Network Working Group
Internet-Draft
Intended status: Experimental
Expires: April 26, 2019

J. Watson
UC Berkeley
S. Li
EFF
C. Man
Stanford University
October 23, 2018

# Delegated Distributed Mappings
# draft-watson-dinrg-delmap-01

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 abilities 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; clients can query the local state of any number of participants and receive the correct result barring a compromise at the consensus layer.

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 April 26, 2019.

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 through intermediary registries by the Internet Numbers Registry [RFC7249]. An enormous amount of value and trust is therefore placed in these assignments (in this draft, _mappings_) yet they are dangerously ephemeral. Delegating authorities, either maliciously or accidentally, can 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 by publishing authenticated time-locked commitments to namespace ownership entries. Known entities identified by public

key are assigned namespaces (e.g. domain prefixes) under which they are authorized to create mapping records, or _cells_. A namespace's cells 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 service clients with bogus entries. We maintain security and consistency through 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, and grant consensus nodes the ability to enforce high-level rules associated with the tables. 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 delegate a portion of its own namespace recursively, but must document the specific range and delegee in on of the table's cells. Since each delegation forms a new table, for which a delegee is the sole authority, assigned namespace ranges must be unique. Consensus can also 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. 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 entries. 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. The cell's "update_sig" must be made by either the "owner_key", or when created, the authority of the table containing the cell, as is described below. 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;

        signature update_sig; /* Table signs cell creation, owner signs updates
*/
    };
```

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 an 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 legitimate mappings they can create. Finally, an _allowance_ must be provided to limit the upper-bound size of a delegated table. Negative allowance values indicates no limit is placed on the 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 to limit the storage burden on consensus nodes. A table with a non-negative allowance may not grant a delegee a negative one. 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-negative. This must be validated during consensus before adding new cells to a table, which can be done at every consensus node because table entry counts are visible publicly.

```
    struct delegatecell {
        opaque namespace<>;
        publickey delegee;
        signature authority_sig;  /* Delegator solely controls inclusion in
table  */
        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 mapping. It is an explicit guarantee on behalf of the table's authority that the mapping will remain valid until at least the specified time. Therefore, while value cell owners may modify their cell at any time (e.g. 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 reconfiguration is customizable to the use case. Larger services should use longer delegation periods for stability whereas small namespaces with a smaller number of users should use shorter delegations.

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

struct cell {
    unsigned hyper create_time;     /* 64-bit UNIX timestamps */
    unsigned hyper *revision_time;
    unsigned hyper commitment_time;
    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<>;
};
```

To generalize correctness, each table must conform with a prefix-based rule: for every cell "c" in a table controlling namespace "x",

"x" must be a prefix of "c" and there cannot exist another cell "c'" such that "c" is a prefix of "c'". While there exist many more hierarchical delegation mechanisms, 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.

### 2.3. 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;
    int allowance;
}

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

A significant question is how to properly administer entries in this listing, which we address in Security Considerations.

## 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 || refers to 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-layer requirements

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 best protocol to support authenticated delegation is an open question.

### 4.1. Interface

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

Most input values to the consensus layer will consist of cell updates, but the same mechanism is ideally suited for updates to the root key listing, as previously discussed. 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 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

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

3. For updates to value cells:

   * be signed with the table authority's public key for new mappings

   * be signed only by the current "owner_key" if the cell commitment has not yet expired, or by either the owner or table authority upon expiration for updates to the value or owner keys

   * 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

   * not grant unlimited (negative) allowance unless the delegating table also has an unlimited allowance

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

## 5. Security Considerations

### 5.1. DoS mitigation

Full consensus nodes must maintain complete, up-to-date table state in order to correctly validate and apply updates. A significant concern is limiting computation and storage resources expended as the result of malicious entities operating in the delegation structure. This is doubly important because of the explicit lack of trust from a delegee to its delegating namespace. While this prevents higher-level organizations from making arbitrary changes to delegated namespaces (as is currently possible in the CA hierarchy), a delegee may choose to incur unreasonable storage costs by filling their table with millions of garbage cells. Of course, since the delegee has a commitment to controlling the specific namespace for a certain time period, these cells cannot be removed. We recognize that this requires the provider to place some amount of trust in their users to consume resources responsibly, and attempt to limit misuse.

The allowances included in each delegation work to address this, since it explicitly defines an agreement between the delegator and delegee as to the expected size required for correct operation. Since allowances are provided at the root level as well (ignoring unlimited allowances) there exists an upper bound on the total number of cells that consensus nodes should expect to be required to maintain. Importantly, the ability to unlimit the table size (as well as further delegations) increases the risk of misuse but provides significant flexibility for well-known systems like DNS and IP allocation. This can be mitigated by assigning unlimited allowances only to well-known entities where real-world accountability limits the urge to misbehave. Consensus nodes are also encouraged to rate-limit excessive "SetOperation"s from clients to further limit this issue.

### 5.2. Consensus node compromise

We rely on the safety properties of the underlying consensus layer to provide a consistent view of the delegated mapping tables. This ensures that no honest node will serve mappings to clients that have not succeeded at reaching consensus. There is nothing directly preventing compromised consensus nodes from maliciously serving entries (e.g. incorrect DNS zone records) to clients as they see fit, _however_, they must also provide an inclusion proof and expose their Merkle root hash. As noted previously, clients and other auditing parties may compare roots and discover misbehavior. The proof associated with a query is unequivocal proof that is sufficient to ignore the compromised node in further consensus rounds. Past individual compromise, the exact point at which a network of consensus nodes can completely violate safety varies from protocol to protocol (majority hashing power attack in Bitcoin, no quorum

intersection of well-behaved nodess in SCP, etc.). Thus, it is not secure to rely only on a small group of nodes hosted by one or two distinct entities for consensus, as they are easily targeted. The generalized delegated mappings mechanism described in this draft allows parties from radically different sectors to collectively provide security, limiting the impact of a small number of malicious nodes. Finally, in the case of an extremely large-scale compromise, mappings stored in prior trees with known root hashes are still valid - they cannot be modified without forging the inclusion proof whose root hash the client will verify.

### 5.3. Upstream compromise

As in any hierarchical delegation system, some amount of trust must be placed in the upstream provider. With this work, we strive to minimize the amount and nature of trust that any entity has to place in their upstream dependencies.

In a regular authenticated delegation system, the network must unilaterally trust a particular namespace operator to not equivocate (i.e. not present different states of the database to different entities) and to not hijack control of a particular delegated entry. Under consensus, nodes can trust that a single entity cannot force the network to equivocate, and entities can audit the database for any misbehavior. The time-bound commitments to namespace delegations limit misuse to the brief renewal window, during which end-entities can monitor the network for misbehavior.

Although an upstream entity can still unilaterally censor and deny service to a particular entity for the namespace that they control, their ability to hijack an existing delegee's entries is both limited and auditable.

### 5.4. Root listing governance

Relying on a centralized party in the long term to reliably and consistently manage the root key listing would create a centralized point of failure, so we consider alternative mechanisms of governing the root of the structure presented in this draft. 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, 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. 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. 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 potential real-world information) determine whether or not to accept the change. Note that if the consensus layer is compromised as discussed above, the governance of the root listing also becomes insecure.

# 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-04 (work in progress), June 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

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

Email: jlwatson@eecs.berkeley.edu

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