Network Working Group
Internet-Draft
Intended status: Experimental
Expires: May 8, 2019

N. Barry
Stellar Development Foundation
G. Losa
UCLA
D. Mazieres
Stanford University
J. McCaleb
Stellar Development Foundation
S. Polu
Stripe Inc.
November 4, 2018

The Stellar Consensus Protocol (SCP) draft-mazieres-dinrg-scp-05

Abstract

SCP is an open Byzantine agreement protocol resistant to Sybil attacks. It allows Internet infrastructure stakeholders to reach agreement on a series of values without unanimous agreement on what constitutes the set of important stakeholders. A big differentiator from other Byzantine agreement protocols is that, in SCP, nodes determine the composition of quorums in a decentralized way: each node selects sets of nodes it considers large or important enough to speak for the whole network, and a quorum must contain such a set for each of its members.

Status of This Memo

This Internet-Draft is submitted in full conformance with the provisions of BCP 78 and BCP 79.

Internet-Drafts are working documents of the Internet Engineering Task Force (IETF). Note that other groups may also distribute working documents as Internet-Drafts. The list of current Internet-Drafts is at https://datatracker.ietf.org/drafts/current/.

Internet-Drafts are draft documents valid for a maximum of six months and may be updated, replaced, or obsoleted by other documents at any time. It is inappropriate to use Internet-Drafts as reference material or to cite them other than as "work in progress."

This Internet-Draft will expire on May 8, 2019.

Table of Contents

1. Introduction 2
2. The Model 3
2.1. Slice infrastructures 3
2.2. Input and output 4
3. Protocol 5
3.1. Federated voting 5
3.2. Basic types 8
3.3. Quorum slices 9
3.4. Nominate message 10
3.5. Ballots 12
3.6. Prepare message 13
3.7. Commit message 17
3.8. Externalize message 18
3.9. Summary of phases 19
3.10. Message envelopes 20
4. Security considerations 21
5. Acknowledgments 21
6. References 22
6.1. Normative References 22
6.2. Informative References 22
Authors' Addresses 23

1. Introduction

Various aspects of Internet infrastructure depend on irreversible and transparent updates to data sets such as authenticated mappings [I-D.watson-dinrg-delmap]. Examples include public key certificates and revocations, transparency logs [RFC6962], preload lists for HSTS [RFC6797] and HPKP [RFC7469], and IP address delegation [I-D.paillisse-sidrops-blockchain].

The Stellar Consensus Protocol (SCP) specified in this draft allows Internet infrastructure stakeholders to collaborate in applying irreversible transactions to public state. SCP is an open Byzantine agreement protocol that resists Sybil attacks by allowing individual parties to specify minimum quorum memberships in terms of specific trusted peers. Each participant chooses combinations of peers on which to depend such that these combinations can be trusted in aggregate. The protocol guarantees safety so long as these dependency sets transitively overlap and contain sufficiently many honest nodes correctly obeying the protocol.

Though bad configurations are theoretically possible, several analogies provide an intuition for why transitive dependencies overlap in practice. For example, given multiple entirely disjoint Internet-protocol networks, people would have no trouble agreeing on the fact that the network containing the world's top web sites is _the_ Internet. Such a consensus can hold even without unanimous agreement on what constitute the world's top web sites. Similarly, if network operators listed all the ASes from whom they would consider peering or transit worthwhile, the transitive closures of these sets would contain significant overlap, even without unanimous agreement on the "tier-1 ISP" designation. Finally, while different browsers and operating systems have slightly different lists of valid certificate authorities, there is significant overlap in the sets, so that a hypothetical system requiring validation from "all CAs" would be unlikely to diverge.

A more detailed abstract description of SCP and its rationale, including an English-language proof of safety, is available in [SCP]. In particular, that reference shows that a necessary property for safety, termed _quorum intersection despite ill-behaved nodes_, is sufficient to guarantee safety under SCP, making SCP optimally safe against Byzantine node failure for any given configuration.

This document specifies the end-system logic and wire format of the messages in SCP.

2. The Model

This section describes the configuration and input/output values of the consensus protocol.

2.1. Slice infrastructures

The SCP protocol achieves consensus on what we call a _slice infrastructure_, defined by a set of _nodes_ and and, for each node, a set of _quorum slices_ that determine quorum membership in a

decentralized way. Each _node_ in has a digital signature key and is named by the corresponding public key, which we term a "NodeID".

Each node choses one or more quorum slices, which are sets of nodes that all include the node itself. A quorum slice represents a large or important enough set of peers that the node selecting the quorum slice believes the slice collectively speaks for the whole network.

A _quorum_ is a non-empty set of nodes containing at least one quorum slice of each of its members. For instance, suppose "v1" has the single quorum slice "{v1, v2, v3}", while each of "v2", "v3", and "v4" has the single quorum slice "{v2, v3, v4}". In this case, "{v2, v3, v4}" is a quorum because it contains a slice for each member. On the other hand "{v1, v2, v3}" is not a quorum, because it does not contain a quorum slice for "v2" or "v3". The smallest quorum including "v1" in this example is the set of all nodes "{v1, v2, v3, v4}".

Unlike traditional Byzantine agreement protocols, nodes in SCP only care about quorums to which they belong themselves (and hence that contain at least one of their quorum slices). Intuitively, this is what protects nodes from Sybil attacks. In the example above, if "v3" deviates from the protocol, maliciously inventing 96 Sybils "v5, v6, ..., v100", the honest nodes' quorums will all still include one another, ensuring that "v1", "v2", and "v4" continue to agree on output values.

Every message in the SCP protocol specifies the sender's quorum slices. Hence, by collecting messages, a node dynamically learns what constitutes a quorum and can decide when a particular message has been sent by a quorum to which it belongs. (Again, nodes do not care about quorums to which they do not belong themselves.)

2.2. Input and output

SCP produces a series of output _values_ for consecutively numbered _slots_. At the start of a slot, higher-layer software on each node supplies a candidate input value. Nodes then exchange protocol messages to agree on one or a combination of nodes' input values as the slot's output value. After a pause to assemble new input values, the process repeats for the next slot, with a 5-second interval between slots.

A value typically encodes a set of actions to apply to a replicated state machine. During the pause between slots, nodes accumulate the next set of actions, amortizing the cost of consensus on one slot over arbitrarily many individual state machine operations.

In practice, only one or a small number of nodes' input values actually affect the output value for any given slot. As discussed in Section 3.4, which nodes' input values to use depends on a cryptographic hash of the slot number and node public keys. A node's chances of affecting the output value depend on how often it appears in other nodes' quorum slices.

From SCP's perspective, values are just opaque byte arrays whose interpretation is left to higher-layer software. However, SCP requires a _validity_ function (to check whether a value is valid) and a _combining function_ that reduces multiple candidate values into a single _composite_ value. When nodes nominate multiple values for a slot, SCP nodes invoke this function to converge on a single composite value. By way of example, in an application where values consist of sets of transactions, the combining function could take the union of transaction sets. Alternatively, if values represent a timestamp and a set of transactions, the combining function might pair the highest nominated timestamp with the transaction set that has the highest hash value.

3. Protocol

The protocol consists of exchanging digitally-signed messages bound to nodes' quorum slices. The format of all messages is specified using XDR [RFC4506]. In addition to quorum slices, messages compactly convey votes on sets of conceptual statements. The core technique of voting with quorum slices is termed _federated voting_. We describe federated voting next, then detail protocol messages in the subsections that follow.

The protocol goes through four phases: NOMINATE, PREPARE, COMMIT, and EXTERNALIZE. The NOMINATE and PREPARE phases run concurrently (though NOMINATE's messages are sent earlier and it ends before PREPARE ends). The COMMIT and EXTERNALIZE phases are exclusive, with COMMIT occurring immediately after PREPARE and EXTERNALIZE immediately after COMMIT.

3.1. Federated voting

Federated voting is a process through which nodes _confirm_ statements. Not every attempt at federated voting may succeed--an attempt to vote on some statement "a" may get stuck, with the result that nodes can confirm neither "a" nor its negation "!a". However, when a node succeeds in confirming a statement "a", federated voting guarantees two things:

1. No two well-behaved nodes will confirm contradictory statements in any configuration and failure scenario in which any protocol

can guarantee safety for the two nodes (i.e., quorum intersection for the two nodes holds despite ill-behaved nodes).

2. If a quorum "I" is guaranteed safety by #1 even when all nodes in "!I" are malicious, and one node in "I" confirms a statement "a", then eventually every member of "I" will also confirm "a".

Intuitively, these conditions are key to ensuring agreement among nodes as well as a weak form of liveness (the non-blocking property [building-blocks]) that is compatible with the FLP impossibility result [FLP].

As a node "v" collects signed copies of a federated voting message "m" from peers, two thresholds trigger state transitions in "v" depending on the message. We define these thresholds as follows:

- _quorum threshold_: When every member of a quorum to which "v" belongs (including "v" itself) has issued message "m"

- _blocking threshold_: When at least one member of each of "v"'s quorum slices (a set that does not necessarily include "v" itself) has issued message "m"

Each node "v" can send several types of message with respect to a statement "a" during federated voting:

- _vote_ "a" states that "a" is a valid statement and constitutes a promise by "v" not to vote for any contradictory statement, such as "!a".

- _accept_ "a" says that nodes may or may not come to agree on "a", but if they don't, then the system has experienced a catastrophic set of Byzantine failures to the point that no quorum containing "v" consists entirely of correct nodes. (Nonetheless, accepting "a" is not sufficient to act on it, as doing so could violate agreement, which is worse than merely getting stuck from lack of a correct quorum.)

- _vote-or-accept_ "a" is the disjunction of the above two messages. A node implicitly sends such a message if it sends either _vote_ "a" or _accept_ "a". Where it is inconvenient and unnecessary to differentiate between _vote_ and _accept_, a node can explicitly send a _vote-or-accept_ message.

- _confirm_ "a" indicates that _accept_ "a" has reached quorum threshold at the sender. This message is interpreted the same as _accept_ "a", but allows recipients to optimize their quorum

checks by ignoring the sender's quorum slices, as the sender asserts it has already checked them.

Figure 1 illustrates the federated voting process. A node "v" votes for a valid statement "a" that doesn't contradict statements in past _vote_ or _accept_ messages sent by "v". When the _vote_ message reaches quorum threshold, the node accepts "a". In fact, "v" accepts "a" if the _vote-or-accept_ message reaches quorum threshold, as some nodes may accept "a" without first voting for it. Specifically, a node that cannot vote for "a" because it has voted for "a"'s negation "!a" still accepts "a" when the message _accept_ "a" reaches blocking threshold (meaning assertions about "!a" have no hope of reaching quorum threshold barring catastrophic Byzantine failure).

If and when the message _accept_ "a" reaches quorum threshold, then "v" has confirmed "a" and the federated vote has succeeded. In effect, the _accept_ messages constitute a second vote on the fact that the initial vote messages succeeded. Once "v" enters the confirmed state, it may issue a _confirm_ "a" message to help other nodes confirm "a" more efficiently by pruning their quorum search at "v".

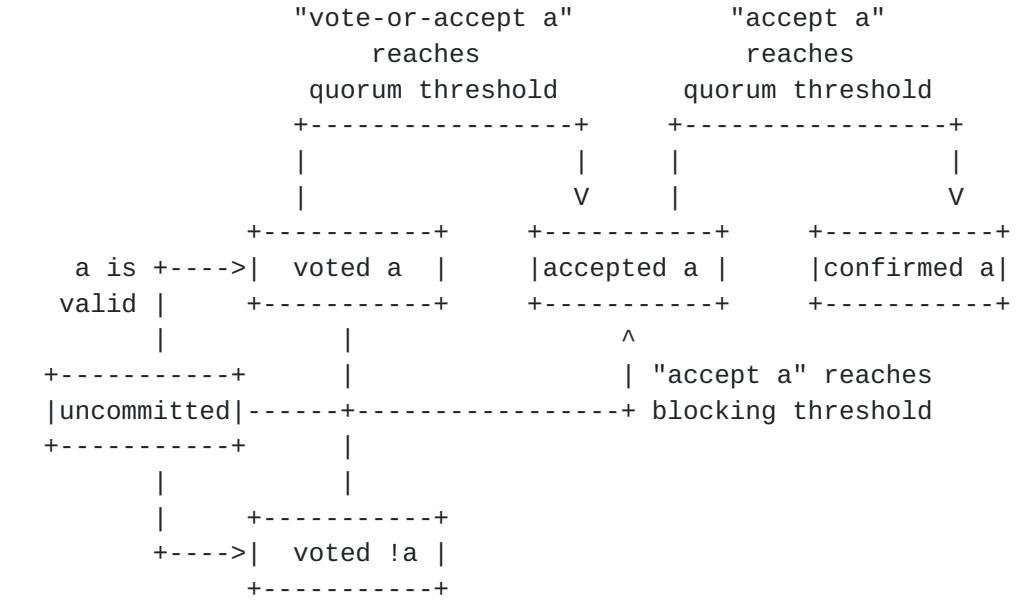

Figure 1: Federated voting process

Note several important invariants. A node may not vote for two contradictory statements or accept two contradictory statements. Moreover, a node may not vote for a statement that contradicts a message it has already accepted (which could lead to accepting a contradictory statement). However, a node is allowed to vote for one

statement and then accept a contradictory statement when a blocking threshold of accept messages contradicts the vote.

3.2. Basic types

SCP employs 32- and 64-bit integers, as defined below.

```
typedef unsigned int uint32;
typedef int int32;
typedef unsigned hyper uint64;
typedef hyper int64;
```

SCP uses the SHA-256 cryptograhpic hash function [RFC6234], and represents hash values as a simple array of 32 bytes.

```
typedef opaque Hash[32];
```

SCP employs the Ed25519 digital signature algorithm [RFC8032]. For cryptographic agility, however, public keys are represented as a union type that can later be compatibly extended with other key types.

```
typedef opaque uint256[32];

enum PublicKeyType
{
    PUBLIC_KEY_TYPE_ED25519 = 0
};

union PublicKey switch (PublicKeyType type)
{
case PUBLIC_KEY_TYPE_ED25519:
    uint256 ed25519;
};

// variable size as the size depends on the signature scheme used
typedef opaque Signature<64>;
```

Nodes are public keys, while values are simply opaque arrays of bytes.

```
typedef PublicKey NodeID;

typedef opaque Value<>;
```

3.3. Quorum slices

Theoretically a quorum slice can be an arbitrary set of nodes. However, arbitrary predicates on sets cannot be encoded concisely. Instead we specify quorum slices as any set of k-of-n members, where each of the n members can either be an individual node ID, or, recursively, another k-of-n set.

```
// supports things like: A,B,C,(D,E,F),(G,H,(I,J,K,L))
// only allows 2 levels of nesting
struct SCPSlices
{
    uint32 threshold;           // the k in k-of-n
    PublicKey validators<>;
    SCPSlices1 innerSets<>;
};
struct SCPSlices1
{
    uint32 threshold;           // the k in k-of-n
    PublicKey validators<>;
    SCPSlices2 innerSets<>;
};
struct SCPSlices2
{
    uint32 threshold;           // the k in k-of-n
    PublicKey validators<>;
};
```

Let "k" be the value of "threshold" and "n" the sum of the sizes of the "validators" and "innerSets" vectors in a message sent by some node "v". A message "m" sent by "v" reaches quorum threshold at "v" when three things hold:

1. "v" itself has issued (digitally signed) the message,

2. The number of nodes in "validators" who have signed "m" plus the number of "innerSets" that (recursively) meet this condition is at least "k", and

3. These three conditions apply (recursively) at some combination of nodes sufficient for condition #2.

A message reaches blocking threshold at "v" when the number of "validators" making the statement plus (recursively) the number "innerSets" reaching blocking threshold exceeds "n-k". (Blocking threshold depends only on the local node's quorum slices and hence does not require a recursive check on other nodes like step #3 above.)

As described in Section 3.10, every protocol message is paired with a cryptographic hash of the sender's "SCPSlices" and digitally signed. Inner protocol messages described in the next few sections should be understood to be received alongside such a quorum slice specification and digital signature.

3.4. Nominate message

For each slot, the SCP protocol begins in a NOMINATE phase, whose goal is to devise one or more candidate output values for the consensus protocol. In this phase, nodes send nomination messages comprising a monotonically growing set of values:

```
struct SCPNominate
{
    Value voted<>;      // X
    Value accepted<>;   // Y
};
```

The "voted" and "accepted" sets are disjoint; any value that is eligible for both sets is placed only in the "accepted" set.

"voted" consists of candidate values that the sender has voted to nominate. Each node progresses through a series of nomination _rounds_ in which it may increase the set of values in its own "voted" field by adding the contents of the "voted" and "accepted" fields of "SCPNominate" messages received from a growing set of peers. In round "n" of slot "i", each node determines an additional peer whose nominated values it should incorporate in its own "SCPNominate" message as follows:

- Let "Gi(m) = SHA-256(i || m)", where "||" denotes the concatenation of serialized XDR values. Treat the output of "Gi" as a 256-bit binary number in big-endian format.

- For each peer "v", define "weight(v)" as the fraction of quorum slices containing "v".

- Define the set of nodes "neighbors(n)" as the set of nodes v for which "Gi(1 || n || v) < 2^{256} * weight(v)", where "1" and "n" are both 32-bit XDR "int" values. Note that a node is always its own neighbor because conceptually a node belongs to all of its own quorum slices.

- Define "priority(n, v)" as "Gi(2 || n || v)", where "2" and "n" are both 32-bit XDR "int" values.

For each round "n" until nomination has finished (see below), a node starts _echoing_ the available peer "v" with the highest value of "priority(n, v)" from among the nodes in "neighbors(n)". To echo "v", the node merges any valid values from "v"'s "voted" and "accepted" sets into its own "voted" set.

XXX - expand "voted" with only the 10 values with lowest Gi hash in any given round to avoid blowing out the message size?

Note that when echoing nominations, nodes must exclude and neither vote for nor accept values rejected by the higher-layer application's validity function. This validity function must not depend on state that can permanently differ across nodes. By way of example, it is okay to reject values that are syntactically ill-formed, that are semantically incompatible with the previous slot's value, that contain invalid digital signatures, that contain timestamps in the future, or that specify upgrades to unknown versions of the protocol. By contrast, the application cannot reject values that are incompatible with the results of a DNS query or some dynamically retrieved TLS certificate, as different nodes could see different results when doing such queries.

Nodes must not send an "SCPNominate" message until at least one of the "voted" or "accepted" fields is non-empty. When these fields are both empty, a node that has the highest priority among its neighbors in the current round (and hence should be echoing its own votes) adds the higher-layer software's input value to its "voted" field. Nodes that do not have the highest priority wait to hear "SCPNominate" messages from the nodes whose nominations they are echoing.

If a particular valid value "x" reaches quorum threshold in the messages sent by peers (meaning that every node in a quorum contains "x" either in the "voted" or the "accepted" field), then the node at which this happens moves "x" from its "voted" field to its "accepted" field and broadcasts a new "SCPNominate" message. Similarly, if "x" reaches blocking threshold in a node's peers' "accepted" field (meaning every one of a node's quorum slices contains at least one node with "x" in its "accepted" field), then the node adds "x" to its own "accepted" field (removing it from "voted" if applicable). These two cases correspond to the two conditions for entering the "accepted" state in Figure 1.

A node stops adding any new values to its "voted" set as soon as any value "x" reaches quorum threshold in the "accepted" fields of received "SCPNominate" messages. Following the terminology of Section 3.1, this condition corresponds to when the node confirms "x" as nominated. Note, however, that the node continues adding new values to "accepted" as appropriate. Doing so may lead to more

values becoming confirmed nominated even after the "voted" set is closed to new values.

A node always begins nomination in round "1". Round "n" lasts for "1+n" seconds, after which, if no value has been confirmed nominated, the node proceeds to round "n+1". A node continues to echo votes from the highest priority neighbor in prior rounds as well as the current round. In particular, until any value is confirmed nominated, a node continues expanding its "voted" field with values nominated by highest priority neighbors from prior rounds even when the values appeared after the end of those prior rounds.

As defined in the next two sections, the NOMINATE phase ends when a node has confirmed "prepare(b)" for some any ballot "b", as this is the point at which the nomination outcome no longer influences the protocol. Until this point, a node must continue to transmit "SCPNominate" messages as well as to expand its "accepted" set (even if "voted" is closed because some value has been confirmed nominated).

3.5. Ballots

Once there is a candidate on which to try to reach consensus, a node moves through three phases of balloting: PREPARE, COMMIT, and EXTERNALIZE. Balloting employs federated voting to chose between _commit_ and _abort_ statements for ballots. A ballot is a pair consisting of a counter and candidate value:

```
// Structure representing ballot <n, x>
struct SCPBallot
{
    uint32 counter; // n
    Value value;    // x
};
```

We use the notation "<n, x>" to represent a ballot with "counter == n" and "value == x".

Ballots are totally ordered with "counter" more significant than "value". Hence, we write "b1 < b2" to mean that either "(b1.counter < b2.counter)" or "(b1.counter == b2.counter && b1.value < b2.value)". Values are compared lexicographically as a strings of unsigned octets.

The protocol moves through federated voting on successively higher ballots until nodes confirm "commit(b)" for some ballot "b", at which point consensus terminates and outputs "b.value" for the slot. To ensure that only one value can be chosen for a slot and that the

protocol cannot get stuck if individual ballots get stuck, there are two restrictions on voting:

1. A node cannot vote for both "commit(b)" and "abort(b)" on the same ballot (the two outcomes are contradictory), and

2. A node may not vote for or accept "commit(b)" for any ballot "b" unless it has confirmed "abort" for every lesser ballot with a different value or already accepted "commit(b')" for some "b' < b" with "b'.value == b.value".

The second condition requires voting to abort large numbers of ballots before voting to commit a ballot "b". We call this _preparing_ ballot "b", and introduce the following notation for the associated set of abort statements.

- "prepare(b)" encodes an "abort" statement for every ballot less than "b" containing a value other than "b.value", i.e., "prepare(b) = { abort(b1) | b1 < b AND b1.value != b.value }".

- "vote prepare(b)" stands for a set of _vote_ messages for every "abort" statement in "prepare(b)".

- Similarly, "accept prepare(b)", "vote-or-accept prepare(b)", and "confirm prepare(b)" encode sets of _accept_, _vote-or-accept_, and _confirm_ messages for every "abort" statement in "prepare(b)".

Using this terminology, a node must confirm "prepare(b)" before issuing a _vote_ or _accept_ message for the statement "commit(b)".

3.6. Prepare message

The first phase of balloting is the PREPARE phase. During this phase, as soon as a node has a valid candidate value (see the rules for "ballot.value" below), it begins sending the following message:

```
struct SCPPrepare
{
    SCPBallot ballot;        // current & highest prepare vote
    SCPBallot *prepared;     // highest accepted prepared ballot
    uint32 aCounter;         // lowest non-aborted ballot counter or 0
    uint32 hCounter;         // h.counter or 0 if h == NULL
    uint32 cCounter;         // c.counter or 0 if !c || !hCounter
};
```

This message compactly conveys the following (conceptual) federated voting messages:

- "vote-or-accept prepare(ballot)"
- If "prepared != NULL": "accept prepare(prepared)"
- If "aCounter != 0": "accept abort(b)" for every "b" with "b.counter < aCounter"
- If "hCounter != 0": "confirm prepare(<hCounter, ballot.value>)"
- If "cCounter != 0": "vote commit(<n, ballot.value>)" for every "cCounter <= n <= hCounter"

Note that to be valid, an "SCPPrepare" message must satisfy the following conditions:

- If "prepared != NULL", then "prepared <= ballot" and "aCounter <= prepared.counter",
- If "prepared == NULL", then "aCounter == 0", and
- "cCounter <= hCounter <= ballot.counter".

Based on the federated vote messages received, each node keeps track of what ballots have been accepted and confirmed prepared. It uses these ballots to set the following fields of its own "SCPPrepare" messages as follows.

ballot
: The current ballot that a node is attempting to prepare and commit. The rules for setting each field are detailed below. Note that the "value" is updated when and only when "counter" changes.

ballot.counter
: The counter is set according to the following rules:

  * Upon entering the PREPARE phase, the "counter" field is initialized to 1.
  * When a node sees messages from a quorum to which it belongs such that each message's "ballot.counter" is greater than or equal to the local "ballot.counter", the node arms a timer to fire in a number of seconds equal to its "ballot.counter + 1" (so the timeout lengthens linearly as the counter increases). Note that for the purposes of determining whether a quorum has a particular "ballot.counter", a node considers "ballot" fields in "SCPPrepare" and "SCPCommit" messages. It also considers

"SCPExternalize" messages to convey an implicit "ballot.counter" of "infinity".

* If the timer fires, a node increments the ballot counter by 1.

* If nodes forming a blocking threshold all have "ballot.counter" values greater than the local "ballot.counter", then the local node immediately cancels any pending timer, increases "ballot.counter" to the lowest value such that this is no longer the case, and if appropriate according to the rules above arms a new timer. Note that the blocking threshold may include ballots from "SCPCommit" messages as well as "SCPExternalize" messages, which implicitly have an infinite ballot counter.

* *Exception*: To avoid exhausting "ballot.counter", its value must always be less then 1,000 plus the number of seconds a node has been running SCP on the current slot. Should any of the above rules require increasing the counter beyond this value, a node either increases "ballot.counter" to the maximum permissible value, or, if it is already at this maximum, waits up to one second before increasing the value.

ballot.value

Each time the ballot counter is changed, the value is also recomputed as follows:

* If any ballot has been confirmed prepared, then "ballot.value" is taken to to be "h.value" for the highest confirmed prepared ballot "h". (Note that once this is the case, the node can stop sending "SCPNominate" messages, as "h.value" supersedes any output of the nomination protocol.)

* Otherwise (if no such "h" exists), if one or more values are confirmed nominated, then "ballot.value" is taken as the output of the deterministic combining function applied to all confirmed nominated values. Note that because the NOMINATE and PREPARE phases run concurrently, the set of confirmed nominated values may continue to grow during balloting, changing "ballot.value" even if no ballots are confirmed prepared.

* Otherwise, if no ballot is confirmed prepared and no value is confirmed nominated, but the node has accepted a ballot prepared (because "prepare(b)" meets blocking threshold for some ballot "b"), then "ballot.value" is taken as the value of the highest such accepted prepared ballot.

* Otherwise, if no value is confirmed nominated and no value is accepted prepared, then a node cannot yet send an "SCPPrepare" message and must continue sending only "SCPNominate" messages.

prepared
: The highest accepted prepared ballot not exceeding the "ballot" field, or NULL if no ballot has been accepted prepared. Recall that ballots with equal counters are totally ordered by the value. Hence, if "ballot = <n, x>" and the highest prepared ballot is "<n, y>" where "x < y", then the "prepared" field in sent messages must be set to "<n-1, y>" instead of "<n, y>", as the latter would exceed "ballot". In the event that "n = 1", the prepared field may be set to "<0, y>", meaning 0 is a valid "prepared.counter" even though it is not a valid "ballot.counter". It is possible to confirm "prepare(<0, y>)", in which case the next "ballot.value" is set to "y". However, it is not possible to vote to commit a ballot with counter 0.

aCounter
: The lowest counter such that all ballots with lower counters have been accepted aborted. This value is set whenever "prepared.value" changes, since the definition of prepare implies that all ballots below the lesser of two prepared ballots have been aborted. Specifically, if the value of "prepared" just changed from "oldPrepared" where "prepared.value != oldPrepared.value", then "aCounter" is set to "oldPrepared.counter" if "oldPrepared.value < prepared.value", and "oldPrepared.counter+1" otherwise.

hCounter
: If "h" is the highest confirmed prepared ballot and "h.value == ballot.value", then this field is set to "h.counter". Otherwise, if no ballot is confirmed prepared or if "h.value != ballot.value", then this field is 0. Note that by the rules above, if "h" exists, then "ballot.value" will be set to "h.value" the next time "ballot" is updated.

cCounter
: The value "cCounter" is maintained based on an internally-maintained _commit ballot_ "c", initially "NULL". "cCounter" is 0 while "c == NULL" or "hCounter == 0", and is "c.counter" otherwise. "c" is updated as follows:

  * If either "(prepared > c && prepared.value != c.value)" or "(aCounter > c.counter)", then reset "c = NULL".

  * If "c == NULL" and "hCounter == ballot.counter" (meaning "ballot" is confirmed prepared), then set "c" to "ballot".

Note these rules preserve the invariant that a node cannot vote for contradictory statements (namely committing and aborting the same ballot) by conservatively assuming a node may have voted to abort anything below "ballot". Hence, whenever "c" changes, it can either change to "NULL" or to "ballot", but is never set to anything below the current "ballot".

A node leaves the PREPARE phase and proceeds to the COMMIT phase when there is some ballot "b" for which the node confirms "prepare(b)" and accepts "commit(b)". (If nodes never changed quorum slice mid-protocol, it would suffice to accept "commit(b)". Also waiting to confirm "prepare(b)" makes it easier to recover from liveness failures by removing Byzantine faulty nodes from quorum slices.)

3.7. Commit message

In the COMMIT phase, a node has accepted "commit(b)" for some ballot "b", and must confirm that statement to act on the value in "b.counter". A node sends the following message in this phase:

```
struct SCPCommit
{
    SCPBallot ballot;       // b
    uint32 preparedCounter; // prepared.counter
    uint32 hCounter;        // h.counter
    uint32 cCounter;        // c.counter
};
```

The message conveys the following federated vote messages, where "infinity" is 2^{32} (a value greater than any ballot counter representable in serialized form):

- "accept commit(<n, ballot.value>)" for every "cCounter <= n <= hCounter"
- "vote-or-accept prepare(<infinity, ballot.value>)"
- "accept prepare(<preparedCounter, ballot.value>)"
- "confirm prepare(<hCounter, ballot.value>)"
- "vote commit(<n, ballot.value>)" for every "n >= cCounter"

A node computes the fields in the "SCPCommit" messages it sends as follows:

ballot

This field is maintained identically to how it is maintained in the PREPARE phase, though "ballot.value" can no longer change, only "ballot.counter". Note that the value "ballot.counter" does not figure in any of the federated voting messages. The purpose of continuing to update and send this field is to assist other nodes still in the PREPARE phase in synchronizing their counters.

preparedCounter
: This field is the counter of the highest accepted prepared ballot--maintained identically to the "prepared" field in the PREPARE phase. Since the "value" field will always be the same as "ballot", only the counter is sent in the COMMIT phase.

cCounter
: The counter of the lowest ballot "c" for which the node has accepted "commit(c)". (No value is included in messages since "c.value == ballot.value".)

hCounter
: The counter of the highest ballot "h" for which the node has accepted "commit(h)". (No value is included in messages since "h.value == ballot.value".)

As soon as a node confirms "commit(b)" for any ballot "b", it moves to the EXTERNALIZE phase.

3.8. Externalize message

A node enters the EXTERNALIZE phase when it confirms "commit(b)" for any ballot "b". As soon as this happens, SCP outputs "b.value" as the value of the current slot. In order to help other nodes achieve consensus on the slot more quickly, a node reaching this phase also sends the following message:

```
struct SCPExternalize
{
    SCPBallot commit;         // c
    uint32 hCounter;          // h.counter
};
```

An "SCPExternalize" message conveys the following federated voting messages:

- "accept commit(<n, commit.value>)" for every "n >= commit.counter"

- "confirm commit(<n, commit.value>)" for every "commit.counter <= n <= hCounter"

- "accept prepare(<infinity, commit.value>)"

- "confirm prepare(<hCounter, commit.value>)"

The fields are set as follows:

commit
: The lowest confirmed committed ballot.

hCounter
: The counter of the highest confirmed committed ballot.

3.9. Summary of phases

Table 1 summarizes the phases of SCP for each slot. The NOMINATE and PREPARE phases begin concurrently. However, a node initially does not send "SCPPrepare" messages but only listens for ballot messages in case "accept prepare(b)" reaches blocking threshold for some ballot "b". The COMMIT and EXTERNALIZE phases then run in turn after PREPARE ends. A node may externalize (act upon) a value as soon as it enters the EXTERNALIZE phase.

The point of "SCPExternalize" messages is to help straggling nodes catch up more quickly. As such, the EXTERNALIZE phase never ends. Rather, a node should archive an "SCPExternalize" message for as long as it retains slot state.

Phase	Begin	End
NOMINATE	previous slot externalized and 5 seconds have elapsed since NOMINATE ended for that slot	some ballot is confirmed prepared
PREPARE	begin with NOMINATE, but send "SCPPrepare" only once some value confirmed nominated or accept "prepare(b)" for some ballot b	accept "commit(b)" for some ballot "b"
COMMIT	accept "commit(b)" for some ballot "b"	confirm "commit(b)" for some ballot "b"
EXTERNALIZE	confirm "commit(b)" for some ballot "b"	slot state garbage-collected

Table 1: Phases of SCP for a slot

3.10. Message envelopes

In order to provide full context for each signed message, all signed messages are part of an "SCPStatement" union type that includes the "slotIndex" naming the slot to which the message applies, as well as the "type" of the message. A signed message and its signature are packed together in an "SCPEnvelope" structure.

```
enum SCPStatementType
{
    SCP_ST_PREPARE = 0,
    SCP_ST_COMMIT = 1,
    SCP_ST_EXTERNALIZE = 2,
    SCP_ST_NOMINATE = 3
};

struct SCPStatement
{
    NodeID nodeID;      // v (node signing message)
    uint64 slotIndex;   // i
    Hash quorumSetHash; // hash of serialized SCPSlices

    union switch (SCPStatementType type)
    {
    case SCP_ST_PREPARE:
        SCPPrepare prepare;
    case SCP_ST_COMMIT:
        SCPCommit commit;
    case SCP_ST_EXTERNALIZE:
        SCPExternalize externalize;
    case SCP_ST_NOMINATE:
        SCPNominate nominate;
    }
    pledges;
};

struct SCPEnvelope
{
    SCPStatement statement;
    Signature signature;
};
```

4. Security considerations

If nodes do not pick quorum slices well, the protocol will not be safe.

5. Acknowledgments

The Stellar development foundation supported development of the protocol and produced the first production deployment of SCP. The IRTF DIN group including Dirk Kutscher, Sydney Li, Colin Man, Piers Powlesland, Melinda Shore, and Jean-Luc Watson helped with the framing and motivation for this specification. The mobilecoin team contributed the "aCounter" optimization. We also thank Bob

Glickstein for finding bugs in drafts of this document and offering many useful suggestions.

6. References

6.1. Normative References

[RFC4506] Eisler, M., Ed., "XDR: External Data Representation Standard", STD 67, RFC 4506, DOI 10.17487/RFC4506, May 2006, <https://www.rfc-editor.org/info/rfc4506>.

[RFC6234] Eastlake 3rd, D. and T. Hansen, "US Secure Hash Algorithms (SHA and SHA-based HMAC and HKDF)", RFC 6234, DOI 10.17487/RFC6234, May 2011, <https://www.rfc-editor.org/info/rfc6234>.

[RFC8032] Josefsson, S. and I. Liusvaara, "Edwards-Curve Digital Signature Algorithm (EdDSA)", RFC 8032, DOI 10.17487/RFC8032, January 2017, <https://www.rfc-editor.org/info/rfc8032>.

6.2. Informative References

[building-blocks] Song, Y., van Renesse, R., Schneider, F., and D. Dolev, "The Building Blocks of Consensus", 9th International Conference on Distributed Computing and Networking pp. 54-72, 2008.

[FLP] Fischer, M., Lynch, N., and M. Lynch, "Impossibility of Distributed Consensus with One Faulty Process", Journal of the ACM 32(2):374-382, 1985.

[I-D.paillisse-sidrops-blockchain] Paillisse, J., Rodriguez-Natal, A., Ermagan, V., Maino, F., Vegoda, L., and A. Cabellos-Aparicio, "An analysis of the applicability of blockchain to secure IP addresses allocation, delegation and bindings.", draft-paillisse-sidrops-blockchain-02 (work in progress), June 2018.

[I-D.watson-dinrg-delmap] Watson, J., Li, S., and C. Man, "Delegated Distributed Mappings", draft-watson-dinrg-delmap-01 (work in progress), October 2018.

[RFC6797] Hodges, J., Jackson, C., and A. Barth, "HTTP Strict Transport Security (HSTS)", RFC 6797, DOI 10.17487/RFC6797, November 2012, <https://www.rfc-editor.org/info/rfc6797>.

[RFC6962] Laurie, B., Langley, A., and E. Kasper, "Certificate Transparency", RFC 6962, DOI 10.17487/RFC6962, June 2013, <https://www.rfc-editor.org/info/rfc6962>.

[RFC7469] Evans, C., Palmer, C., and R. Sleevi, "Public Key Pinning Extension for HTTP", RFC 7469, DOI 10.17487/RFC7469, April 2015, <https://www.rfc-editor.org/info/rfc7469>.

[SCP] Mazieres, D., "The Stellar Consensus Protocol: A Federated Model for Internet-level Consensus", Stellar Development Foundation whitepaper , 2015, <https://www.stellar.org/papers/stellar-consensus-protocol.pdf>.

Authors' Addresses

Nicolas Barry
Stellar Development Foundation
170 Capp St., Suite A
San Francisco, CA 94110
US

Email: nicolas@stellar.org

Giuliano Losa
UCLA
3753 Keystone Avenue #10
Los Angeles, CA 90034
US

Email: giuliano@cs.ucla.edu

David Mazieres
Stanford University
353 Serra Mall, Room 290
Stanford, CA 94305
US

Email: dm@uun.org

Jed McCaleb
Stellar Development Foundation
170 Capp St., Suite A
San Francisco, CA 94110
US

Email: jed@stellar.org

Stanislas Polu
Stripe Inc.
185 Berry Street, Suite 550
San Francisco, CA 94107
US

Email: stan@stripe.com