Network Working Group
Internet-Draft
Intended status: Standards Track
Expires: March 13, 2021

A. Mayrhofer
nic.at GmbH
D. Klesev

M. Sabadello
Danube Tech GmbH

September 9, 2020

The Decentralized Identifier (DID) in the DNS
draft-mayrhofer-did-dns-04

Abstract

This document specifies the use of the URI Resource Record Type to publish Decentralized Identifiers (DIDs) in the DNS.

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 March 13, 2021.

Table of Contents

1. Introduction

Decentralized Identifiers (DIDs) [W3C-DID] use a Uniform Resource Identifier (URI) scheme [RFC3986] to identify persons, organizations, or things in decentralized infrastructure, such as blockchains and distributed ledgers.

DIDs are structured around "methods", each method defining the syntax of the "method specific identifier" and the operations on the respective DIDs (See Section 3.2 of [W3C-DID] and [DID-METHODS]). For many methods, the method specific identifier is not human-friendly (such as hash values, referring to transactions on a blockchain). Most DIDs are therefore inherently hard to memorize for humans.

By referring to DIDs from the Domain Name System (DNS), those hard to memorize identifiers can be discovered via well known, human friendly and widely established names. This document specifies how DIDs can be published in the DNS for discovery on the base of host names and email addresses.

Since DIDs use a URI scheme ('did'), this specification leverages the existing URI DNS Resource Record Type (RRType) [RFC7553]. Records are scoped using the '_did' global underscore node name, as described in Section 3.1.

2. Terminology

"Owner name", "Priority", "Weight" and "Target" refer to the respective fields of the URI RRType, as specified in Section 4 of RFC 7553.

The key words "MUST", "MUST NOT", "REQUIRED", "SHALL", "SHALL NOT", "SHOULD", "SHOULD NOT", "RECOMMENDED", "NOT RECOMMENDED", "MAY", and "OPTIONAL" in this document are to be interpreted as described in BCP 14 [RFC2119] [RFC8174] when, and only when, they appear in all capitals, as shown here.

3. Use of the 'URI' RRType

DIDs use an URI scheme ('did:'), so the most suitable option to publish DIDs in the DNS is the use of the 'URI' RRType. During the development of this document, various alternatives were considered, see Section 6 for a list.

- When Decentralized Identifiers (DIDs) are published in the DNS, the 'URI' RRType MUST be used.

3.1. Owner Name Scoping, Target

[RFC8552] describes the advantages of scoping an existing RRType over the definition (and complex deployment) of a new RRType. The "URI" RRType is specifically mentioned as one example where scoping is particularly useful (and part of the design).

When DIDs are published in the DNS

- the records MUST be scoped by setting the global (highest-level) underscore name of the URI RRset to '_did' (0x5F 0x64 0x69 0x64),

- and the Target field of all records in the RRset MUST contain a URI of the 'did:' URI scheme.

3.2. Weight, Priority

The semantics of the Weight and Priority fields remain. When a client encounters a DID method it does not support, it SHOULD consider the respective URI "unreachable" for the purpose of record

selection, and proceed to the record with the next-lowest-numbered Priority, in accordance with Section 4.2 of RFC 7553.

4. Location of the Records

4.1. Host Names

In order to discover the set of DIDs associated with a Host Name, a client prepends the given Host Name with the '_did' global underscore name to create the Owner name, and then queries the resulting Query Name for the URI RRType set.

4.2. Email Addresses (Experimental)

To discover DIDs associated with email addresses, the (experimental) model from DNS-Based Authentication of Named Entities (DANE) Bindings for OpenPGP [RFC7929] is used. A client prepares the email address following the procedure outlined in Section 5 in RFC7929 the form the Query Name, but in step 5 MUST use the string '_mailto._did' instead of '_openpgpkey' as the second left-most label. Subsequently, the client performs a DNS query, but MUST use the URI RRType as Query Type (rather than the OPENPGPKEY RRType described in said section).

5. Example

The following example is a URI Resource Record which refers from the host name "example.net" to a Decentralized Identifier using the 'sov' method:

```
_did.example.net.  IN URI 100 10 "did:sov:1234abcd"
```

6. Considered Alternatives

During the development of this document, the following alternatives were considered: A dedicated RRType, TXT records, an Enumservice, Well-Known URIs, direct registration in the Service Name Registry. Using the URI RRType was found to be the option with the least impact on existing specifications and highest interoperability potential. Support for URI RRTypes is widespread in DNS software, which means that implementation and deployment of the proposed protocol should be possible without any changes to underlying infrastructure.

Furthermore, the Identifiers and Discovery Working Group of the Decentralized Identity Foundation (DIF) is considering a .well-known URL based approach to discovering DIDs from web sites.

7. Acknowledgements

Acknowledgements will be added here.

8. IANA Considerations

Per [RFC8552] IANA is requested to add the following entry to the DNS Underscore Global Scoped Entry Registry:

RR Type	_NODE NAME	REFERENCE
URI	_did	{THISRFC}

Table 1: Underscore Global Registry Entry Registration for '_did'

Note to RFC Editor: Please replace the above "{THISRFC}" text with a reference to this document's RFC number.

Note that IANA has already created a provisional URI scheme registration for the 'did:' scheme itself.

9. Security Considerations

Most of the considerations outlined in the base specification of the URI RRType (RFC7553) also apply to the DID use case - particularly the concerns around downgrade attacks when the record is not signed with the help of DNSSEC. Note that the DID resolving process itself (out of scope of this document) can provide additional security information. The "Linked Domain Service Endpoint" of a DID document can be used to back-reference to the Domain which was originally used to discover that DID. Such a "closed loop" (similar to verifying DNS reverse lookups against their corresponding forward lookups) would increase the confidence in non-DNSSEC scenarios.

Including a DID in the DNS allows for correlation of that DID with DNS information (and potentially registration information of that DNS name). Therefore DIDs which are supposed to be private SHOULD NOT be added to the DNS.

10. Changes

[Note to RFC Editors: This whole section is to be removed before publication]

10.1. draft-mayrhofer-did-dns-04

- Reworded "Alternatives"
- Added text about backreference using DID's Linked Domain Service Endpoint.

10.2. draft-mayrhofer-did-dns-03

- Updated DID spec to v1.0 document
- Minor editorial changes to make text more clear.

10.3. draft-mayrhofer-did-dns-02

- Updated attrleaf reference to RFC8552
- Changed author information for D. Klesev
- Added sentence on .well-known discovery scheme

10.4. draft-mayrhofer-did-dns-01

- email addresses further scoped with '_mailto._did'
- Changed protocol registration to attrleaf drafts
- Made clear requirements regarding use of the URI scheme
- Added privacy aspect to security considerations

10.5. draft-mayrhofer-did-dns-00

- Initial version

11. References

11.1. Normative References

[RFC2119] Bradner, S., "Key words for use in RFCs to Indicate Requirement Levels", BCP 14, RFC 2119, DOI 10.17487/RFC2119, March 1997, <https://www.rfc-editor.org/info/rfc2119>.

[RFC7553] Faltstrom, P. and O. Kolkman, "The Uniform Resource Identifier (URI) DNS Resource Record", RFC 7553, DOI 10.17487/RFC7553, June 2015, <https://www.rfc-editor.org/info/rfc7553>.

[RFC8174] Leiba, B., "Ambiguity of Uppercase vs Lowercase in RFC 2119 Key Words", BCP 14, RFC 8174, DOI 10.17487/RFC8174, May 2017, <https://www.rfc-editor.org/info/rfc8174>.

[RFC8552] Crocker, D., "Scoped Interpretation of DNS Resource Records through "Underscored" Naming of Attribute Leaves", BCP 222, RFC 8552, DOI 10.17487/RFC8552, March 2019, <https://www.rfc-editor.org/info/rfc8552>.

[W3C-DID] W3C, W3C., "Decentralized Identifiers (DIDs) v1.0", February 2020, <https://www.w3.org/TR/did-core/>.

11.2. Informative References

[DID-METHODS] W3C, W3C., "DID Method Registry", June 2018, <https://w3c-ccg.github.io/did-method-registry/>.

[RFC3986] Berners-Lee, T., Fielding, R., and L. Masinter, "Uniform Resource Identifier (URI): Generic Syntax", STD 66, RFC 3986, DOI 10.17487/RFC3986, January 2005, <https://www.rfc-editor.org/info/rfc3986>.

[RFC6116] Bradner, S., Conroy, L., and K. Fujiwara, "The E.164 to Uniform Resource Identifiers (URI) Dynamic Delegation Discovery System (DDDS) Application (ENUM)", RFC 6116, DOI 10.17487/RFC6116, March 2011, <https://www.rfc-editor.org/info/rfc6116>.

[RFC6117] Hoeneisen, B., Mayrhofer, A., and J. Livingood, "IANA Registration of Enumservices: Guide, Template, and IANA Considerations", RFC 6117, DOI 10.17487/RFC6117, March 2011, <https://www.rfc-editor.org/info/rfc6117>.

[RFC6335] Cotton, M., Eggert, L., Touch, J., Westerlund, M., and S. Cheshire, "Internet Assigned Numbers Authority (IANA) Procedures for the Management of the Service Name and Transport Protocol Port Number Registry", BCP 165, RFC 6335, DOI 10.17487/RFC6335, August 2011, <https://www.rfc-editor.org/info/rfc6335>.

[RFC7929] Wouters, P., "DNS-Based Authentication of Named Entities (DANE) Bindings for OpenPGP", RFC 7929, DOI 10.17487/RFC7929, August 2016, <https://www.rfc-editor.org/info/rfc7929>.

Authors' Addresses

Alexander Mayrhofer
nic.at GmbH
Karlsplatz 1/2/9
Vienna 1010
Austria

Email: alex.mayrhofer.ietf@gmail.com

Dimitrij Klesev

Email: dimitrij.klesev@gmail.com

Markus Sabadello
Danube Tech GmbH
Annagasse 8/1/8
Vienna 1010
Austria

Email: markus@danubetech.com

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

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".

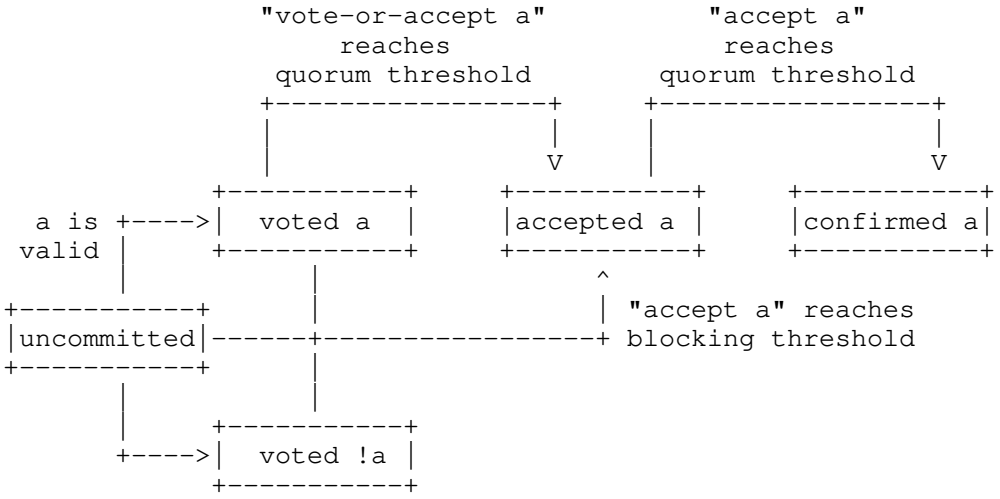

Figure 1: Federated voting process

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

Network Working Group
Internet-Draft
Intended status: Experimental
Expires: October 27, 2019

S. Li
EFF
C. Man
Stanford University
J. Watson
UC Berkeley

April 25, 2019

Delegated Distributed Mappings
draft-watson-dinrg-delmap-02

Abstract

Delegated namespaces underpin almost every Internet-scale system - domain name management, IP address allocation, Public Key Infrastructure, etc. - but are centrally managed by entities with unilateral revocation authority and no common interface. This draft specifies a generalized scheme for delegation that supports explicit time-bound guarantees and limits misuse. Mappings may be secured by any general purpose distributed consensus protocol that supports voting; clients can query the local state of any number of participants and receive the correct result, barring a compromise at the consensus layer.

Table of Contents

1. Introduction

Internet entities rely heavily on delegated namespaces to function properly. Typical web services have been delegated a domain name (after negotitation with an appropriate registrar) under which they host the entirety of their public-facing content, or obtain a public IP range from their ISP, which itself has been delegated to intermediary registries by the Internet Numbers Registry [RFC7249]. An enormous amount of economic value is therefore placed in these assignments (in this draft, _mappings_) yet they are dangerously ephemeral. Delegating authorities, either maliciously or accidentally, may unilaterally revoke or replace mappings they've made, compromising infrastructure security. Presented in this draft is a generalized mechanism for securely managing such mappings and their delegations. Known entities identified by public key are assigned namespaces (e.g. domain prefixes) under which they are

authorized to create mapping records, or _cells_. Cells in a namespace are grouped into logical units we term _tables_.

Alone, this structure does not ensure security, given that any hosting server could arbitrarily modify cells or serve bogus entries to unwitting clients. We maintain security and consistency by relying on a distributed consensus algorithm. While detailed descriptions of varying consensus protocols are out of scope for this draft, we provide for a general-purpose interface between the delegation structure and a consensus layer. At a minimum, the consensus layer must apply mapping updates in a consistent order, prevent equivocation, disallow unauthorized modification, grant consensus nodes the ability to enforce high-level rules associated with the tables, and perform voting among nodes to decide top-level governance. We find that federated protocols such as the Stellar Consensus Protocol [I-D.mazieres-dinrg-scp] are promising given their capability for open participation, broad diversity of interests among consensus participants, and providing accountability for malicious behavior. Clients may query any number of trusted servers to retrieve a correct result, barring widespread collusion.

The ability to impose consistency yields several useful properties. The foremost is enforcing delegation semantics: a table's authority may choose to recursively delegate a portion of its own namespace, but must document the specific range and delegee in one of its table's cells. Since delegation forms a new table, for which a delegee is the sole authority, assigned namespace ranges must be unique. Consensus is also used to enforce that the delegating authority not make modifications to any delegated table and thus need not be trusted by the delegee.

In addition, we provide explicit support for commitments that enforce an explicit lower-bound on the duration of delegations. Otherwise valid changes to cells that have a valid commitment are disallowed, including revoking delegations. Upon expiration, however, the same namespace may be delegated to another party.

Finally, decentralized infrastructure is highly visible and commonly misused. As mappings are replicated among consensus nodes, of primary concern is resource exhaustion. We limit undesired abuse of the structure by embedding recursive scale restrictions inside mappings, verified and ratified at consensus time. Combined with time-bounded delegations, this ensures that the system is resistant to spam in the short-term and can remove misbehaving hierarchies in the long-term.

The remainder of this draft specifies the structure for authenticated mapping management as well as its interfaces to consensus protocol implementations and users.

2. Structure

Trust within the delegation structure is based on public key signatures. Namespace authorities must sign mapping additions, modifications, delegations, and revocations to their table as proof to the consensus participants that such changes are legitimate. For the sake of completeness, the public key and signature types are detailed below. All types in this draft are described in XDR [RFC4506].

```
typedef publickey opaque<>; /* Typically a 256 byte RSA signature */

struct signature {
    publickey pk;
    opaque data<>;
};
```

2.1. Cells

Cells are the basic unit of the delegation structure. In general, they compose an authenticated record of a mapping that may be queried by clients. We describe two types of cells:

```
enum celltype {
    VALUE = 0,
    DELEGATE = 1
};
```

Value cells store individual mapping values. They resolve a lookup key to an arbitrary value, for example, an encryption key associated with an email address or the zone files associated with a particular domain. The public key of the cell's owner (e.g. the email account holder, the domain owner) is also included, as well as a signature authenticating the current version of the cell. Since the cell's contents are controlled by the owner, its "value_sig" must be made by the "owner_key". The cell owner may rotate their public key at any time by signing the update with the old key.

```
struct valuecell {
    opaque value<>;
    publickey owner_key;
    /* Owner signs contents */
    signature value_sig;
};
```

Delegate cells have a similar structure but different semantics. Rather than resolving to an individual mapping, they authorize the _delegee_ to create arbitrary value cells within a table mapped to the assigned namespace. This namespace must be a subset of the _delegator_'s own namespace range. Like the table authority, the delegee is uniquely identified by their public key. Each delegate cell and subsequent updates to the cell are signed by the delegator - this ensures that the delegee cannot unilaterally modify its namespace, which limits the range of mappings they can create to those legitimately assigned to them.

```
struct delegatecell {
    opaque namespace<>;
    publickey delegee;
    /* Table authority controls delegations, not delegee */
    signature delegation_sig;
    unsigned int allowance;
};
```

Both cell types share a set of common data members, namely a set of UNIX timestamps recording the creation time and, if applicable, the time of last modification. An additional commitment timestamp must be present in every cell. Each commitment is an explicit guarantee on behalf of the table's authority that the mapping will remain valid until at least the time specified. Therefore, while value cell owners may modify their cell's contents at any time (e.g. this scheme supports key rotation), the authority cannot change or remove the cell until its commitment expires, as enforced by the consensus nodes. Similarly, delegated namespaces are guaranteed to be valid until the commitment timestamp expiration, although after expiration, they can be reassigned to other parties. Likely, most long-term delegations will be renewed (with a new commitment timestamp) before the expiration of the current period. The tradeoff between protecting delegees from arbitrary authority action and allowing quick delegation reconfiguration is customizable to the use case. Likely, widely-used services will see larger delegation periods for stability whereas small namespaces with relatively unknown delegees will experience shorter delegations.

An _allowance_ must be provided, which limits the upper-bound size of a delegated table. For value cells, the allowance value is ignored and set to 0. Importantly, for delegate cells, an allowance with value 0 indicates no limit is placed on the size of the delegated table. Given that the delegee has complete control over the contents of their table, it is emphatically not recommended to grant a delegatecell an unlimited allowance, in order to limit the storage burden on consensus nodes. This limit is recursive along delegations - the total number of cells in a table plus the sum of allowances

among its "delegatecells" must be less than or equal to the table's allowance, if non-zero. Futher, a table with a non-zero allowance may not grant a delegee an unlimited allowance. These properties must be validated during consensus before adding new cells to a table, which can be performed at every consensus node because table entry counts are visible publicly.

Finally, a valid table cell's timestamps and allowance is signed by the table authority and placed in "authority_sig".

```
union innercell switch (celltype type) {
case VALUE:
    valuecell vcell;
case DELEGATE:
    delegatecell dcell;
};

struct cell {
    /* 64-bit UNIX timestamps */
    unsigned hyper create_time;
    unsigned hyper *revision_time;
    unsigned hyper commitment_time;
    /* Ignored by value cells */
    unsigned int allowance;
    signature authority_sig;
    innercell c;
}
```

2.2. Tables

Every cell is stored in a table, which groups all the mappings created by a single authority public key for a specific namespace. Individual cells are referenced by an application-specific label in a lookup table. _The combination of a lookup key and a referenced cell value forms a mapping_.

```
struct tableentry {
    opaque lookup_key<>;
    cell c;
}
```

Delegating the whole or part of a namespace requires adding a new lookup key for the namespace and a matching delegate cell. Each delegation must be validated in the context of the other table entries and the table itself. For example, the owner of a table delegated an /8 IPv4 block must not to delegate the same /16 block to two different tables.

```
struct table {
    tableentry entries<>;
};
```

2.3. Prefix-based Delegation Correctness

To generalize correctness, each table must conform with a prefix-based rule: for every cell with value or delegation subset "c" in a table controlling namespace "n", "n" must (1) be a prefix of "c" and (2) there cannot exist another cell with value or delegation subset "c2" such that "c" is a prefix of "c2".

While there exist many more hierarchical naming schemes, many can be simply represented in a prefix scheme. For example, suffix-based delegations, including domain name hierarchies, can use reversed keys internally and perform a swap in the application layer before displaying any results to clients. Likewise, 'flat' delegation schemes where there is no explicit restriction can use an empty prefix.

2.4. Root Key Listing

Each linked group of delegation tables for a particular namespace is rooted by a public key stored in a flat root key listing, which is the entry point for lookup operations. Well-known application identifier strings denote the namespace they control. We describe below how lookups can be accomplished on the mappings.

```
struct rootentry {
    publickey namespace_root_key;
    string application_identifier<>;
    signature listing_sig;
    unsigned int allowance;
}

struct rootlisting {
    rootentry roots<>;
}
```

A significant question is how to properly administer entries in this listing, since a strong authority, such as a single root key, can easily protect the listing from spam and malicious changes, but raises important concerns about misuse. Concurrent work on IP address allocation [IP-blockchain] explores using a Decentralized Autonomous Organization built on the Ethereum blockchain to manage all delegations where proper behavior is economically motivated. We identify similar challenges: controlling spam and misuse, while operating in a decentralized manner.

In this draft, however, we focus on enabling governance through consensus operations. For that reason, potential root entries are nominated with a proposed allowance, which will restrict the total number of cells currently supported by an application. For large systems such as IP delegation or well-known entities like the IETF, the limit can be disabled as discussed earlier in this draft. It is important that decisions regarding root listing membership be made by the consensus nodes themselves, since they bear the largest burden to store tables, communicate with other nodes, and service client queries. This structure further allows table authorities to focus on content-specific administration of their own namespaces, which is not provided for in the generic delmap semantics. If an application begins to run out of allowance (too many cells or large delegations), it can sign and nominate a new "rootentry" for the same application identifier with a larger value, at which point the other nodes can (given global knowledge of table sizes and growth rates, along with additional real-world information, if applicable) determine whether or not to accept the change.

Thus, this draft explicitly requires prospective consensus algorithms to provide a mechanism for inter-node voting on governance issues. This is already common in protocols that provide for periodic updates: quorums in slice infrastructures like Stellar, Bitcoin-style percentage- and time-based agreement windows, or hard forks. Finally, although the possibility of fundemental disagreement in governance between different sets of consensus nodes is real, in realistic settings, in the worst case such groups should not necessarily continue to trust each other in consensus relationships and may indicate a (valid) need to split.

3. Interacting with a Consensus Node

3.1. Storage Format

Delegation tables are stored in a Merkle hash tree, described in detail in [RFC6962]. In particular, it enables efficient lookups and logarithmic proofs of existence in the tree, and prevents equivocation between different participants. Among others, we can leverage Google's [Trillian] Merkle tree implementation which generalizes the datastructures used in Certificate Transparency. In map mode, the tree can manage arbitrary key-value pairs at scale, but critically, this requires flattening the delegation links such that each table may be queried, while ensuring that a full lookup from the application root is made for each mapping.

Given a "rootentry", the corresponding table in the Merkle tree can be queried at the following key (where || indicates concatenation):

```
root_table_name = app_id || namespace_root_key
```

It follows that tables for delegated namespaces are found at:

```
table = root_table_name || delegee_key_1 || ... || delegee_key_n
```

And finally, individual entries are identified by the namespace lookup key:

```
cell = table || desired_lookup_key
```

Once an entry is found in the tree, a logarithmic proof can be constructed with the hashes of the siblings of each node in the tree's path to the entry.

```
struct merkleproof {
    opaque sibling_hashes[32]<>;
    cell entry_cell;
    signature tree_sig;
}
```

The entry is hashed together with each "sibling_hash" - if the total matches the known tree root hash, then the entry must have been in the tree.

3.2. Client Interface

The presence of a natural mapping structure motivates an external client interface similar to a key-value store.

```
struct MerkleRootOperation { }

struct MerkleRootReturn {
    opaque root_hash[32];
    signature tree_sig;
}
```

It is important to note that the client should not rely on a root hash that has been provided by a single server to verify a "merkleproof", instead querying multiple consensus nodes using this interface. Upon discovering that different servers are advertising non-matching hashes, the signed proof should be used to prove to other clients/nodes that one or more malicious trees are equivocating.

```
enum ReturnCode {
    CELL = 0,
    TABLE = 1,
    ERROR = 2
}

struct GetOperation {
    string application_identifier;
    opaque full_lookup_key<>;
}

union GetReturn switch (ReturnCode ret) {
case CELL:
    cell value;
    merkleproof p;
case TABLE:
    table t;
    merkleproof p;
case ERROR:
    string reason;
}
```

Given an application identifier and the fully-qualified lookup key, the map described in the previous section can be searched recursively. At each table, we find the cell whose name matches a prefix of the desired lookup key. If the cell contains a "valuecell", it is returned if the cell's key matches the lookup key exactly, else an "ERROR" is returned. If the cell contains a "delegatecell", it must contain the key for the next table, on which the process is repeated. If no cell is found by prefix-matching, the node should return "ERROR" if the key has not been fully found, else the table itself (containing all of the current cells) is provided to the client. As in every interaction with the delegated mapping structure, users should verify the attached proof. Verifying existence of an entry follows from the same method.

```
struct SetOperation {
    string application_identifier;
    opaque full_lookup_key<>;
    cell c;
}

struct SetRootOperation {
    rootentry e;
    bool remove;
}

union SetReturn switch (ReturnCode ret) {
case SUCCESS:
    opaque empty;
case ERROR:
    string reason;
}
```

Creating or updating a cell at a specified path requires once again the full lookup key, as well as the new version of the cell to place. The new cell must be well-formed under the validation checks described in the previous section, else an "ERROR" is returned. For example, updating a cell's owner without a signature by the previous owning key should not succeed. Both value cells and new/updated delegations may be created through this method. Removing cells from tables (after their commitment timestamps have expired) can be accomplished by replacing the value or delegated namespace with an empty value and setting the owner's key to that of the table authority. Asking the consensus layer to approve a new root entry follows a similar process, although the application identifier and lookup key is unnecessary (see "SetRootOperation"). Nodes can also trigger votes to remove entries from the root key listing to redress misbehaving applications.

4. Consensus

Safety is ensured by reaching distributed consensus on the state of the tree. The general nature of a Merkle tree as discussed in the previous section enables almost any consensus protocol to support delegated mappings, with varying guarantees on the conditions under which safety is maintained and different trust implications. For example, a deployment on a cluster of nodes running a classic Byzantine Fault Tolerant consensus protocol such as [PBFT] requires a limited, static membership and can tolerate compromises in up to a third of its nodes. In comparison, proof-of-work schemes including many cryptocurrencies have open membership but rely on economic incentives and distributed control of hashing power to provide safety, and federated consensus algorithms like the Stellar Consensus

Protocol (SCP) [I-D.mazieres-dinrg-scp] combine dynamic members with real-world trust relationships but require careful configuration. Determining which scheme, if any, is the "correct" protocol to support authenticated delegation is an open question.

4.1. Interface

Explicit requirement for voting HERE

At a minimum, the consensus layer is expected to provide mechanisms for nodes to

1. Submit new values (commonly cell, but also root listing, updates) for consensus

2. Receive externalized values to which the protocol has committed

3. Validate values received from other nodes for each iteration of the protocol, as specified below

4. Voting mechanism for making root listing governance decisions

Specific protocols may require additional functionality from the delegated mapping layer, which should be implemented to ensure that valid updates are eventually applied (assuming a working consensus layer).

4.2. Validation

Incorrect (potentially malicious) updates to the Merkle tree should be rejected by nodes participating in consensus. Given the known prefix-delegation scheme, each node can apply the same validation procedure without requiring table-specific or application-specific knowledge. Validation also provides a simple mechanism for rate-limiting actors attempting to perform DoS attacks, as only the most recent change to a particular cell need be retained, and the total number of updates to any particular table or overall can be capped. Upon any modification to the delegation tables, a "SetOperation" or "SetRootOperation" as defined in the previous section, the submitted change to the consensus layer should:

1. Reference an existing application identifier in the root key listing and a valid table if applicable.

2. For updates to all cells:

    * contain an unmodified "create_time" or a current timestamp if a new cell

* contain a current "revision_time" in the case of an update

* set a "commitment_time" greater than or equal to the previous commitment

* not grant unlimited allowance (value 0) to delegate cells unless the delegating table also has an unlimited allowance

* result in a total table size ("valuecell" count + "delegatecell" allowances) less than or equal to the table allowance, if not unlimited

* contain a valid signature of the overall cell data by the table authority

3. For updates to value cells:

* be signed only by the current "owner_key" if the cell commitment has not yet expired, or by a new owner upon expiration

* have a lookup key in the table that belongs to the authority's namespace

* not conflict with other cells in its table, breaking the prefix-delegation property

4. For updates to delegate cells:

* be signed by the table authority's public key for new delegations or updates

* retain the same "namespace" and "delegee" value unless the "commitment_time" is expired

* contain a valid namespace owned by the authority delegating the cell

* not conflict with other values or delegations in the same table, breaking the prefix-delegation property

Only after a round of the consensus protocol is successful are the changes exposed to client lookups.

4.3. SCP

While consensus can be reached with many protocols, this section describes how the interface described above can be satisfied by SCP.

Updates to the delegation tables consist of the table change itself (the new version of the cell). Since SCP does not need specific knowledge of the format of these proofs, they directly form consensus on the opaque values submitted to the consensus layer. Once a combination of proofs are agreed to as outputs for a given slot, they are externalized to the mapping layer and applied to the local node's table states. [I-D.mazieres-dinrg-scp] requires this layer to provide a _validity_ function that is applied to each input value, allowing nodes to detect misformed cells that violate the delgation semantics as defined by the previous subsection.

SCP asks the higher-level protocol to define a _combining_ function to compose multiple candidate values. In this application, we can take the union of valid updates proposed by the consensus nodes, rejecting duplicate updates to the same cell in favor of the most up-to-date timestamp.

Finally, SCP by specification uses federated voting to confirm values, which can be used directly to propose and validate modifications to the root key listing.

5. Security Considerations

The security of the delegation tables is primarily tied to the safety properties of the underlying consensus layer. Further, incorrect use of the public key infrastructure authenticating each mapping or compromise of a namespace root key can endanger mappings delegated by the key after their commitments expire.

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>.

[Trillian] Google, "Trillian: General Transparency", n.d., <https://github.com/google/trillian>.

6.2. Informative References

[I-D.mazieres-dinrg-scp]
Barry, N., Losa, G., Mazieres, D., McCaleb, J., and S. Polu, "The Stellar Consensus Protocol (SCP)", draft-mazieres-dinrg-scp-05 (work in progress), November 2018.

[IP-blockchain]
Angieri, S., Garcia-Martinez, A., Liu, B., Yan, Z., Wang, C., and M. Bagnulo, "An experiment in distributed Internet address management using blockchains", 2018, <https://arxiv.org/pdf/1807.10528.pdf>.

[PBFT] Castro, M. and B. Liskov, "Practical Byzantine Fault Tolerance", 1999, <http://pmg.csail.mit.edu/papers/osdi99.pdf>.

[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>.

[RFC7249] Housley, R., "Internet Numbers Registries", RFC 7249, DOI 10.17487/RFC7249, May 2014, <https://www.rfc-editor.org/info/rfc7249>.

Acknowledgments

We are grateful for the contributions and feedback on design and applicability by David Mazieres, as well as help and feedback from many members of the IRTF DIN research group, including Dirk Kutscher and Melinda Shore.

This work was supported by The Stanford Center For Blockchain Research.

Authors' Addresses

Sydney Li
Electronic Frontier Foundation
815 Eddy Street
San Francisco, CA 94109
US

Email: sydney@eff.org

Colin Man
Stanford University
353 Serra Mall
Stanford, CA 94305
US

Email: colinman@cs.stanford.edu

Jean-Luc Watson
UC Berkeley
Cory Hall, 545W
Berkeley, CA 94720
US

Email: jlwatson@eecs.berkeley.edu