Simplified SCP

David Mazières, Giuliano Losa, and Eli Gafni

March, 2019

Introduction

We present a simplified version of the Stellar Consensus Protocol (SCP) that we hope is easier to read than the SCP whitepaper and internet draft.

Federated Byzantine agreement

Traditional Byzantine agreement solves the consensus problem in a closed system of N nodes. Typically, protocols assume $N = 3f + 1$ for some positive integer f, and then guarantee safety and some form of liveness so long as at most f nodes are faulty. Conceptually, these protocols involve voting with a quorum size of $2f + 1$. If there are only $N = 3f + 1$ nodes, then any two quorums of size $2f + 1$ must overlap in at least $f + 1$ nodes, so that even if f of the overlapping nodes are faulty, the two quorums share at least one non-faulty node. This non-faulty common node can ensure quorums never reach contradictory conclusions.

Federated Byzantine agreement is an adaptation of Byzantine agreement to systems with open membership, where different nodes have different ideas about which nodes are important and what a quorum should contain. For example, node v_1 might consider that a quorum needs to contain $\geq 3/4$ of the nodes in some set S, while node v_2 believes that a quorum should contain $> 2/3$ of some similar but not identical set S'. Nodes might also have less symmetric requirements on quorums—e.g., node v_3 might require that a quorum contain a majority of the nodes run by company X and a majority of the nodes run by company Y, where X runs many more nodes than Y.

The fact that node v_1 believes a quorum should contain $3/4$ of set S does not mean that $3/4$ of set S is a quorum. Rather, we say that any $3/4$ of set S constitutes a **quorum slice** for node v_1. A quorum is a set that contains at least one quorum slice for each of its members. More precisely, since faulty nodes can choose deficient or inconsistent quorum slices, we say that a **quorum** is a non-empty set of nodes containing at least one quorum slice for each of its non-faulty members.

Federated Byzantine agreement is thus a generalization of Byzantine agreement. If every node chooses the exact same quorum slices, then federated Byzantine agreement is just Byzantine agreement—a mechanism to ensure agreement among all non-faulty nodes while tolerating

specific failure scenarios. When different nodes choose different quorum slices, however, only a subset of non-faulty nodes may be guaranteed agreement under certain failure scenarios. Safety still depends on quorum overlap, but is no longer a system-wide property. Specifically, we say non-faulty nodes v_1 and v_2 are **intertwined** if every quorum containing v_1 intersects every quorum containing v_2 in at least one non-faulty node. A federated Byzantine agreement protocol can guarantee agreement between two nodes only if they are intertwined. Since SCP provides this guarantee, we say it is optimal for safety.

When a set of non-faulty nodes I is a quorum and every two nodes of I are intertwined in the "projected system" that results from removing nodes not in I from all quorum slices, then we say that I is **intact**. SCP guarantees liveness under eventual synchrony for intact sets—even without members knowing the intact set's membership.

The quorum slice model enables two key mechanisms: federated voting and federated leader election. In **federated voting**, nodes vote on statements (e.g., "transaction set x is nominated," as explained in the next section) and, through a two-step protocol, may *confirm* statements with the confidence that intertwined nodes will never confirm contradictory statements. Moreover, an intact set of nodes I enjoys additional guarantees: members of I can never confirm statements that no member of I voted for, and if any member of I confirms a statement, eventually all members of I will confirm the statement provided nodes keep taking steps in the protocol. Of course, even for intact nodes, any particular instance of federated voting can get permanently stuck should nodes split their votes among contradictory statements such that no statement can garner a unanimous quorum.

Federated leader selection allows nodes to pick one or a small number of leaders pseudorandomly in a way that respects the priorities implicit in quorum slices. When repeated, members of an intact set will eventually all (transitively) follow the same unique intact leader with probability 1.

Protocol description

SCP comprises three interlocking parts. *Nomination* is a mechanism by which nodes converge on a set of candidate values. *Balloting* is the core of SCP, which, when successful, actually chooses an output value. *Timeout* is a mechanism through which nodes give up on a ballot that has not chosen a value and try again.

Nomination

Nomination entails federated voting on statements of the form:

- **NOMINATE** x: States that x is a valid candidate consensus value.

Nodes may vote to nominate multiple values—different NOMINATE statements are never contradictory. However, once a node confirms any NOMINATE statement, it stops voting to nominate new values. Federated voting still allows a node to confirm new NOMINATE statements it didn't vote for, which allows members of an intact set to confirm one another's nominated values while withholding new votes.

The (evolving) result of nomination is a deterministic combination of all values in confirmed NOMINATE statements. For example, if x represents a set of transactions, nodes can take the union of sets, the largest set, or the set with the highest hash, so long as all nodes do the same thing. Because nodes withhold new votes after confirming the first NOMINATE statement, the set of confirmed NOMINATE statements can contain only finitely many values. The fact that confirmed values reliably spread through an intact set means that intact nodes eventually converge on the same set of set of nominated values and hence the same nomnation result. However, the convergences occurs at an unknown point arbitrarily late in the protocol.

Nodes employ federated leader selection to reduce the number of different values in NOMINATE statements. Only a leader who has not already voted for a NOMINATE statement may introduce a new x in a NOMINATE vote. Other nodes wait to hear from a leader and then just copy leaders' NOMINATE votes. To accommodate failure, the set of leaders keeps growing as timeouts occur, but in practice only a few nodes introduce new values of x.

Balloting

A strawman consensus protocol might directly vote to output the result of nomination. Should one member of an intact set confirm an output, eventually all members will do so and hence reach agreement. Unfortunately, nodes run the risk of attempting this vote before nomination has converged, in which case different nodes will vote to output different values and the whole vote may get permanently *stuck* with no possibility of any value ever gaining a quorum.

What's worse, nodes cannot reliably determine that a vote has gotten stuck. Suppose an output value is one vote short of a quorum. Perhaps a node that seems to have crashed is actually just slow to propagate its vote, but did in fact complete the quorum. Or perhaps a malicious node sent contradictory votes to different peers, convincing some peers that an output value reached quorum and others that a quorum is impossible.

To address stuck votes, SCP votes on a series of numbered ballots. Even if a stuck vote precludes choosing a value in ballot n, we can time out and try again with ballot $n + 1$. However, since nodes may be unable to determine that a vote is stuck, a node can come to one of three conclusions about the result of ballot n:

1. No value was or will ever be chosen by a quorum in ballot n.
2. Value x was chosen by a quorum in ballot n.
3. No value other than x was or will ever be chosen in ballot n, and while the vote on x seems stuck, other nodes may have concluded the result was outcome 1 or 2, or a node may later transition from outcome 3 to 1 or 2.

The principle invariant in SCP is that all ballots with outcome 2 or 3 must have the same value x. We ensure this through federated voting on two statements:

- **PREPARE** $\langle n, x \rangle$: This states that no value other than x was or will ever be chosen in any ballot $\leq n$. (A node may not vote for this statement if it prevoiusly cast a vote for COMMIT $\langle n', x' \rangle$ with $x' \neq x$ and that vote might still complete a quorum.)

- **COMMIT** $\langle n, x \rangle$: This states that value x is chosen in ballot n.

A node can output x after confirming COMMIT $\langle n, x \rangle$ for any n, but may not vote for COMMIT $\langle n, x \rangle$ without first confirming PREPARE $\langle n, x \rangle$. Nodes set x to the nomination result when $n = 1$, and choose x as described in Timeouts below for subsequent ballots.

Note the full SCP protocol compactly encodes the sender's entire relevant voting history in every message, avoiding the need to retransmit old messages that are lost. In simplified SCP, we just assume reliable message delivery between non-faulty nodes. Full SCP also adds a final EXTERNALIZE message for the archives, which lets slow nodes catch up weeks after the fact, even if node retirements preclude assembling a quorum with historical slices.

Timeouts

If nodes are unable to confirm a COMMIT statement for the current ballot, they give up after a timeout, which gets longer with each ballot so as to adjust to arbitrary bounds on network delay. Nodes start the timer once they are part of a quorum that is all at the current (or a later) ballot counter n. If at any point every one of a node's quorum slices contains a node with a higher ballot counter, the node immediately skips to the lowest ballot such that this is no longer the case, regardless of any timers.

The timer mechanism synchronizes nodes' ballots: The clock starts only when a quorum is on the latest ballot, but if there are intact stragglers, the quorum that started the clock will intersect all slices of at least one intact straggler, which will immediately catch up and in turn help other intact nodes catch up until eventually all intact nodes have caught up.

When ballot n times out on node v, v picks a value x and votes for PREPARE $\langle n + 1, x \rangle$. There are two strategies for picking x:

1. If v has confirmed any PREPARE $\langle n', x' \rangle$ statement, it sets x to x' from the one with the highest n'. Otherwise, v sets x to the current result of nomination.

2. v uses a single round of federated leader selection to choose a leader. If v chooses itself as leader, it uses strategy 1. Otherwise, v copies its leader's PREPARE vote if it can legally do so; old COMMIT votes by v restrict what PREPARE statements it can vote for unless the COMMIT votes have been overruled and can no longer complete a quorum. If a second timeout occurs and v still has not heard from its leader or overruled its past conflicting COMMIT votes, v falls back to strategy 1.

   Note that when v is catching up because every one of its quorum slices contains a node at a higher ballot, v cannot wait for its leader and must immediately use strategy 1 if it has not already heard a PREPARE message from its leader.

Under partial synchrony, strategy 1 guarantees termination with fail-stop nodes, but allows malicious nodes with good network timing to delay termination until they are manually removed from quorum slices. Strategy 2 guarantees termination with probability 1 with Byzantine failures. Since fail-stop behavior is the common case, a hybrid approach is to follow strategy 2 only every k ballots (e.g., $k = 5$).

Building blocks

This section details the two lower-level building blocks used in the above protocol description, namely *federated voting* and *federated leader selection*.

Federated voting

Federated voting consists of nodes casting vote messages that also specify their quorum slices. Recipients of these messages can dynamically discover quorums based on voters' stated slices. Quorum overlap ensures intertwined nodes will not find themselves in contradictory unanimous quorums.

However, knowing that no quorum will contradict a is insufficient to confirm a; when a member of an intact set confirms a, we also need to guarantee that the rest of the set will eventually do so. This requires ensuring not just that a received a quorum, but that a quorum *knows* that a received a quorum. Furthermore, this quorum must be able to convince other nodes, including ones that voted against a, to confirm a.

To this end, federated voting employs a three-phase protocol in which nodes first vote for a statement (broadcasting a message to this effect), then accept it (again broadcasting the fact), and finally confirm it. A node v may *vote* for any valid statement a consistent with its other outstanding votes and accepted statements. v *accepts* a when v is a member of a quorum in which every node either votes for a or accepts a. Even if v did not vote for a, if every one of v's quorum slices contains a node accepting a and v has accepted nothing contradicting a, then v also accepts a; the set of accepting nodes intersecting all of v's quorum slices *overrules* any contradictory votes v may have previously cast by proving these contradictory votes could not have been part of a quorum. Finally, when v is a member of a quorum in which every node accepts a, then v *confirms* a.

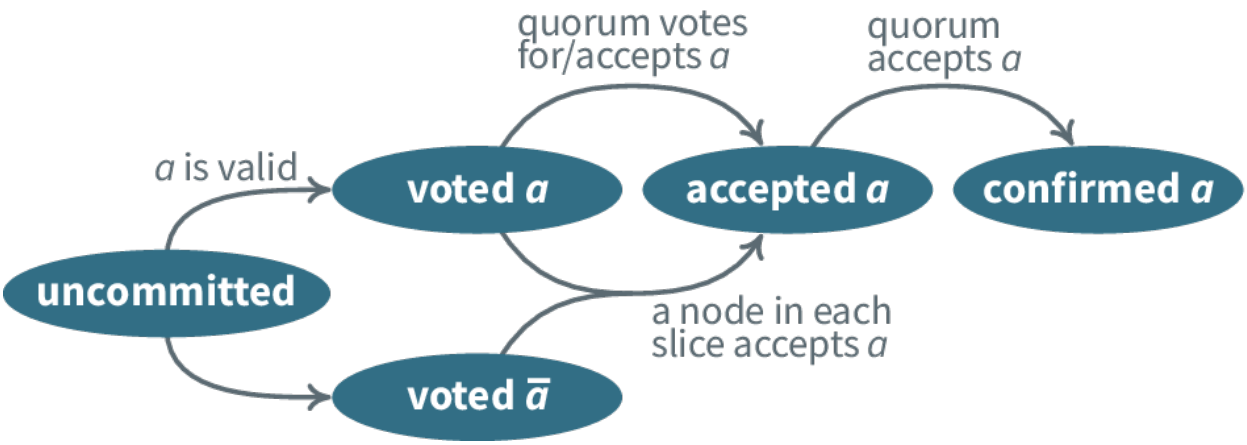

Figure 1: federated voting process

Two intertwined nodes cannot confirm contradictory statements, as the two required quorums would have to share a non-faulty node that could not accept contradictory statements. In addition, if a node in an intact set I confirms a statement, then either the quorum that

accepted the statement contains I, or that quorum will intersect all of the quorum slices of at least one other member of I. Hence, through a cascading effect, eventually all of I will accept and confirm a. Of course, there is no guarantee that an intact node will ever confirm a in the first place, which is why the balloting protocol is designed to survive stuck votes.

Federated leader selection

Leader selection allows each node to pick leaders in such a way that nodes generally only choose one or a small number of leaders. To accommodate leader failure, leader selection proceeds through rounds. If leaders of the current round appear not to be fulfilling their responsibilities, then after a certain timeout period nodes proceed to the next round to expand the set of leaders they follow.

Each round employs two unique cryptographic hash functions, H_0 and H_1, that output integers in the range $[0, h_{\max})$. For instance, we can set $H_i(m) = \text{SHA256}(i \| b \| n \| r \| m)$, where b is the overall SCP instance (block or ledger number), n identifies the ballot for which the leader is being selected (0 for nomination or the ballot number for timeout), r is the leader selection round number, and $h_{\max} = 2^{256}$. Within a round, we define the priority of node v as:

$$\text{priority}(v) = H_1(v)$$

One strawman would be for each node to choose as leader the node v with the highest priority(v). This approach works well with nearly identical quorum slices, but doesn't properly capture the importance of nodes in imbalanced configurations. For instance, if Europe and China each contribute 3 nodes to every quorum, but China runs 1,000 nodes and Europe 4, then China will have the highest-priority node 99.6% of the time.

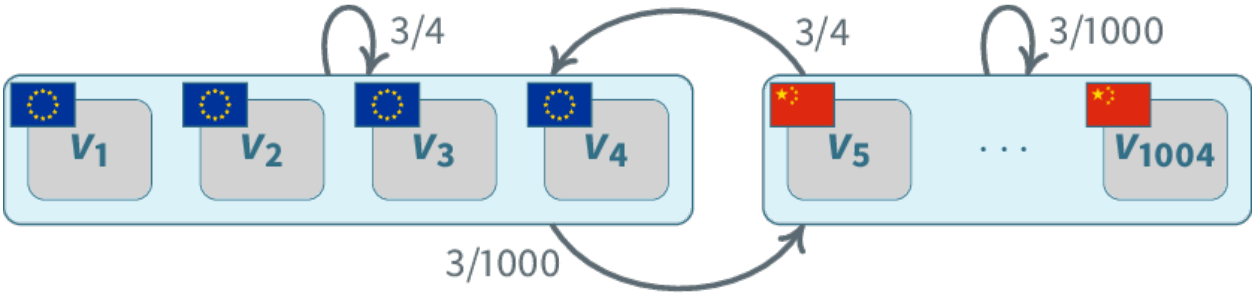

Figure 2: imbalanced configuration

We therefore introduce a notion of slice weight, where weight$(u, v) \in [0, 1]$ is the fraction of node u's quorum slices containing node v. When node u is selecting a new leader, it only considers neighbors, defined as follows:

$$\text{neighbors}(u) = \{\, v \mid H_0(v) < h_{\max} \cdot \text{weight}(u, v) \,\}$$

A node u then starts with an empty set of leaders, and at each round adds to it the node v in neighbors(u) with the highest priority(v). If the neighbors set is empty in any round, u instead adds the node v with lowest value of $H_0(v)/\text{weight}(u, v)$.