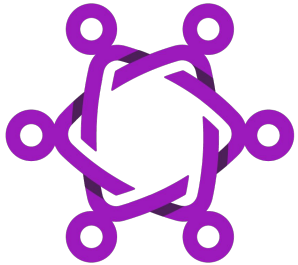


AppSync / GCEDiff

draft-barnes-mls-appsync

draft-mahy-mls-gce-diff

Two simplifying assumptions in RFC 9420

1. We can update GroupContext extensions simply by sending an entirely new set of extensions (in a GroupContextExtensions proposal)

2. We can decide whether a Commit needs an UpdatePath based solely on the proposal types of proposals being committed

Assumptions break down in some use cases

1. We can update GroupContext extensions simply by sending an entirely new set of extensions (in a GroupContextExtensions proposal)

   Resending is expensive if extensions store a lot of data

2. We can decide whether a Commit needs an UpdatePath based solely on the proposal types of proposals being committed

   GCE always has to send an UpdatePath, even if innocuous

Q1: Should we enable diffs on extensions?

```
struct {
  ExtensionType extension_type;
  OperationType operation;
  select (operation) {
    case remove:
      struct {};
    case add:
      opaque extension_data<V>;
    case replace:
      opaque extension_data<V>;
    case diff:
      DiffType diff_type;
      opaque diff_data<V>;
  };
} ExtensionDiff;

struct {
  ExtensionDiff diffs<V>;
} GroupContextExtensionsDiff;
```

New proposal type with a diff format on GroupContext.extensions

Add/Remove/Replace/Diff on individual extensions

Extensible set of diff formats (map, list, opaque, etc.)

Might require application input on whether a proposal is valid

Q2: Should we allow diffs without path?

Value	Name	Recommended	Ext	Path	Reference
0x0000	RESERVED	-	-	-	[RFC9420]
0x0001	add	Y	Y	N	[RFC9420]
0x0002	update	Y	N	Y	[RFC9420]

Following the rules in RFC 9420, GCEdiff would always require an UpdatePath

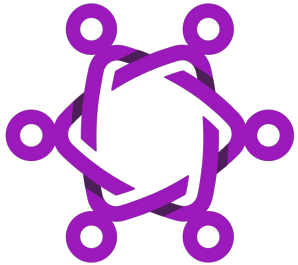
… because it might be updating something unsafe, like GCE

This is silly if extensions hold app state (e.g., AppSync / MIMI)

We could define an alternative rule for whether GCEdiff requires a path…

e.g., by defining “path required” at the extension level

… or define an AppSync proposal with the same mechanics but never requires a path



Light Clients

draft-kiefer-mls-light

Problem: Download and Memory

MLS requires that clients download, validate, and store the group's ratchet tree

Each participant only uses one MLSCiphertext in a Commit

… but has to download O(N) data to verify the Commit signature

In large groups, these objects can be L A R G E

In a 1,000-participant Webex meeting (empty tree):

Tree: 3.5MB in memory / 2.3MB gzip'ed

Commit: 391KB

Light Clients

A **light client** is a member of the group that **does not have the ratchet tree**

A light client **cannot commit**

A light client **cannot process a normal Commit**

Instead:

- Light client joins with only a Welcome
- DS transforms Commit into per-light-client LightCommit

A light client can join and follow the group with O(log N) download / memory

Corollaries

Each group must have at least one full (non-light) client

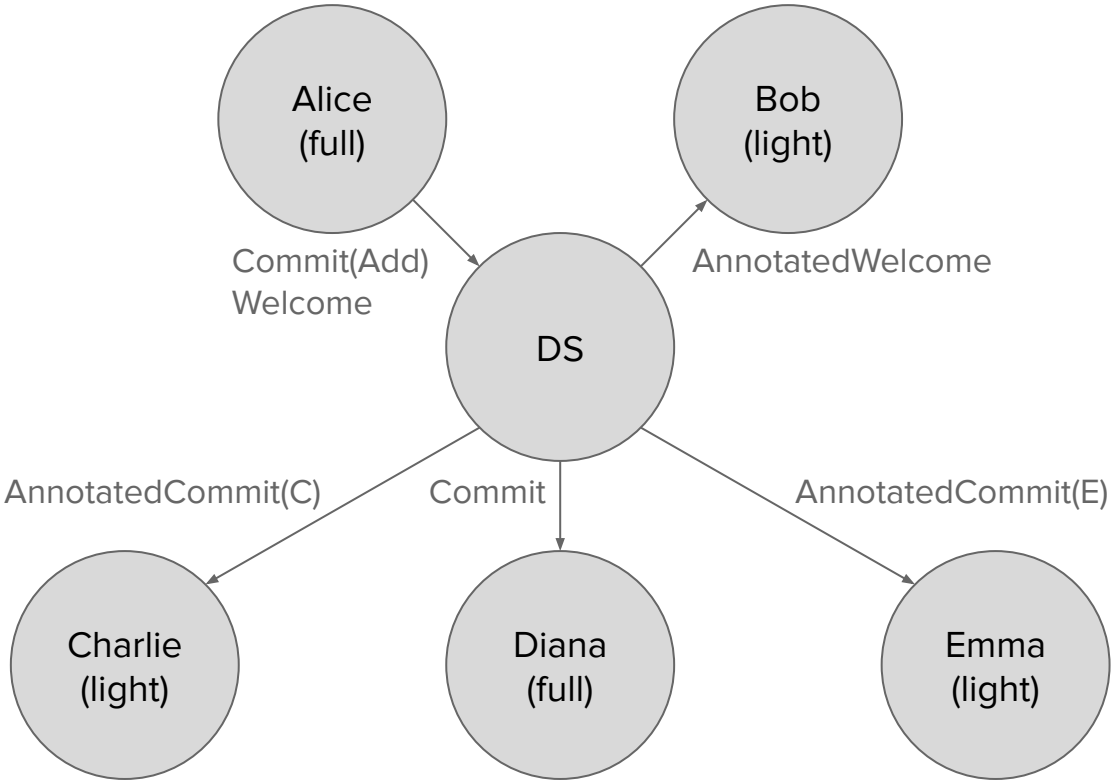
- Someone has to do the commits, otherwise nobody can be added!

Clients can transition between light and full

- Light -> Full: Download and validate the tree
- Full -> Light: Delete local copy of the tree

DS needs to be aware of which clients are light / full

Operating a Group with Light Clients



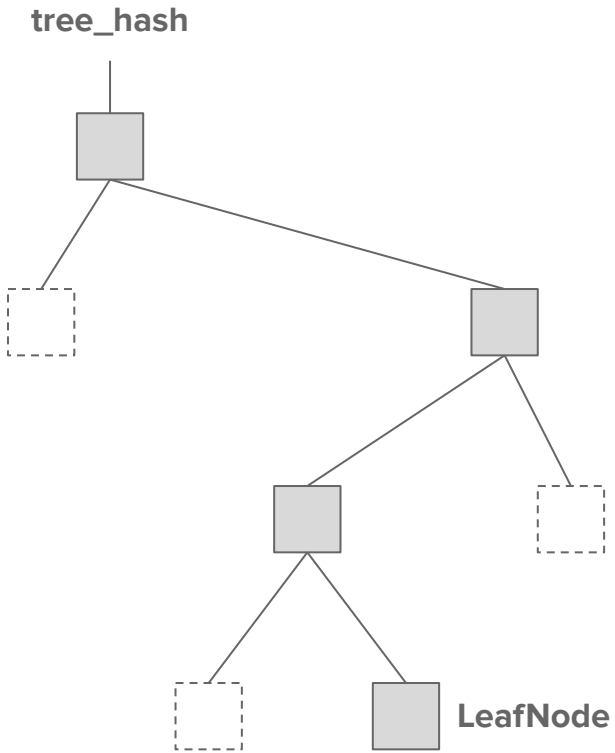
Tree Slices & Membership Proofs

Light clients don’t have the whole tree, but they do have the tree hash

Using Merkle-tree-like constructions, a light client can verify that a specific leaf node is in the tree.

These proofs are O(log N) in size and verification complexity

- Nodes along direct path
- Hashes along copath



AnnotatedWelcome & AnnotatedCommit

```
struct {
    T message;
    MembershipProof sender_membership_proof;
} SenderAuthenticatedMessage<T>;

struct {
    SenderAuthenticated<Welcome> welcome;
    MembershipProof joiner_membership_proof;
} AnnotatedWelcome;

struct {
    MLSMessage commit;
    optional<MembershipProof> sender_membership_proof;
    opaque tree_hash_after<V>;
    optional<uint32> resolution_index;

    MembershipProof sender_membership_proof_after;
    MembershipProof receiver_membership_proof_after;
} AnnotatedCommit;
```

SenderAuthenticatedMessage provides a membership proof so you can verify the signature on a message

Skip tree validation

AnnotatedWelcome provides a proof that the joiner is in the group

AnnotatedCommit provides:

- Coordinates for processing the path
- Proofs of membership after the commit

Light clients still need to see Proposals

Integrates well with Split Commits

Operation	RFC MLS	Light MLS	Split Commits	Light + Split
Join	O(N)	O(log N)	O(N)	O(log N)
Process Commit	O(N)	O(N)	O(log N)	O(log N)

Table 1: Download scaling under protocol variations

Split Commits change Commits from O(N) to O(log N)

If we do AnnotatedSplitCommit, we get everything O(log N) for light clients

Summary

Ratchet Tree and Commits are heavy in large groups

Light clients can join and follow with O(log N) download and memory

... at the cost of not being able to authenticate the whole group

Three main changes:

SenderAuthenticatedMessage - Allow authentication without the whole tree

AnnotatedWelcome - Join without downloading & validating the tree

AnnotatedCommit - Update a limited view of the tree