly the set of befouled nodes and since DSets are closed under intersection, B^* is also a DSet. □

4.1 Federated Voting

We can now explicate the steps of federated voting, the procedure that nodes in a FBAS $\langle V, Q \rangle$ use to reach consensus. We assume that nodes want to agree on a statement α. The first step nodes need to take is to vote for α. Then nodes will proceed with accepting α.

Definition 8. A node v **votes** for a value or a statement α iff :

1. α is valid and consistent with all the statements that v has already voted for
2. v has never voted for any statement contradicting α and promises never to do so in the future

Definition 9. A quorum U_α ratifies a value α iff every $u \in U_\alpha$ votes for α.
A node v ratifies α if $v \in U_\alpha$ and U_α ratifies α

A node will not be able to vote for α, if he has previously voted for a contradicting statement. He may however want to change his stance and accept α, if enough nodes have voted for α. There are two ways a node can accept a statement α.

Definition 10. blocking set Let $v \in V$ be a node in a FBAS $\langle V, Q \rangle$. A set $B \subseteq V$ is a blocking set of v iff $\forall q \in Q(v), q \cap B \neq \emptyset$

Definition 11. Accept A node v in a FBAS $\langle V, Q \rangle$ accepts a value or α iff it has never accepted a statement contradicting α and :

1. There exists a quorum U s.t $v \in U$ and $\forall u \in U$, u has either voted for or accepted α
2. Every member of a blocking set of v has accepted α

A blocking set of a node u can interfere with his node-liveness, if it contains ill-behaved nodes, since by definition that blocking set overlaps with all u's quorum slices. To guarantee node-liveness for a node u, that node needs to have a quorum slice comprising only of correct nodes.

Theorem 3. Let $B \subseteq V$ be a set of nodes in a FBAS $\langle V, Q \rangle$. $\langle V, Q \rangle$ enjoys quorum availability despite B iff B is not a blocking set for any $v \in V \setminus B$.

Proof. $\forall v \in V \setminus B, B$ is not a blocking set $\Leftrightarrow \forall v \in V \setminus B, \exists q \in Q(v)$ such that $q \subseteq V \setminus B$. By the definition of quorum availability despite B either $V \setminus B$ is a quorum or $V = B$ and the equivalence holds in both cases. □

Corollary 1. The DSet of befouled nodes is not a blocking set for any intact node v.

Proof. Let $\langle V, Q \rangle$ be an FBAS with quorum intersection and $B \subseteq V$ the set of befouled nodes. By Theorem 1, B is a DSet and by the definition of a DSet, $\langle V, Q \rangle$ enjoys quorum availability despite B
Then, by Theorem 2, B is not a blocking set for any $v \in V \setminus B$. But $V \setminus B$ is the set of all intact nodes. Therefore, the DSet of befouled nodes is not a blocking set for any intact node.

□

Corollary 2. In a FBAS $\langle V, Q\rangle$, with quorum intersection, every intact node v has at least one quorum slice comprising only of intact nodes.

Proof. Let $\langle V, Q\rangle$ be an FBAS with quorum intersection and $B \subseteq V$ the set of befouled nodes. Then B is not a blocking set of any intact node. But since B is a DSet, $\langle V, Q\rangle$ has quorum availability despite B
By the definition of a DSet either $B = V$ or $V \setminus B$ is a quorum in $\langle V, Q\rangle$.
If $B = V$, then there are no intact nodes in $\langle V, Q\rangle$ and the proposition holds.
If $V \setminus B$ is a quorum in $\langle V, Q\rangle$, then that quorum contains only intact nodes, and for every node v in $V \setminus B$ $\exists q \in Q(v) : q \subseteq Q$. That quorum slice can only contain intact nodes. □

Definition 12. A statement is called *irrefutable* if no intact node can vote against it

The third and final step of federated voting is to commit a statement α.

Definition 13. Confirm A quorum U_α confirms α iff all members of U_α have accepted α.
A node v confirms α if $v \in U_\alpha$ and U_α confirms α

To sum up, the steps of federated voting are shortly described below :
Let $\langle V, Q\rangle$ be a FBAS and α a statement. In order to agree on α, there has to exist a quorum U_α that confirms α, which means that every member of U_α has to fist accept α and then confirm it. For a node v to accept α, it has to either have already voted for α or one of v's blocking sets has accepted α. Every value that nodes agree on, has gone through two or 3 different states (accepted, confirmed or voted accepted, confirmed).

4.2 Nomination Protocol

The Nomination Protocol produces for every node v a set of candidate values Z. Using a deterministic function $combine()$, v will get the input value z for the Ballot protocol.
The Nomination Protocol uses federated voting. Every node chooses for every round $r, 0 \leq r \leq n$, of the Nomination protocol a leader and nominates the same values with the leader, if possible. After the round $r = n$, v always follows the leader of round $r = n$, v_n. Only if $v_0 = v$, then v can introduce a new value to nominate (his input for the Nomination protocol). To select the leader for each round, every node v computes the following :

- $weight(v, v') = \frac{|\{q | q \in Q(v) \wedge v' \in q\}|}{|Q(v)|}$
- $neighbors(v, r) = \{v' \mid G_i(C_1, r, v') < h_{max} \cdot weight(v, v')\}$
- $priority(r, v') = G(C_2, r, v')$

where C_1, C_2 are constants, r is the round number and G_i is a slot specific hash function for slot i, whose range can be interpreted as the set of integers $\{0, ..., h_{max} - 1\}$.

Every node broadcasts, at the end of every round, a message $m = \langle v, i, X, Y, D \rangle$, where v is the node broadcasting the message, i is the slot the nodes are trying to update, X is the set of nominated values, is the set of accepted values and D is v's quorum slices.

Algorithm 3 Nomination Protocol, run by node v, for slot i, where $Input()$ is the input tape and $Receive()$ the reading tape

1: Initialize $X, Y, Z, M, N \leftarrow \emptyset$ ▷ M : set of messages received in the current round
2: ▷ N :set of latest message by each node
3: $t \leftarrow Input()$
4: $r \leftarrow 0$
5: $\langle v_1, ..., v_n \rangle \leftarrow Priorities()$ ▷ computes the leaders for each round
6: $Sets()$
7: **while** $\neg externalize$ **do** ▷ The Nomination protocol terminates when the Ballot protocol terminates
8: $M \leftarrow Receive()$
9: $N \leftarrow UpdateMsg(N, M)$
10: $Z \leftarrow Candidate(Z, Y, M)$
11: $Y \leftarrow Accept(X, Y, N)$
12: **if** $(Z = \emptyset)$ **then**
13: **if** $(r \leq n)$ **then**
14: $m \leftarrow findmsg(v_r, N)$
15: $X \leftarrow Vote(X, m, r, t)$
16: $r \leftarrow r + 1$
17: **else**
18: $m \leftarrow findmsg(v_n, N)$
19: $X \leftarrow Vote(X, m, r, t)$
20: **else**
21: Start the Ballot Protocol
22: $Broadcast(\langle v, i, X, Y, D \rangle)$

The functions used by the Nomination Protocol are :

- $Priorities()$ is the function that computes and returns the set of leaders for every round.
- $Sets()$ is the function that computes the quorums and v's blocking sets.
- $findmsg(u, N)$ is the function that finds u's latest message in N

The $UpdateMsg(N, M)$ updates the set of latest messages sent by each node, N, by comparing X_u and Y_u from the message last received from u to those already stored in u's message in N. In each round, X and Y either grow or stay the same, which means that the latest message sent by u will have a bigger X or Y set.

Algorithm 4

1: **function** UPDATEMSG(M, N)
2: **for** all $m \in M$ **do**
3: $m^* \leftarrow findmsg(m.u, N)$
4: **if** $(\mid m.X \mid > \mid m^*.X \mid) \vee (\mid m.Y \mid > \mid m^*.Y \mid)$ **then**
5: $m^* \leftarrow m$
6: return(N)

The function $Candidate()$ updates the set of candidate values, Z. A value y is added to Z if $candidateQuorum(y) = True$.

- $candidateQuorum(y) = True \Leftrightarrow \exists U \in Q : \forall u \in U[y \in Y_u]$, where $Q = \{U|U : quorum, \}$.

Algorithm 5

1: **function** CANDIDATE(Z, Y, N)
2: **for** all $y \in Y$ **do**
3: **if** $candidateQuorum(y)$ **then**
4: $Z \leftarrow Z \cup \{y\}$
5: return(Z)

The $Accept()$ function updates the set of accepted values Y. A value y is added to Y if $blocing(y) = True$ or $quorum(y) = True$ and $consistent(y, Y) = True$.

- $blocking(y) = True \Leftrightarrow \exists B \in \Lambda_v : \forall u \in B(y \in Y_u)$, where $\Lambda_v = \{B|B : blockingset\}$ the set of blocking sets of v.
- $quorum(y) = True \Leftrightarrow \exists U \in Q : \forall u \in U[(y \in X_u) \vee (y \in Y_u)]$, where $Q = \{U|U : quorum, \}$.

Algorithm 6

1: **function** ACCEPT(X, Y, N)
2: **for** all $x \in N$ **do**
3: **for** all $x \in m.X$ **do**
4: **if** $consistent(x, Y) \bigwedge (blocking(x) \bigvee quorum(x)$ **then**
5: $Y \leftarrow Y \cup \{x\}$
6: **for** all $y \in m.Y$ **do**
7: **if** $consistent(y, Y) \bigwedge (blocking(y) \bigvee quorum(y)$ **then**
8: $Y \leftarrow Y \cup \{y\}$
9: return(Y)

The $Vote()$ function updates the set of nominated values, X.A value x is added in X if it is valid ($valid(x) = TRUE$) and consistent with X ($consistent(x, X) = TRUE$).
Both predicates are defined by the application for which the protocol is used.

Algorithm 7

1: **function** VOTE(X, m, r, t)
2: **if** $r = 0 \wedge v_0 = v$ **then**
3: $X \leftarrow X \cup \{t\}$
4: **else**
5: **for** all $x \in m.X$ **do**
6: **if** $valid(x) \bigwedge consistent(x, X)$ **then**
7: $X \leftarrow X \cup \{x\}$
8: return(X)

4.3 Ballot Protocol

When nodes have a composite value, they start the Ballot protocol, even though the Nomination protocol continues to update the value.The goal is to commit a ballot and externalize its value. Since only one value is chosen for the slot, all committed and stuck ballots contain the same value. Nodes use federated voting to agree on the statements *commit* b or *abort* b in order to agree on a ballot.

A ballot b is a pair $b = \langle n, x \rangle$,where n is a counter and x the value. If nodes agree on b then x will be externalized for the slot in question. If b, b' are two ballots, then b and b' are compatible if $b \sim b' \Leftrightarrow b.x = b'.x$ and incompatible if $b \nsim b' \Leftrightarrow b.x \neq b'.x$. We write $b \lesssim b'$ if $b \leq b'$ and $b \sim b'$. We define the ballot **0** $=< 0, \perp >$ (invalid ballot) as the smallest of all ballots.

To *commit* b, a node needs first to accept and confirm b is prepared and then accept and confirm *commit* b. A ballot b is *prepared* if all the statements in $\{$*abort b* $\mid b_{old} \lesssim b\}$ are true. The first ballot prepared by each node is $\langle 0, z \rangle$, where $z = \perp$ and is later updated to be the composite value from the nomination protocol.During the protocol, each node stores :

- the current phase ϕ (prepare, confirm , externalize)
- the current ballot b the node is trying to prepare and commit
- the two highest ballots p', p accepted as prepared s.t $p' \lesssim p$
- two ballots c, h :
  - When $\phi = prepare$, h is the highest ballot confirmed prepared, or $h =$ **0** if none exists.
    If $c \neq 0$, c is the lowest and h the highest ballot v has voted to commit and not accept abort
  - When $\phi = confirm$, h is the highest and c the lowest ballot v has accepted commit
  - When $\phi = externalize$, h is the highest and c the lowest ballot v has confirmed commit

  The protocol must ensure that if $c \neq 0$ then $c \lesssim b \lesssim h$.
- the value z to use in the next ballot. If $h = 0$, z is the composite value, else $z = h.x$

- the set of latest messages M

The message sent by v, in each round, depends on ϕ.

When $\phi = prepare$, v broadcasts a message $m = \langle v, i, b, p, p', c.n, h.n, D\rangle$, when $\phi = confirm$, v broadcasts a message $m = \langle v, i, b, p.n, c.n, h.n, D\rangle$ and when $\phi = externalize$, v broadcasts a message $m = \langle v, i, x, c.n, h.n, D\rangle$, where i is the specific slot for which the nodes run the protocol and D is the set of quorum slices.

Algorithm 8 Ballot Protocol,run by node v, for slot i, with input the set of candidate values, Z

$\phi \leftarrow prepare, z \leftarrow \perp, b \leftarrow \langle 0, z\rangle$
$p, p', c, h \leftarrow 0, M \leftarrow \emptyset$
$broadcastOk \leftarrow FALSE$
$Sets()$
while $\phi \neq externalize$ **do**
    if $h = 0$ **then**
        $z \leftarrow combine(Z)$
    if $(z \neq \perp)$ **then**
        if $(b.n = 0)$ **then**
            $b \leftarrow \langle 1, z\rangle$
        $broadcastOk \leftarrow TRUE$
    $M \leftarrow Receive()$
    $AcceptPrepared(\phi, p, p', h, c, b, M)$
    if $\phi = prepare$ **then**
        $ConfirmBallots(\phi, p, p', h, c, b, M)$
        $c \leftarrow lowestBallot(\phi, c, h, p, p', M)$
        $AcceptCommit(\phi, h, c, b, M)$
        if $broadcastOk$ **then**
            $Broadcast(\langle v, i, b, p, p', c.n, h.n, D\rangle)$
    else
        $AcceptCommit(\phi, h, c, b, M)$
        $ConfirmBallots(\phi, p, p', h, c, b, M)$
        $Broadcast(\langle v, i, b, p.n, c.n, h.n, D\rangle)$
    if $\phi \neq externalize$ **then**
        $UpdateBallots(b, h, z, M)$
    $SelfValidating(\phi, b, M)$
$Output : (i, c.x)$

We define the following predicates, to use in the Ballot protocol functions :

- $quorum(b, b') = True \Leftrightarrow \exists U \in Q : \forall u \in U \; b \sim b'$, where $Q = \{U | U : quorum, \}$.
- $blocking(p) = True \Leftrightarrow \exists B \in \Lambda_v : \forall u \in B \; (p \sim p_u) \vee (p \sim p'_u)$, where $\Lambda_v = \{B | B : blocking \; set\}$ the set of blocking sets of v.

The functions used by the Ballot Protocol are:

The $AcceptPrepared()$ function corresponds to steps 1 and 5 from the paper [1].
When v can accept new ballots as prepared, if $\phi = prepare$, the function updates p and p', when v accepts new ballots as prepared and then c, if necessary.
If $\phi = confirm$, then v sets p to be the highest accepted prepared ballot such that $b \sim c$.

Algorithm 9

```
1:  function AcceptPrepared(φ, p, p', h, c, b, M)
2:      for all m ∈ M do
3:          if (φ = prepare) then
4:              if quorum(b_m, b) then
5:                  UpdateAccepted(p, p', b_m)
6:              if blocking(p_m) then
7:                  UpdateAccepted(p, p', p_m)
8:              if blocking(p'_m) then
9:                  UpdateAccepted(p, p', p'_m)
10:         else
11:             if (b_m ~ c) ∧ (b_m.n > p.n) then
12:                 p ← b_m
```

The $ConfrimBallots(\phi, p, p', h, c, b, M)$ corresponds to steps 2 and 7 from the paper.
When v can confirm new ballots as prepared, if $\phi = prepare$, the function raises h to the highest of those ballots and sets $z = h.x$. If $\phi = confirm$, then v sets c and h to be the lowest and highest of those ballots and ϕ to $externalize$.

Algorithm 10

1: **function** CONFRIMBALLOTS($\phi, p, p', h, c, b,$)
2: **for** all $m \in M$ **do**
3: **if** $(\phi = prepare)$ **then**
4: **if** $(p_m.n > h.n) \wedge [quorum(p_m, p) \vee quorum(p_m, p')]$ **then**
5: $h \leftarrow p_m$
6: $z \leftarrow h.x$
7: **if** $(p'_m.n > h.n) \wedge [quorum(p'_m, p) \vee quorum(p'_m, p')]$ **then**
8: $h \leftarrow p'_m$
9: $z \leftarrow h.x$
10: **else**
11: **if** $quorum(c_m, c)$ **then**
12: **if** $c.n > c_m.n$ **then**
13: $c \leftarrow c_m$
14: **if** $h.n < h_m.n$ **then**
15: $h \leftarrow h_m$
16: $\phi \leftarrow externalize$

The $lowestBallot()$ function corresponds to steps 3 and 6 from the paper.

If $\phi = prepare, c = 0, b \leq h$ and neither $p \gtrsim h$ nor $p' \gtrsim h$, the function sets c to be the lowest ballot, such that $b \lesssim c \lesssim h$. If $\phi = confirm$ the function raises c to the lowest ballot, such that v accepts all $\{$*commit b'* $\mid c \lesssim b' \lesssim h\}$.

Algorithm 11

1: **function** LOWESTBALLOT(ϕ, c, h, p, p', M)
2: **for** all $m \in M$ **do**
3: **if** $\phi = prepare$ **then**
4: **if** $(c = 0) \wedge (b \leq h) \wedge [(p \lesssim h) \vee (p' \lesssim h)]$ **then**
5: set c to the lowest ballot s.t. $b \lesssim c \lesssim h$
6: **else**
7: raise c to the lowest ballot such that v accepts all $\{$*commit b'* $\mid c \lesssim b' \lesssim h\}$
8: return(c)

The $AcceptCommit()$ function corresponds to steps 4 and 6 from the paper. When v can accept to commit a ballot, if $\phi = prepare$, the function sets c to the lowest and h to the highest ballot such that v accept *commit b'* for all b' s.t. $c \lesssim b' \lesssim h$. The function also sets $z = h.x$ after h has been updated and if $h \gtrsim b$ and ϕ to $confirm$. If $\phi = confirm$, then v sets c and h to be the lowest and highest of the ballots he accepted to commit.

Algorithm 12

1: **function** ACCEPTCOMMIT(ϕ, c, h, b, M)
2: **for** all $m \in M$ **do**
3: **if** $c = 0 \wedge [quorum(h_m, c) \vee blocking(h_m)]$ **then**
4: $c \leftarrow h_m$
5: **if** $quorum(c_m, c) \vee blocking(c_m)$ **then**
6: **if** $\phi = prepare$ **then**
7: **if** $c.n > c_m.n$ **then**
8: $c \leftarrow c_m$
9: **if** $h.n > h_m.n$ **then**
10: $h \leftarrow h_m$
11: $z \leftarrow h.x$
12: **if** $\neg(h \lesssim b)$ **then**
13: $b \leftarrow h$
14: $\phi \leftarrow confirm$
15: **else**
16: **if** $(h \lesssim h_m)$ **then**
17: $h \leftarrow h_m$
18: $c \leftarrow lowestBallot(\phi, c, h, p, p', M)$

The function $UpdateBallots()$ corresponds to steps 8 and 9 from the paper. We define :

- $fallenBehind(M) = True \Leftrightarrow \exists S \subseteq M$ such that $\{u_{m'} \mid m' \in S\}$ is a blocking set and $\forall m' \in S, b_{m'}.n > b_v.n$

Algorithm 13

1: **function** UPDATEBALLOTS(b, h, z, M)
2: **if** $(b.n < h.n)$ **then**
3: $b \leftarrow h$
4: **if** $fallenBehind(M)$ **then**
5: $n \leftarrow lowestcounter(M)$
6: $b \leftarrow \langle n, z \rangle$

- $SelfValidatig()$

  Nodes use the $SelfValidatig()$ function to update b when a ballot fails or takes long enough that it might fail.

  A node v sets an alarm, when $\phi_v \neq externalize$ and there exists a set $S \subseteq M$ such that $U = \{u_m \mid m \in S\}$ is a quorum and $\forall m \in S, (b_m.n \geq b_v.n)$. When the alarm fires, v updates b_v by setting $b_v \leftarrow < b_v.n+1, z_v >$

- $Sets()$ is the function that computes the quorums and v's blocking sets.

- $lowestcounter(M)$ returns the lowest counter n such that $fallenBehind(M) = False$

The $UpdateAccepted()$ function is used in $AcceptedPrepared()$ to update the accepted ballots if necessary .

Algorithm 14

1: **function** UPDATEACCEPTED(p, p', b)
2: **if** $(p \lesssim b) \wedge (p' \lesssim b)$ **then**
3: $p' \leftarrow p$
4: $p \leftarrow b$
5: **if** $(p \lesssim b)$ **then**
6: $p \leftarrow b$
7: **if** $(p' \lesssim b)$ **then**
8: $p' \leftarrow b$

4.4 Persistence and Liveness for the SCP

To argue about persistence and liveness for the SCP protocol, we first need to show that both the Nomination and the Ballot protocol satisfy certain properties. Specifically, for persistence we need show that the Ballot protocol satisfies the following agreement property :

Definition 14. Aggreement : All intact nodes return the same value.

For the desired liveness property, defined in chapter 2, we will define a validity property for the Ballot protocol and using that property we show that the Ballot protocol satisfies liveness as well. But to guarantee liveness for the Stellar Consensus protocol we need both the Nomination and the Ballot protocol to satisfy liveness.

We already know that intact nodes satisfy the node-liveness property of the Stellar Consensus Protocol, but that property is not enough to guarantee the liveness property necessary for a robust transaction ledger. Intact nodes may be able to externalize values without the participation of ill-behaved nodes, but it is not ensured that those values are also inputs of intact nodes. To clarify that, we note that nodes externalize values through the Ballot protocol and indeed intact nodes will externalize values that where the input of an intact node *for the Ballot protocol*. This means that persistence for the Stellar Consensus protocol is satisfied if the Ballot protocol satisfies persistence. However, a node's input will end up in the confirmed ballot and eventually in the slot, only if that value is included in the composite value produced by the Nomination protocol. Given all that, it is necessary for both the Nomination and the Ballot protocol to satisfy liveness for the Stellar protocol to satisfy liveness as well.

4.4.1 Persistence for the Ballot protocol

Our first step is to show that all intact nodes will eventually terminate the protocol.

Proposition 1. *Let $\langle V, Q\rangle$ be an FBAS with quorum intersection. Then all intact nodes in $\langle V, Q\rangle$ will eventually terminate.*

Proof. Let $\langle V, Q\rangle$ be an FBAS with quorum intersection and $B \subseteq V$ the set of befouled nodes. By Theorem 2, B is a DSet and by the definition of a DSet, either $B = V$ or $V \setminus B$ is a quorum in $\langle V, Q\rangle$.
If $B = V$, then there are no intact nodes in $\langle V, Q\rangle$ and the proposition holds.
If $B \subset V$, then $V \setminus B$ is a quorum in $\langle V, Q\rangle$ that contains only intact nodes.
A node v terminates when it reaches the $externalize$ phase. Every node starts with $\varphi = prepare$ and moves to $\varphi = confirm$ and then $\varphi = externalize$, following the protocol.
To get from $\varphi = prepare$ to $\varphi = confirm$, v needs to accept a ballot as prepared, then confirm a ballot is prepared and finally accept a ballot is committed. By the definition of *accept* v needs either a quorum U, with $v \in U$ or a blocking set of v to accept a statement.
To get from $\varphi = confirm$ to $\varphi = externalize$, v needs to confirm a ballot is committed. By the definition of *confirm*, v needs a quorum U with $v \in U$ to confirm a statement.
If v is intact, then there is such a quorum, $U = V \setminus B$ and using that quorum, v can reach the externalize phase and terminate. Since the Nomination protocol terminates when the Ballot protocol terminates, it follows that all intact nodes will eventually terminate the protocol. □

Theorem 4. Let U be a quorum in a FBAS $\langle V, Q\rangle$ and $B \subseteq V$ a set of nodes. Let $U' = U \setminus B$. If $U' \neq \emptyset$, then U' is a quorum in $\langle V, Q\rangle^B$

Proof. By the definition of a quorum, for every $v \in U$ there is a $q \in Q(v)$ such that $q \subseteq U$. Since $U' \subseteq U$, every $v \in U'$ has a quorum slice $q \in Q(v)$ such that $q \setminus B \subseteq U'$. Using the *delete* notation, we have that $\forall v \in U'$, $\exists q \in Q^B(v)$ such that $q \subseteq U'$ and because $U' \subseteq U \subseteq B$ it follows that U' is a quorum in $\langle V, Q\rangle^B$. □

Proposition 2. *Two intact nodes, in a FBAS $\langle V, Q\rangle$ with quorum intersection, cannot confirm contradictory statements.*

Proof. Let B be the set of all befouled nodes. Let v_1, v_2 be two intact nodes and a_1, a_2 two contradictory statements. We assume v_1 confirms a_1 and v_2 confirms a_2.
By the definition of *confirm*, there exist quorums U_1 and U_2 that confirm a_1 and a_2 respectively. Since $\langle V, Q\rangle$ has quorum intersection, $\exists v' \in U_1 \cap U_2$ and v' confirms both a_1 and a_2. It follows that v' is ill-behaved and by Theorem 1 B is the intersection of all DSets containing all ill-behaved nodes, $\Rightarrow \quad v' \in B$.
By Theorem 2, $U_1^* = U_1 \setminus B$ and $U_2^* = U_2 \setminus B$ are quorums in $\langle V, Q\rangle^B$ that confirm a_1 and a_2 respectively. Because B is a DSet, $\langle V, Q\rangle^B$ has quorum intersection, therefore $U_1^* \cap U_2^* \neq \emptyset$. Let v^* be a node in $U_1^* \cap U_2^*$. If v^* was befouled, then $v^* \in B$, but $v^* \in U_1^* = U_1 \setminus B \Rightarrow v^*$ is intact. But v^* confirms a_1 and a_2 which contradicts the definition of intact nodes. □

If two ballots b, b' are incompatible $(b \nsim b')$, then *accept(b)* and *commit(b)* are contradictory to *accept(b')* and *commit(b')* respectively.

Theorem 5. In a FBAS $\langle V, Q \rangle$, with quorum intersection, the Ballot Protocol satisfies the agreement property for intact nodes. Specifically, there is a round after which all intact nodes return a ballot with the same value x.

Proof. Nodes executing the Ballot protocol, return a value x, when they have confirmed *commit*(b) for a ballot $b = \langle n, x \rangle$.
By proposition 2, two intact nodes can't confirm incompatible ballots as committed and by proposition 1, all intact nodes will eventually terminate. It follows that all intact nodes, executing the Ballot protocol will return the same value. □

Theorem 6. In a FBAS $\langle V, Q \rangle$ with quorum intersection, the Stellar Consensus protocol satisfies the persistence property of a robust transaction ledger.

Proof. The theorem follows directly from theorem 5. By theorem 5, all intact nodes that run the Ballot protocol return the same value (agreement property). Given the agreement property and the fact that nodes never delete or overwrite values in a slot, it is easy to conclude that once intact nodes agree on a value, that value is stable and no intact node will ever report a conflicting value for that slot.

□

4.4.2 Liveness for the Nomination and the Ballot protocol

To examine the liveness property of SCP with first show that the Ballot protocol satisfies that property. Specifically, we show that given a transaction tx that all intact nodes want to include in the ledger, if tx is already included in the composite value of an intact node, then tx will be included in the current slot. For that purpose we define the following property :

Definition 15. Ballot Validity : The output returned by an intact node v equals the input of some intact node u

Proposition 3. *An intact node only accepts $prepare(b)$ statements, for ballots $b = \langle n, x \rangle$ where $b.x$ is the input of an intact node.*

Proof. Let v be an intact node.By Definition 11, v accepts $prepare(b)$ for some ballot b if :

1. There exists a quorum U s.t $v \in U$ and $\forall u \in U$, u has either voted for or accepted $prepare(b)$
   A node votes for $prepare(b)$ if $b.x$ is its input.
2. Every member of a blocking set of v has accepted $prepare(b)$

We use induction on the number of rounds r to prove the proposition.
For $r = 1$, since condition 2 cannot hold (no node has accepted any statement yet), v accepts $prepare(b)$ only if there exists a quorum U s.t $v \in U$ and $\forall u \in U$, u has voted for $prepare(b)$. But v votes for $prepare(b)$ only if $b.x$ is v's input. Since v is intact the proposition holds.

We assume that in round r, the proposition holds and no intact node has accepted a $prepare(b)$ statement, where $b.x$ is not the input of an intact node.
In round $r+1$, v accepts $prepare(b)$ for some ballot b. If v accepted $prepare(b)$ through condition 1, then there exists a quorum U s.t $v \in U$ and $\forall u \in U$, u has either voted for or accepted $prepare(b)$. By the inductive hypothesis, it follows that $b.x$ is the input of an intact node.
If v accepted $prepare(b)$ through condition 2, then by corollary 1 and the inductive hypothesis it follows that $b.x$ is the input of an intact node. □

Theorem 7. In a FBAS $\langle V, Q \rangle$ with quorum intersection, the Ballot Protocol satisfies the Ballot Validity property for intact nodes.

Proof. Nodes executing the Ballot protocol, return a value $c.x$, when they have confirmed ***commit***(c), for ballot c. To prove that the Ballot Protocol satisfies the validity property, we examine the possible values that c may carry during the execution of the protocol.
While $\phi = prepare$, c is the lowest ballot v has voted to commit and not accepted an ***abort***(c). To vote for ***commit***(c), a node first has to accept $prepare(c)$ and then confirm $prepare(c)$. By proposition 3, intact nodes cannot accept $prepare(c)$, if $c.x$ is not the input of an intact node. As a consequence, intact nodes cannot vote for a $commit(c)$ statement, if $c.x$ is not the input of an intact node.
While $\phi = confirm$, c is the lowest ballot for which v has accepted ***commit***(c) and when $\phi = externalize$, c is the lowest ballot for which v has confirmed ***commit***(c). Since intact nodes cannot vote for $commit(c)$, if $c.x$ is not the input of an intact node, an intact node v will only have ballots carrying the input of intact nodes reaching the $externalize$ phase in his execution of the protocol.

□

The Nomination Protocol has the following properties :

1. Intact nodes can produce at least one candidate value.
   Property 1 is achieved because nominate statements are *irrefutable*. Nodes never vote against nominating a particular value and until the first candidate value is confirmed, intact nodes can vote to nominate any value.

2. At some point, the set of possible candidate values stops growing.
   Property 2 is ensured because once each intact node confirms at least one candidate value, which will happen in a finite amount of time, no intact node will vote to nominate any new values.

3. If any intact node considers x to be a candidate value, then eventually every intact node will consider x to be a candidate value.
   Property 3 follows directly from proposition 2

Theorem 8. In a FBAS $\langle V, Q \rangle$, with quorum intersection, there is a round after which all intact nodes will have the same composite value.

Proof. The theorem follows directly from the three properties of the nomination protocol. Each intact node will only ever vote to nominate a finite number of values. In the absence of ill-behaved nodes, intact nodes will converge on the same set of candidate values. Ill-behaved nodes may delay this convergence by introducing new values, which for a period of time *may be candidates at some but not all intact nodes.* Such values will need to have garnered votes from well-behaved nodes, which limits them to a finite set. Eventually, ill-behaved nodes will either stop perturbing the system or rum out of new candidate values to inject, in which case intact nodes will converge. □

We now need to examine whether the Nomination protocol satisfies liveness as well. To do that we make the following observations :

By theorem 8, we know that eventually all intact nodes will have the same composite value, which means all intact nodes will have the same set of candidate values Z, since the $combine()$ function that produces the composite value is deterministic. Furthermore, by theorem 6 we know that the output returned by an intact node v equals the input of some intact node u (Ballot validity).To argue about liveness of the Nomination protocol, we need to examine the values that are included in the composite value.

Let x be the input value of a node v. Then obviously x can be included in the composite value of v if and only if $v_0 = v$, since in any other case x will never be introduced in the protocol. In a FBAS with quorum intersection, if all intact nodes want to include the same transaction tx to the ledger and for every intact node v, it holds that $v_0 = v$, then there is a quorum consisting only of intact nodes and using that quorum, intact node will include tx in the composite value. Unfortunately, that is not possible.

Proposition 4. *In a FBAS $\langle V, Q\rangle$, with quorum intersection, not all input values of intact nodes can be included in the composite value.*

Proof. Let $\langle V, Q\rangle$ be an FBAS with quorum intersection and $B \subseteq V$ the set of befouled nodes.By Theorem 1, B is a DSet.

If $B = V$, then there are no intact nodes in $\langle V, Q\rangle$ and the proposition holds.

If $B \subset V$, then $U = V \setminus B$ is a quorum in $\langle V, Q\rangle$ that contains only intact nodes. It follows that for every intact node v there is a quorum slice $q \in Q(v)$ that only contains intact nodes ($q \subseteq U$). Then for every round r, $neighbors(v, r) = \{v' \mid G_i(C_1, r, v') < h_{max} \cdot weight(v, v')\}$ will contain an intact node.

We assume that for every intact node $v_0 = v$ and let v be the intact node with minimum $priority(0, v)$ among intact nodes. Since for every round r, $neighbors(v, r)$ will contain an intact node, there is an intact node $u \in neighbors(v, 0)$. But v has minimum $priority(0, v_1)$ among intact nodes, hence $priority(0, v) < priority(0, u)$. Then v's leader for $r = 0$ must be u which contradicts the hypothesis. □

We now need to explore when the input of an intact node is included in the composite value.

Let x be the input value of an intact node v. We assume that $x \in Z_v$. Then there is a quorum U that confirms the statement *accept(x)*. Every node in U, at some point during the execution of the protocol accepted x. It must hold that every node in U is well-behaved. If there was an ill-behaved node in U, then he would try to prevent x from being confirmed, by not claiming to accept it. It follows that $\forall u \in U,\ u$ is well-behaved.

By the definition of *accept* every node u in U either voted x or there was blocking set of u claiming to accept x. We examine the first node that accepts x. Let that node be v'. Since v' is the first node to accept x, v' can only accept x by condition 1, hence there is a quorum U^* that ratifies x. If U^* contains an ill-behaved node once again, that node will try to prevent x from being accepted, by not voting for it. Hence, U^* can only contain well-behaved nodes. Since those nodes voted for x it must hold that they either had v as leader in some round r, or they had a leader that followed v in a previous round and no ill-behaved node was their leader between that round and round r. By theorem 8, we know that since $x \in Z_v$ and v is intact, eventually x will be included in the Z set of all intact nodes.

Remark :

Liveness for the Nomination protocol depends on the quorum slices of intact nodes and their leaders in every round. The following example helps to clarify that.

LIVENESS WITH HIGH PROBABILITY

Let $\langle V, Q \rangle$ be a FBAS that has no ill-behaved nodes and $\forall v \in V,\ V \subseteq Q(v)$. Since the $priority()$ function is the same for all nodes and every node trusts every other node in V it follows that for every round r there will be one common leader for all the nodes.

Let v be the leader for $r = 0$. Then x_v (v's input) will be the only value introduced to the protocol. But to have x_v in the composite value, there must be at least one more round that v is the leader.

The probability that a node is the leader in round r is $p_l = \frac{1}{n}$.

The probability that a node is not the leader in rounds 1 to n is $p_{notLeder} = (\frac{n-1}{n})^{n-1}$. Then the probability that a node is leader in at least one of the rounds $1, 2, ..., r$ is $p = 1 - (\frac{n-1}{n})^{n-1} \underset{n\to\infty}{\longrightarrow} 1 - \frac{1}{e}$.

Chapter 5

Conclusion

5.1 Summing up

The protocols we studied maintain a ledger of transactions, divided in slots. Every slot stores a block of transactions, those that the nodes agreed on while executing the protocol for that specific slot. In the beginning of this thesis we defined the notion of *robust transaction ledger* and the properties a protocol needs to satisfy to implement such a ledger. We also determined the execution model under which we would study our consensus protocols. The protocols were contemplated under an environment Z that generates all the nodes that run a protocol Π and the adversary A and a controller C. Together (Z, C) are responsible for the execution of the protocol. The necessary functionalities for the model are the ”Diffuse” and the ”Key registration” functionality.

Under that model we first examined the Raft protocol and showed that indeed, Raft implements a robust transaction ledger. The protocol is separated in two parts *Leader election* and *Log replication*. Nodes first elect a leader, using the $LeaderElection()$ function. A node is elected leader if there is a majority of nodes that vote for him. After the leader is elected, nodes continue with the $LogReplication()$ function. It is the leader's job to service client requests and ensure that there are no inconsistencies between his log and the logs of follower nodes. Raft enjoys several properties that combined guarantee the persistence and liveness property of a robust transaction ledger, given that there is a majority of honest nodes.

We proceeded by presenting the Stellar Consensus protocol. We analysed both the Nomination protocol and the Ballot protocol, Stellar's components. The Nomination protocol takes nodes inputs and produces a composite value that is then used as input for the Ballot protocol. We showed that the Ballot protocol satisfies both persistence and liveness. Since the Ballot protocol satisfies persistence we concluded that the Stellar Consensus protocol satisfies persistence as well. However, for liveness with need both of Stellar's components to satisfy the property to in order for Stellar to have the property. Liveness for the Nomination protocol depends on the intact nodes' quorum slices and their leaders for every round.

5.2 Future work

An obvious next step is to examine the conditions under which the Nomination protocol is guaranteed to have liveness. Since in a FBAS $\langle V, Q \rangle$, with quorum intersection, not all input values of intact nodes can be included in the composite value, we step to that end is to examine the conditions that quorum slices and leaders need to comply with, so that the Nomination protocol satisfies a validity property such as the following :

Definition 16. Nomination Validity : At least one input value of an intact node will end up in the composite value.

In the paper *In Search of an Understandable Consensus Algorithm* [4], there is section devoted to cluster membership changes. Specifically, a ”joint consensus” mechanism is proposed to deal with changes. Joint consensus works by combining both old and new configurations. Any node from either configuration can be leader and for agreement majorities from both configurations are required. In such a situation we need to explore whether the protocol maintains both persistence and liveness.

All our conclusions concerning the SCP, are also made under the assumption, that quorums and by extension quorum slices remain the same between rounds during the execution of both protocols. However, since nodes may stop participating in the protocol, in reality quorums may change between rounds. Once again we need to explore whether the protocol maintains its properties under reconfiguration and furthermore if a mechanism similar to that of ”joint consensus” proposed in Raft, could be used in the SCP protocol to move from one version of the network to another.

Bibliography

[1] David Mazieres, Stellar Development Foundation
The Stellar Consensus Protocol: A Federated Model for Internet-level Consensus

[2] Leslie Lamport Paxos Made Simple, 01 November 2001

[3] Miguel Castro, Barbara Liskov Practical Byzantine Fault Tolerance *Proceedings of the 3d Symposium of Operating System Design and Implementation, New Orleans, USA, 1999*

[4] Diego Ongaro and John Ousterhout In Search of an Understandable Consensus Algorithm *Stanford University*

[5] http://cryptorials.io/glossary/

[6] Satoshi Nakamoto Bitcoin: A Peer-to-Peer Electronic Cash System

[7] Juan A. Garay, Aggelos Kiayias, Nikos Leonardos The Bitcoin Backbone Protocol: Analysis and Applications

[8] http://dcg.ethz.ch/lectures/
Principles of Distributed Computing (lecture collection)

[9] Aggelos Kiayias, Alexander Russell, Bernardo David, Roman Oliynykov Ouroboros: A Provably Secure Proof-of-Stake Blockchain Protocol

[10] Brian M. Oki, Barbara H. Liskov Viewstamped Replication: A New Primary Copy Method to Support Highly-Available Distributed Systems

[11] Jae Kwon Tendermint: Consensus without Mining

[12] Nicolas van Saberhagen CryptoNote v 2.0

[13] Michael J. Fischer, Nancy A. Lynch, Michael S. Paterson Impossibility of Distributed Consensus with One Faulty Process

[14] http://thesecretlivesofdata.com/raft/