Each room is hosted at a hub server
All communications go via the hub
  - Messages
  - State changes
  - Ancillary things like KeyPackage fetches

Anticipate that not every server will be willing to talk to every other server
  - A and B might not be willing to talk directly
  - But if users from A and B are in a room hosted at C...
  - A needs to be able to talk to B via C
**Terminology**

The unit of MIMI functionality is a **room**

A room has a **state**, which has a few components:

- An **authorization policy**
- A list of **users** who are **participants** in the room
- An MLS group, including a list of **clients** who are **members** of the group

Participants can either be **active** or **inactive**. Active participants have at least one client that is a member of the MLS group.
Terminology Illustrated

- Authz Policy
  - User 1 Capas
  - User 2 Capas
  - User 3 Capas
  - Admission Policy
  - etc.

- Participant List
  - User 1 (active)
  - User 2 (active)
  - User 3 (inactive)

- MLS Group
  - Client 1a
  - Client 2a
  - Client 2b

- Room State
A user may be a participant only if allowed by the authorization policy.

A client may be a member of the MLS group only if its user is a participant in the room.
Confirmation

Different aspects of the state are managed via separate control mechanisms.

To ensure everyone agrees on state, it is included in the MLS key schedule:
- ... as a GroupContext extension
- If clients don’t agree, they can’t communicate

Each Commit must reflect the current room state.
Order of Operations

Preemption + Confirmation require some coupling between control mechanisms

- User must be on PList before clients added to MLS
- Clients must be removed from MLS before user leaves PList

Sending control messages together will help keep things organized.
Layering

Application

E2E Security

Transport
What is this document?

Not a final protocol!

Explicitly designed as a starting point for the Working Group

Enough mechanism that you can see how basic cases work in detail...

... and envision how the WG would fill in the advanced cases

Some things are notional / stubs!
Layering

Consent, Setup

Policy, Messaging

E2E Security

MLS + DS

HTTPS over mTLS
Overview

- **POST /submit** from Follower to Hub
- **POST /submit** from Follower to Hub
- **POST /notify** from Hub to Follower
- **POST /notify** from Hub to Follower

- Proposal
- Proposal
- Proposal
- Proposal
Basic Scenarios
Alice Creates a Room

[[ No MIMI protocol interaction ]]

Between Alice and her Provider (which becomes the Hub), set up:

- MLS Group
- Room identifier
- Room policy
- Participant list
Alice adds Bob (1)

ClientA1 $\rightarrow$ ServerA: [[ request KPs for Bob ]]
ServerA $\rightarrow$ ServerB: POST /keyMaterial KeyMaterialRequest
ServerB: Verify that Alice is authorized to fetch KeyPackages
ServerB: Mark returned KPs as reserved for Alice’s use
ServerB $\rightarrow$ ServerA: 200 OK KeyMaterialResponse
ServerA $\rightarrow$ ClientB*: [[ KPs ]]
**Alice adds Bob (2)**

ClientA1: Prepare Commit over AppSync(+Bob), Add*
ServerA: Verify that AppSync, Adds are allowed by policy
ServerA: Identifies Welcome domains based on KP hash in Welcome
ServerA->ServerB: POST /notify/a.example/r/clubhouse Intro{ Welcome, RatchetTree? }
ServerB: Recognizes that Welcome is adding Bob to room clubhouse
ServerB->ClientB*: [[ Welcome, RatchetTree? ]]
Bob adds Cathy (1)
Bob adds Cathy (2)
Bob leaves

ClientB\(1\): Prepare Remove*, AppSync(-Bob)
ClientB\(1\)->ServerB: [[ Remove*, AppSync ]]
ServerB->ServerA: POST /update/a.example/r/clubhouse Remove*, AppSync
ServerA: Verify that Removes, AppSync are allowed by policy; cache
ServerA->ServerB: 200 OK
ServerA->ServerC: POST /notify/a.example/r/clubhouse Proposals
ServerC->ClientC\(1\): [[ Proposals ]]
ClientC\(1\)->ServerC: [[ Commit(Props), Welcome, GroupInfo?, RatchetTree? ]] ServerC->ServerA: POST /update/a.example/r/clubhouse CommitBundle
ServerA: Check whether Commit includes queued proposals; accept
ServerA->ServerC: 200 OK
ServerA->ServerB: POST /notify/a.example/r/clubhouse Commit
ServerB->ServerC: POST /notify/a.example/r/clubhouse Commit
ServerB->ClientB\(1\): [[ Commit ]]
ServerC->ClientC\(2\): [[ Commit ]]
ServerC->ClientC\(1\): [[ Commit ] (up to provider)
Cathy adds a new device
Protocol Details
## Identifiers

<table>
<thead>
<tr>
<th>Identifier type</th>
<th>Example URI</th>
</tr>
</thead>
<tbody>
<tr>
<td>Provider</td>
<td>mimi://a.example</td>
</tr>
<tr>
<td>User</td>
<td>mimi://a.example/u/alice</td>
</tr>
<tr>
<td>Client</td>
<td>mimi://a.example/d/ClientA1</td>
</tr>
<tr>
<td>Room</td>
<td>mimi://a.example/r/clubhouse</td>
</tr>
<tr>
<td>MLS group</td>
<td>mimi://a.example/g/clubhouse</td>
</tr>
</tbody>
</table>

*Table 1: MIMI URI examples*
Server-to-Server Communications

POST /notify HTTP/1.1
Host: hub.example
From: mimi@follower.example
Content-Type: application/mimi

HTTPS over mutually-authenticated TLS

RFC 6125 framework of presented vs. reference identifiers

TLS certificates contain a set of presented identifiers for each endpoint

Server reference identity in Host / :authority

Client reference identity in From
MLS and AppSync

MIMI uses MLS for three things:

1. Protecting application messages
2. Managing which devices have access to the group
3. Ensuring agreement on room state

(1) and (2) are features of base MLS

For (3), we define an AppSync extension – basically a key/value store of application data that the devices in a room all agree on

Two types of sync’ed state: Policy + Participant List
Policy and Participants

Policy Framework:

- Granular capabilities
- Hub-defined roles = sets of capabilities

Participant list associates user identifier to role

Here, we assume that Alice uses ClientA1 to create a room with the following base policy properties:

- Room Identifier: mimi://a.example/r/clubhouse
- Roles: admin = [canAddUser, canRemoveUser, canSetUserRole]

And the following participant list:

- Participants: [[mimi://a.example/u/alice, "admin"]]

26
## HTTP endpoints

<table>
<thead>
<tr>
<th>Direction</th>
<th>Endpoint</th>
<th>Purpose</th>
</tr>
</thead>
<tbody>
<tr>
<td>*</td>
<td>GET /.well-known/mimi-protocol-directory</td>
<td>Learn the endpoints for a server</td>
</tr>
<tr>
<td>F ➡️ H ➡️ F</td>
<td>POST /keyMaterial/{targetUser}</td>
<td>Get initial key material for a user</td>
</tr>
<tr>
<td>F ➡️ H</td>
<td>POST /update/{roomId}</td>
<td>Send Proposals / Commit updating the room</td>
</tr>
<tr>
<td>F ➡️ H</td>
<td>POST /submitMessage/{roomId}</td>
<td>Submit a message for delivery</td>
</tr>
<tr>
<td>H ➡️ F</td>
<td>POST /notify/{roomId}</td>
<td>Receive events / messages</td>
</tr>
<tr>
<td>F ➡️ H</td>
<td>POST /groupInfo/{roomId}</td>
<td>Fetch the GroupInfo for the room</td>
</tr>
</tbody>
</table>
EKR’s Comments about Layering
Where should app state changes be?

Generally agreed that:

- State changes need to be sender-authenticated
- Application state should be confirmed via the MLS key schedule

Should this be realized via:

- Incorporating state changes in the MLS control layer? (current)
- Doing state changes at the MLS app layer, and injecting into the key schedule?
Design Alternatives

draft-02

alternative
Let's punt on this

This is a specific design question we can debate in the WG

File an issue, issue an adoption call
Known Open Issues

Authorization policy
Advanced join/leave flows
Consent
Identifiers
Abuse reporting
Identifier resolution (within Provider scope)
End-to-end Identity
Questions for the Group

Do we agree that the basic scenarios here work?

Does the document articulate a framework that can be elaborated to a full solution?

Ready for adoption?
Backup
Consent

PR #72 has a draft consent mechanism

1. Alice can request consent from Bob to:
   a. Talk to Bob in $ROOM
   b. Talk to Bob in general
2. Alice can cancel a request for consent
3. Alice can proactively grant consent for Bob to contact her
4. Bob can grant Alice’s request
   a. Exactly as Alice asked
   b. More generously – “Contact me any way, not just this room”
   c. More narrowly – “Don’t contact me except through this room”
5. Bob can revoke / deny consent