More Instant Messaging Interoperability
Internet-Draft
Intended status: Standards Track
Expires: 8 January 2026

R. L. Barnes
Cisco
M. Hodgson
The Matrix.org Foundation C.I.C.
K. Kohbrok
Phoenix R&D
R. Mahy
Rohan Mahy Consulting Services
T. Ralston
The Matrix.org Foundation C.I.C.
R. Robert
Phoenix R&D
7 July 2025

More Instant Messaging Interoperability (MIMI) using HTTPS and MLS
draft-ietf-mimi-protocol-04

Abstract

This document specifies the More Instant Messaging Interoperability (MIMI) transport protocol, which allows users of different messaging providers to interoperate in group chats (rooms), including to send and receive messages, share room policy, and add participants to and remove participants from rooms. MIMI describes messages between providers, leaving most aspects of the provider-internal client-server communication up to the provider. MIMI integrates the Messaging Layer Security (MLS) protocol to provide end-to-end security assurances, including authentication of protocol participants, confidentiality of messages exchanged within a room, and agreement on the state of the room.

About This Document

This note is to be removed before publishing as an RFC.

The latest revision of this draft can be found at https://bifurcation.github.io/ietf-mimi-protocol/draft-ralston-mimi-protocol.html. Status information for this document may be found at https://datatracker.ietf.org/doc/draft-ietf-mimi-protocol/.

Discussion of this document takes place on the More Instant Messaging Interoperability Working Group mailing list (mailto:mimi@ietf.org), which is archived at https://mailarchive.ietf.org/arch/browse/mimi/. Subscribe at https://www.ietf.org/mailman/listinfo/mimi/.

Source for this draft and an issue tracker can be found at https://github.com/bifurcation/ietf-mimi-protocol.

Status of This Memo

Copyright Notice

Table of Contents

1. Introduction

The More Instant Messaging Interoperability (MIMI) transport protocol enables providers of end-to-end encrypted instant messaging to interoperate. As described in the MIMI architecture [I-D.barnes-mimi-arch], group chats and direct messages are described in terms of "rooms". Each MIMI protocol room is hosted at a single provider (the "hub" provider"), but allows users from different providers to become participants in the room. The hub provider is

responsible for ordering and distributing messages, enforcing policy, and authorizing messages. It also keeps a copy of the room state, which includes the room policy and participant list, which it can provide to new joiners. Each provider also stores initial keying material for its own users (who may be offline).

This document describes the communication among different providers necessary to support messaging application functionality, for example:

* Sharing room policy

* Adding and removing participants in a room

* Exchanging secure messages

In support of these functions, the protocol also has primitives to fetch initial keying material and fetch the current state of the underlying end-to-end encryption protocol for the room.

Messages sent inside each room are end-to-end encrypted using the Messaging Layer Security (MLS) protocol [RFC9420], and each room is associated with an MLS group. MLS also ensures that clients in a room agree on the room policy and participation. MLS is integrated into MIMI in such a way as to ensure that a client is joined to a room's MLS group only if the client's user is a participant in the room, and that all clients in the group agree on the state of the room (including, for example, the room's participant list).

1.1. Known Gaps

In this version of the document, we have tried to capture enough concrete functionality to enable basic application functionality, while defining enough of a protocol framework to indicate how to add other necessary functionality. The following functions are likely to be needed by the complete protocol, but are not covered here:

Authorization policy: In this document, we introduce a notional concept of roles for participants, and permissions for roles. Concrete authorization policies are defined in [I-D.ietf-mimi-room-policy].

Knock and invite flows: This document describes how user can be added, or how authorized users can add themselves to a group based on the policy of the room. It does not include flows where a user can "knock" to ask to enter a room, nor does it include "invitations", where a user offers information to another user about how to be added to a room.

Identifiers: Certain entities in the MIMI system need to be identified in the protocol. In this document, we define a notional syntax for identifiers, but a more concrete one should be defined.

Authentication While MLS provides basic message authentication, users should also be able to (cryptographically) tie the identity of other users to their respective providers. Further authentication such as tying clients to their users (or the user's other clients) may also be desirable.

2. Conventions and Definitions

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.

Terms and definitions are inherited from [I-D.barnes-mimi-arch]. We also make use of terms from the MLS protocol [RFC9420].

Throughout this document, the examples use the TLS Presentation Language [RFC8446] and the semantics of HTTP [RFC7231] respectively as placeholder a set of binary encoding mechanism and transport semantics.

The protocol layering of the MIMI transport protocol is as follows:

1. An application layer that enables messaging functionality
2. A security layer that provides end-to-end security guarantees:
    * Confidentiality for messages
    * Authentication of actors making changes to rooms
    * Agreement on room state across the clients involved in a room
3. A transport layer that provides secure delivery of protocol objects between servers.

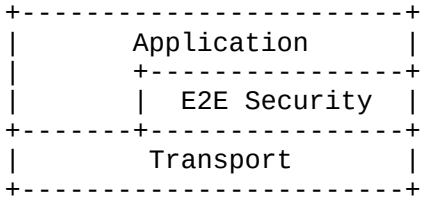


Figure 1: MIMI protocol layering

MIMI uses MLS for end-to-end security, using the MLS AppSync proposal type to efficiently synchronize room state across the clients involved in a room [RFC9420] [I-D.barnes-mls-appsync]. The MIMI transport is based on HTTPS over mutually-authenticated TLS.

3. Example protocol flow

This section walks through a basic scenario that illustrates how a room works in the MIMI protocol. The scenario involves the following actors:

* Service providers a.example, b.example, and c.example represented by servers ServerA, ServerB, and ServerC respectively

* Users Alice (alice), Bob (bob) and Cathy (cathy) of the service providers a.example, b.example, and c.example respectively.

* Clients ClientA1, ClientA2, ClientB1, etc. belonging to these users

* A room clubhouse hosted by hub provider a.example where the three users interact.

Inside the protocol, each provider is represented by a domain name in the host production of the authority of a MIMI URI [RFC3986]. Specific hosts or servers are represented by domain names, but not by MIMI URIs. Examples of different types of identifiers represented in a MIMI URI are shown in the table below:

Identifier type	Example URI
Provider	mimi://a.example
User	mimi://a.example/u/alice
Client	mimi://a.example/d/ClientA1
Room	mimi://a.example/r/clubhouse
MLS group	mimi://a.example/g/clubhouse

Table 1: MIMI URI examples

As noted in [I-D.barnes-mimi-arch], the MIMI protocol only defines interactions between service providers' servers. Interactions between clients and servers within a service provider domain are shown here for completeness, but surrounded by [[double brackets]].

3.1. Alice Creates a Room

The first step in the lifetime of a MIMI room is its creation on the hub server. This operation is local to the service provider, and does not entail any MIMI protocol operations. However, it must establish the initial state of the room, which is then the basis for protocol operations related to the room.

For authorization purposes, MIMI uses permissions based on room-defined roles. For example, a room might have a role named "admin", which has canAddUser, canRemoveUser, and canSetUserRole permisions.

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"]]

ClientA1 also creates an MLS group with group ID mimi://a.example/g/clubhouse and ensures via provider-local operations that Alice's other clients are members of this MLS group.

3.2. Alice adds Bob to the Room

Adding Bob to the room entails operations at two levels. First, Bob's user identity must be added to the room's participant list. Second, Bob's clients must be added to the room's MLS group.

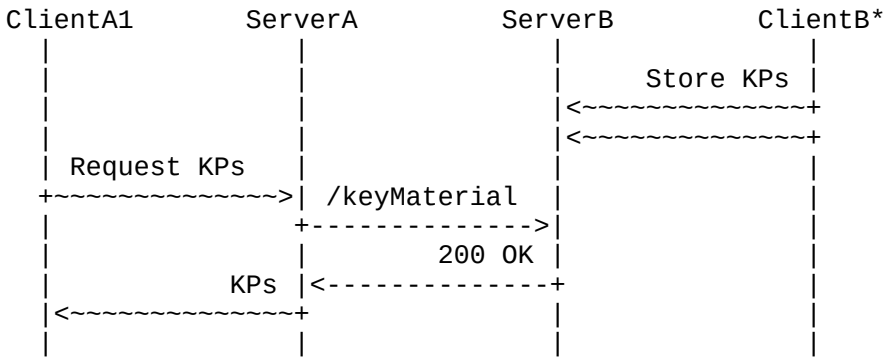
The process of adding Bob to the room thus begins by Alice fetching key material for Bob's clients. Alice then updates the room by sending an MLS Commit over the following proposals:

* An AppSync proposal updating the room state by adding Bob to the participant list

* Add proposals for Bob's clients

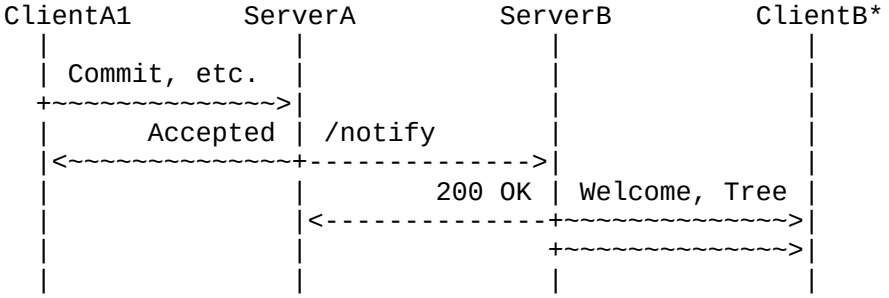
The MIMI protocol interactions are between Alice's server ServerA and Bob's server ServerB. ServerB stores KeyPackages on behalf of Bob's devices. ServerA performs the key material fetch on Alice's behalf, and delivers the resulting KeyPackages to Alice's clients. Both ServerA and ServerB remember the sources of the KeyPackages they handle, so that they can route a Welcome message for those KeyPackages to the proper recipients -- ServerA to ServerB, and ServerB to Bob's clients.

> *NOTE:* In the protocol, it is necessary to have consent (see Section 8) and access control on these operations. We have elided that step here in the interest of simplicity.



ClientB*->ServerB: [[Store KeyPackages]]
ClientA1->ServerA: [[request KPs for Bob]]
ServerA->ServerB: POST /keyMaterial KeyMaterialRequest
ServerB: Verify that Alice is authorized to fetch KeyPackages
ServerB: Mark returned KPs as reserved for Alice's use
ServerB->ServerA: 200 OK KeyMaterialResponse
ServerA: Remember that these KPs go to b.example
ServerA->ClientA1: [[KPs]]

Figure 2: Alice Fetches KeyPackages for Bob's Clients

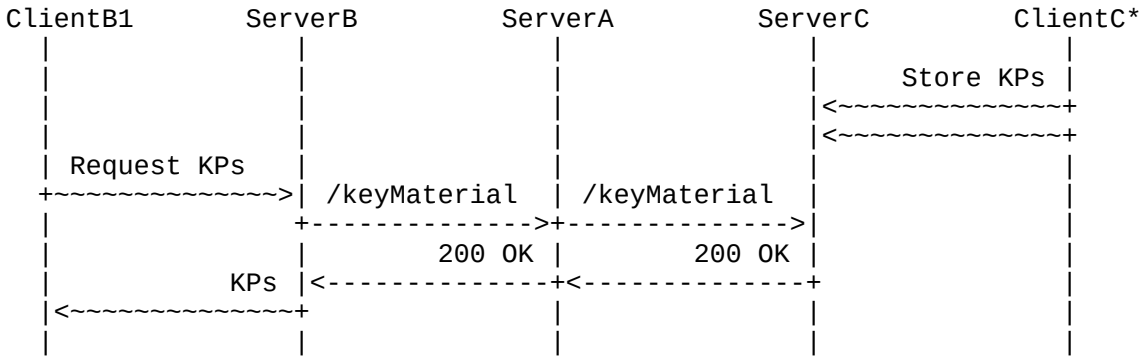


ClientA1: Prepare Commit over AppSync(+Bob), Add*
ClientA1->ServerA: [[Commit, Welcome, GroupInfo?, RatchetTree?]]
ServerA: Verify that AppSync, Adds are allowed by policy
ServerA: Identifies Welcome domains based on KP hash in Welcome
ServerA->ServerB: POST /notify/a.example/r/clubhouse Intro{ Welcome,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

Figure 3: Alice Adds Bob to the Room and Bob's Clients to the MLS Group

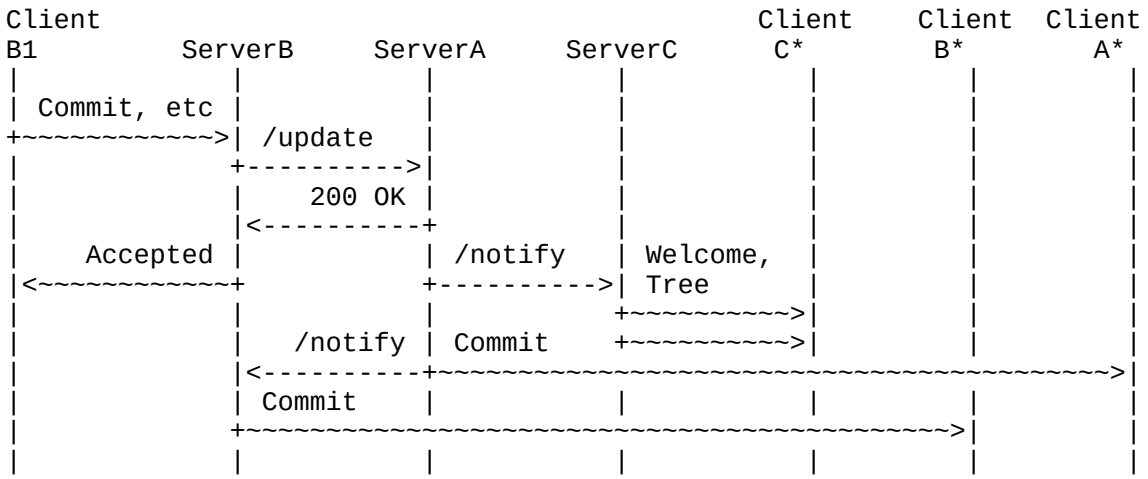
3.3. Bob adds Cathy to the Room

The process of adding Bob was a bit abbreviated because Alice is a user of the hub service provider. When Bob adds Cathy, we see the full process, involving the same two steps (KeyPackage fetch followed by Add), but this time indirected via the hub server ServerA. Also, now that there are users on ServerB involved in the room, the hub ServerA will have to distribute the Commit adding Cathy and Cathy's clients to ServerB as well as forwarding the Welcome to ServerC.




```
ClientC*->ServerC: [[ Store KeyPackages ]]
ClientB1->ServerB: [[ request KPs for Bob ]]
ServerB->ServerA: POST /keyMaterial KeyMaterialRequest
ServerA->ServerC: POST /keyMaterial KeyMaterialRequest
ServerB: Verify that Bob is authorized to fetch KeyPackages
ServerB: Mark returned KPs as reserved for Bob's use
ServerC->ServerA: 200 OK KeyMaterialResponse
ServerA: Remember that these KPs go to b.example
ServerA->ServerB: 200 OK KeyMaterialResponse
ServerB->ClientB1: [[ KPs ]]
```

Figure 4: Bob Fetches KeyPackages for Cathy's Clients

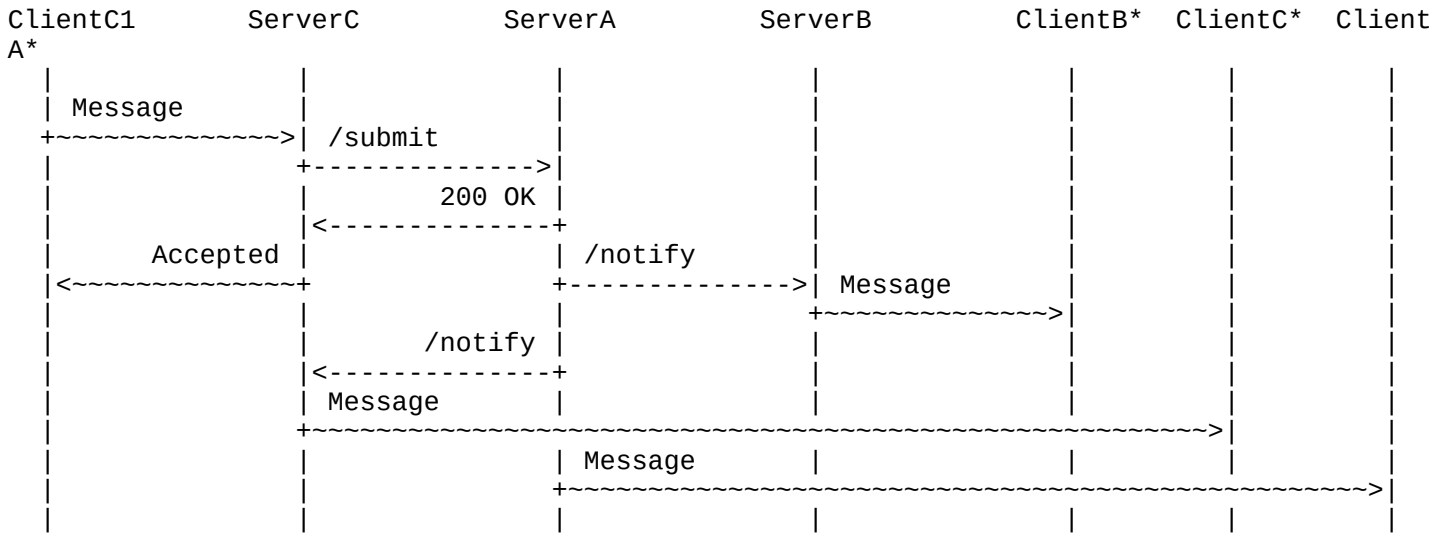


ClientB1: Prepare Commit over AppSync(+Cathy), Add*
ClientB1->ServerB: [[Commit, Welcome, GroupInfo?, RatchetTree?]]
ServerB->ServerA: POST /update/a.example/r/clubhouse CommitBundle
ServerA: Verify that Adds are allowed by policy
ServerA->ServerB: 200 OK
ServerA->ServerC: POST /notify/a.example/r/clubhouse
Intro{ Welcome, RatchetTree? }
ServerC: Recognizes that Welcome is adding Cathy to clubhouse
ServerC->ClientC*: [[Welcome, RatchetTree?]]
ServerA->ServerB: POST /notify/a.example/r/clubhouse Commit
ServerB->ClientB*: [[Commit]]
ServerA->ClientA*: [[Commit]]

Figure 5: Bob Adds Cathy to the Room and Cathy's Clients to the MLS Group

3.4. Cathy Sends a Message

Now that Alice, Bob, and Cathy are all in the room, Cathy wants to say hello to everyone. Cathy's client encapsulates the message in an MLS PrivateMessage and sends it to ServerC, who forwards it to the hub ServerA on Cathy's behalf. Assuming Cathy is allowed to speak in the room, ServerA will forward Cathy's message to the other servers involved in the room, who distribute it to their clients.



ClientC1->ServerC: [[MLSMessage(PrivateMessage)]]
ServerC->ServerA: POST /submit/a.example/r/clubhouse MLSMessage(PrivateMessage)
ServerA: Verifies that message is allowed
ServerA->ServerC: POST /notify/a.example/r/clubhouse Message{ MLSMessage(PrivateMessage) }
ServerA->ServerB: POST /notify/a.example/r/clubhouse Message{ MLSMessage(PrivateMessage) }
ServerA->ClientA*: [[MLSMessage(PrivateMessage)]]
ServerB->ClientB*: [[MLSMessage(PrivateMessage)]]
ServerC->ClientC*: [[MLSMessage(PrivateMessage)]]

Figure 6: Cathy Sends a Message to the Room

3.5. Bob Leaves the Room

A user removing another user follows the same flow as adding the user. The user performing the removal creates an MLS commit covering Remove proposals for all of the removed user's devices, and an AppSync proposal updating the room state to remove the removed user from the room's participant list.

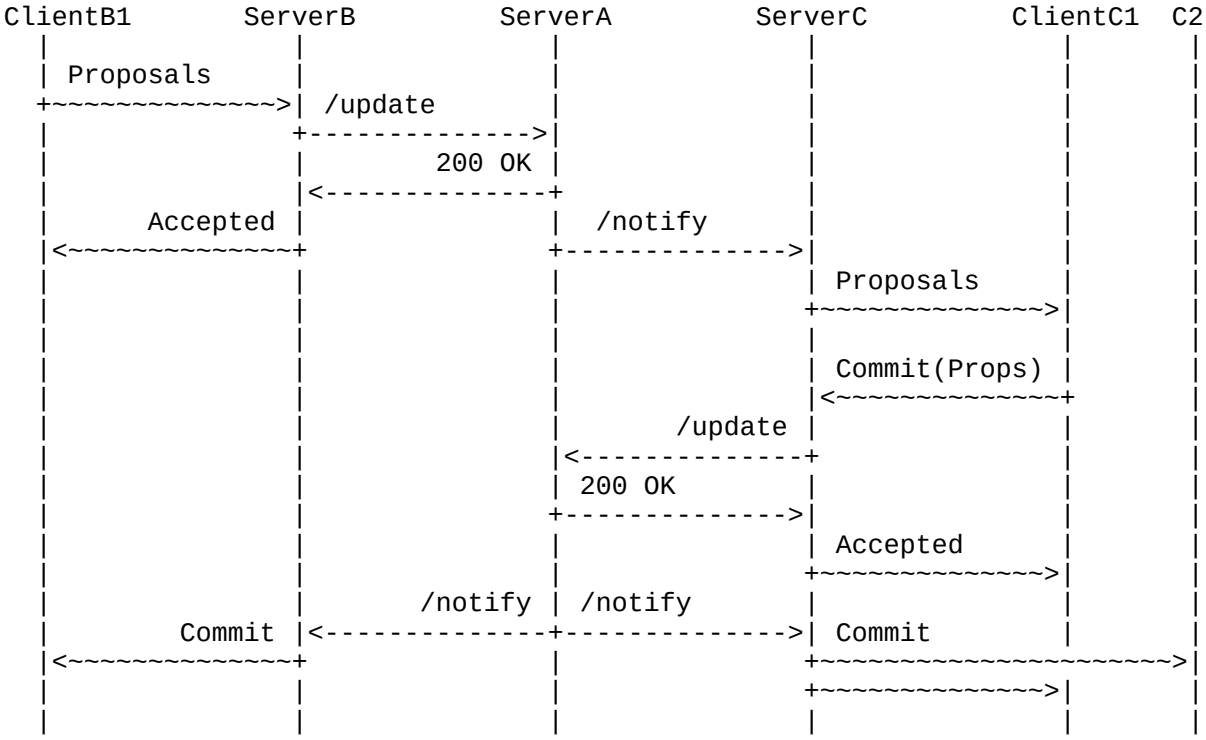
One's own user leaving is slightly more complicated than removing another user, because the leaving user cannot remove all of their devices from the MLS group. Instead, the leave happens in three steps:

1. The leaving client constructs MLS Remove proposals for all of the user's devices (including the leaving client), and an AppSync proposal that removes its user from the participant list.

2. The leaving client sends these proposals to the hub. The hub caches the proposals.

3. The next time a client attempts to commit, the hub requires the client to include the cached proposals.

The hub thus guarantees the leaving client that they will be removed as soon as possible.



ClientB1: Prepare Remove*, AppSync(-Bob)
ClientB1->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
ServerC1->ClientC1: [[Proposals]]
ClientC1->ServerC: [[Commit(Props), Welcome, GroupInfo?, RatchetTree?]]
ServerC->ServerA: POST /update/a.example/r/clubhousee CommitBundle
ServerA: Check whether Commit includes queued proposals; accept
ServerA->ServerC: 200 OK
ServerA->ServerB: POST /notify/a.example/r/clubhouse Commit
ServerA->ServerC: POST /notify/a.example/r/clubhouse Commit
ServerB->ClientB1: [[Commit]]
ServerC->ClientC2: [[Commit]]
ServerC->ClientC1: [[Commit]] (up to provider)

Figure 7: Bob Leaves the Room

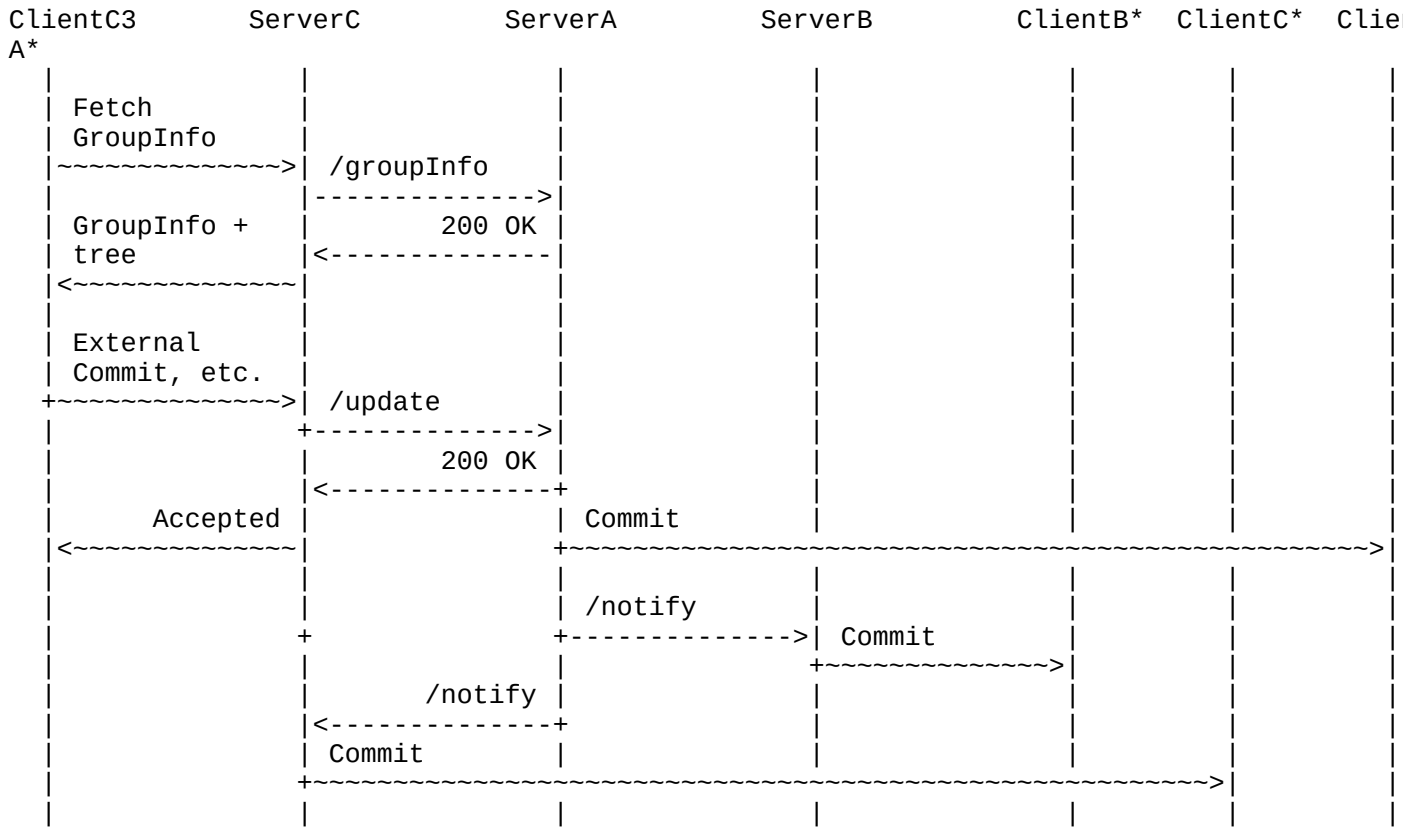
3.6. Cathy adds a new device

Many users have multiple clients often running on different devices (for example a phone, a tablet, and a computer). When a user creates a new client, that client needs to be able to join all the MLS groups associated with the rooms in which the user is a participant.

In MLS in order to initiate joining a group the joining client needs to get the current GroupInfo and ratchet_tree, and then send an External Commit to the hub. In MIMI, the hub keeps or reconstructs a copy of the GroupInfo, assuming that other clients may not be available to assist the client with joining.

For Cathy’s new client (ClientC3) to join the MLS group and therefore fully participate in the room with Alice, ClientC3 needs to fetch the MLS GroupInfo, and then generate an External Commit adding ClientC3.

Cathy’s new client sends the External Commit to the room’s MLS group by sending an /update to the room.



ClientC3->ServerC: [[request current GroupInfo]]
ServerC->ServerA: POST /groupInfo/a.example/clubhouse
ServerA: Verify that ClientC3 has authorization to join the room
ServerA->ServerC: 200 OK w/ current GroupInfo and ratchet tree
ServerC->ClientC3: [[GroupInfo, tree]]
ClientC3: Prepare External Commit Add*
ClientC3->ServerC: [[Commit, GroupInfo?, RatchetTree?]]
ServerC->ServerA: POST /update/a.example/clubhouse CommitBundle
ServerA: Verify that Commit is valid and ClientC3 is authorized
ServerA->ServerC: 200 OK
ServerC->ClientC3: [[External Commit was accepted]]
ServerA->ClientA*: [[Commit]]
ServerA->ServerB: POST /notify/a.example/clubhouse Commit
ServerB->ClientB*: [[Commit]]
ServerA->ServerC: POST /notify/a.example/clubhouse Commit
ServerC->ClientC*: [[Commit]]

Figure 8: Cathy Adds a new Client

4. Services required at each layer

4.1. Transport layer

MIMI servers communicate using HTTPS. The HTTP request MUST identify the source and target providers for the request, in the following way:

* The target provider is indicated using a Host header [RFC9110]. If the provider is using a non-standard port, then the port component of the Host header is ignored.

* The source provider is indicated using a From header [RFC9110]. The mailbox production in the From header MUST use the addr-spec variant, and the local-part of the address MUST contain the fixed string mimi. Thus, the content of the From header will be mimi@a.example, where a.example is the domain name of the source provider.

  NOTE: The use of the From header field here is not really well-aligned with its intended use. The WG should consider whether this is correct, or whether a new header field would be better. Perhaps something like "From-Host" to match Host?

The TLS connection underlying the HTTPS connection MUST be mutually authenticated. The certificates presented in the TLS handshake MUST authenticate the source and target provider domains, according to [RFC6125].

The bodies of HTTP requests and responses are defined by the individual endpoints defined in Section 4.3.

4.2. End-to-End Security Layer

Every MIMI room has an MLS group associated to it, which provides end-to-end security guarantees. The clients participating in the room manage the MLS-level membership by sending Commit messages covering Add and Remove proposals.

Every application message sent within a room is authenticated and confidentiality-protected by virtue of being encapsulated in an MLS PrivateMessage object.

MIMI uses the MLS application state synchronization mechanism ([I-D.barnes-mls-appsync]) to ensure that the clients involved in a MIMI room agree on the state of the room. Each MIMI message that changes the state of the room is encapsulated in an AppSync proposal and transmitted inside an MLS PublicMessage object.

The PublicMessage encapsulation provides sender authentication, including the ability for actors outside the group (e.g., servers involved in the room) to originate AppSync proposals. Encoding room state changes in MLS proposals ensures that a client will not process a commit that confirms a state change before processing the state change itself.

TODO: A little more needs to be said here about how MLS is used. For example: What types of credential are required / allowed? If servers are going to be allowed to introduce room changes, how are their keys provisioned as external signers? Need to maintain the membership and the list of queued proposals.

4.3. Application Layer

Servers in MIMI provide a few functions that enable messaging applications. All servers act as publication points for key material used to add their users to rooms. The hub server for a room tracks the state of the room, and controls how the room's state evolves, e.g., by ensuring that changes are compliant with the room's policy. Non-hub servers facilitate interactions between their clients and the hub server.

In this section, we describe the state that servers keep. The following top level section describes the HTTP endpoints exposed to enable these functions.

4.3.1. Server State

Every MIMI server is a publication point for users' key material, via the keyMaterial endpoint discussed in Section 5.2. To support this endpoint, the server stores a set of KeyPackages, where each KeyPackage belongs to a specific user and device.

Each KeyPackage includes a list of its MLS client's capabilities (MLS protocol versions, cipher suites, extensions, proposal types, and credential types). When claiming KeyPackages, the requester includes the list of RequiredCapabilites to ensure the new joiner is compatible with and capable of participating in the corresponding room.

The hub server for the room stores the state of the room, comprising:

* The _base policy_ of the room, which does not depend on the specific participants in the room. For example, this includes the room _roles_ and their permissions (defined in [I-D.ietf-mimi-room-policy] and _preauthorization_ policy).

* The _participant list_: a list of the users who are participants of the room, and each user's role in the room (defined in Section 7.5).

* Room metadata, such as the room name, description, and image (defined in Section 7.6).

When a client requests key material via the hub, the hub records the KeyPackageRef values for the returned KeyPackages, and the identity of the provider from which they were received. This information is then used to route Welcome message to the proper provider.

4.3.2. Participant List Changes

The participant list can be changed by adding or removing users, or changing a user's role. These changes are described as a list of adds, removes, and role changes, as described in Section 7.5.

```
Add: ["mimi://d.example/u/diana", role: 4 (admin)],
     ["mimi://e.example/u/eric", role: 3 (moderator)],
Remove: ["mimi://b.example/u/bob"],
SetRole: [["mimi://c.example/u/cathy", role: 1 (banned)]]
```

Figure 9: Changing the state of the room

To put these changes into effect, a client or server encodes them in an AppDataUpdate [I-D.ietf-mls-extensions] proposal, signs the proposal as a PublicMessage, and submits them to the update endpoint on the hub.

5. MIMI Endpoints and Framing

This section describes the specific endpoints necessary to provide the functionality in the example flow. The framing for each endpoint includes a protocol so that different variations of the end-to-end encryption can be used.

TODO: Determine the what needs to be included in the protocol. MIMI version, e2e protocol version, etc.

The syntax of the MIMI protocol messages are described using the TLS presentation language format (Section 3 of [RFC8446]).

```
enum {
    reserved(0),
    mls10(1),
    (255)
} Protocol;
```

5.1. Directory

Like the ACME protocol (See Section 7.1.1 of [RFC8555]), the MIMI protocol uses a directory document to convey the HTTPS URLs used to reach certain endpoints (as opposed to hard coding the endpoints).

The directory URL is discovered using the mimi-protocol-directory well-known URI. The response is a JSON document with URIs for each type of endpoint.

```
GET /.well-known/mimi-protocol-directory

{
  "keyMaterial":
     "https://mimi.example.com/v1/keyMaterial/{targetUser}",
  "update": "https://mimi.example.com/v1/update{roomId}",
  "notify": "https://mimi.example.com/v1/notify/{roomId}",
  "submitMessage":
     "https://mimi.example.com/v1/submitMessage/{roomId}",
  "groupInfo":
     "https://mimi.example.com/v1/groupInfo/{roomId}",
  "requestConsent":
     "https://mimi.example.com/v1/requestConsent/{targetUser}",
  "updateConsent":
     "https://mimi.example.com/v1/updateConsent/{requesterUser}",
  "identifierQuery":
     "https://mimi.example.com/v1/identifierQuery/{domain}",
  "reportAbuse":
     "https://mimi.example.com/v1/reportAbuse/{roomId}",
  "proxyDownload"
     "https://mimi.example.com/v1/proxyDownload/{downloadUrl}"
}
```

5.2. Fetch Key Material

This action attempts to claim initial keying material for all the clients of a single user at a specific provider. The keying material is designed for use in a single room and may not be reused. It uses the HTTP POST method.

```
POST /keyMaterial/{targetUser}
```

The target user’s URI is listed in the request path. KeyPackages requested using this primitive MUST be sent via the hub provider of whatever room they will be used in. (If this is not the case, the hub provider will be unable to forward a Welcome message to the target provider).

The path includes the target user URI (with any necessary percent-encoding). The request body includes the protocol (currently just MLS 1.0), and the requesting user. When the request is being made in the context of adding the target user to a room, the request MUST include the room ID for which the KeyPackage is intended, as the target may have only granted consent for a specific room.

For MLS, the request includes a non-empty list of acceptable MLS ciphersuites. Next there is an MLS RequiredCapabilities object, which contains (possibly empty) lists of required credential types, non-default proposal types, and extensions) required by the requesting provider. Next there is a SignaturePublicKey and a corresponding Credential for the requester. Finally, the request includes a signature, using the SignaturePublicKey and covering KeyMaterialRequestTBS.

The request body has the following form.

```
struct {
    opaque uri<V>;
} IdentifierUri;

struct {
    Protocol protocol;
    IdentifierUri requestingUser;
    IdentifierUri targetUser;
    IdentifierUri roomId;
    select (protocol) {
        case mls10:
            CipherSuite acceptableCiphersuites<V>;
            RequiredCapabilities requiredCapabilities;
            SignaturePublicKey requesterSignatureKey;
            Credential requesterCredential;
            /* SignWithLabel(requesterSignatureKey,              */
            /*   "KeyMaterialRequestTBS", KeyMaterialRequestTBS) */
            opaque key_material_request_signature<V>;
    };
} KeyMaterialRequest;

struct {
    Protocol protocol;
    IdentifierUri requestingUser;
    IdentifierUri targetUser;
    IdentifierUri roomId;
    select (protocol) {
        case mls10:
            CipherSuite acceptableCiphersuites<V>;
            RequiredCapabilities requiredCapabilities;
            SignaturePublicKey requesterSignatureKey;
            Credential requesterCredential;
    };
} KeyMaterialRequestTBS;

key_material_request_signature = SignWithLabel(requesterSignatureKey
                      "KeyMaterialRequestTBS", KeyMaterialRequestTBS)
```

The response contains a user status code that indicates keying material was returned for all the user's clients (success), that keying material was returned for some of their clients (partialSuccess), or a specific user code indicating failure. If the user code is success or partialSuccess, each client is enumerated in the response. Then for each client with a _client_ success code, the structure includes initial keying material (a KeyPackage for MLS 1.0). If the client's code is nothingCompatible, the client's capabilities are optionally included (The client's capabilities could be omitted for privacy reasons.)

If the _user_ code is noCompatibleMaterial, the provider MAY populate the clients list. For any other user code, the provider MUST NOT populate the clients list.

Keying material provided from one response MUST NOT be provided in any other response. The target provider MUST NOT provide expired keying material (ex: an MLS KeyPackage containing a LeafNode with a notAfter time past the current date and time).

```
enum {
    success(0);
    partialSuccess(1);
    incompatibleProtocol(2);
    noCompatibleMaterial(3);
    userUnknown(4);
    noConsent(5);
    noConsentForThisRoom(6);
    userDeleted(7);
    (255)
} KeyMaterialUserCode;

enum {
    success(0);
    keyMaterialExhausted(1),
    nothingCompatible(2),
    (255)
} KeyMaterialClientCode;


struct {
    KeyMaterialClientCode clientStatus;
    IdentifierUri clientUri;
    select (protocol) {
        case mls10:
            select (clientStatus) {
                case success:
                    KeyPackage keyPackage;
                case nothingCompatible:
                    optional<Capabilities> clientCapabilities;
            };
    };
} ClientKeyMaterial;

struct {
    Protocol protocol;
    KeyMaterialUserCode userStatus;
    IdentifierUri userUri;
    ClientKeyMaterial clients<V>;
} KeyMaterialResponse;
```

The semantics of the KeyMaterialUserCode are as follows:

* success indicates that key material was provided for every client of the target user.

* partialSuccess indicates that key material was provided for at least one client of the target user.

* incompatibleProtocol indicates that either one of providers supports the protocol requested, or none of the clients of the target user support the protocol requested.

* noCompatibleMaterial indicates that none of the clients was able to supply key material compatible with the requiredCapabilities field in the request.

* userUnknown indicates that the target user is not known to the target provider.

* noConsent indicates that the requester does not have consent to fetch key material for the target user. The target provider can use this response as a catch all and in place of other status codes such as userUnknown if desired to preserve the privacy of its users.

* noConsentForThisRoom indicates that the target user might have allowed a request for another room, but does not for this room. If the provider does not wish to make this distinction, it can return noConsent instead.

* userDeleted indicates that the target provider wishes the requester to know that the target user was previously a valid user of the system and has been deleted. A target provider can of course use userUnknown if the provider does wish to keep or specify this distinction.

The semantics of the KeyMaterialClientCode are as follows:

* success indicates that key material was provided for the specified client.

* keyMaterialExhausted indicates that there was no keying material available for the specified client.

* nothingCompatible indicates that the specified clients had no key material compatible with the requiredCapabilities field in the request.

At minimum, as each MLS KeyPackage is returned to a requesting provider (on behalf of a requesting IM client), the target provider needs to associate its KeyPackageRef with the target client and the hub provider needs to associate its KeyPackageRef with the target provider. This ensures that Welcome messages can be correctly routed to the target provider and client. These associations can be deleted after a Welcome message is forwarded or after the KeyPackage leaf_node.lifetime.not_after time has passed.

5.3. Update Room State

Adds, removes, and policy changes to the room are all forms of updating the room state. They are accomplished using the update transaction which is used to update the room base policy, participation list, or its underlying MLS group. It uses the HTTP POST method.

POST /update/{roomId}

Any change to the participant list or room policy (including authorization policy) is communicated via an AppSync proposal type with the applicationId of mimiParticipantList or mimiRoomPolicy respectively. When adding a user, the proposal containing the participant list change MUST be committed either before or simultaneously with the corresponding MLS operation.

Removing an active user from a participant list or banning an active participant likewise also happen simultaneously with any MLS changes made to the commit removing the participant.

A hub provider which observes that an active participant has been removed or banned from the room, MUST prevent any of its clients from sending or receiving any additional application messages in the corresponding MLS group; MUST prevent any of those clients from sending Commit messages in that group; and MUST prevent it from sending any proposals except for Remove and SelfRemove [I-D.ietf-mls-extensions] proposals for its users in that group.

The update request body is described below, using the RatchetTreeOption and PartialGroupInfo structs defined in [I-D.mahy-mls-ratchet-tree-options]:

```
struct {
  /* A Proposal or Commit which is either a PublicMessage; */
  /* or a SemiPrivateMessage                               */
  MLSMessage proposalOrCommit;
  select (proposalOrCommit.content.content_type) {
    case commit:
      /* Both the Welcome and GroupInfo omit the ratchet_tree */
      optional<Welcome> welcome;
      GroupInfoOption groupInfoOption;
      RatchetTreeOption ratchetTreeOption;
    case proposal:
      /* a list of additional proposals, each represented */
      /* as either PublicMessage or SemiPrivateMessage    */
      MLSMessage moreProposals<V>;
} HandshakeBundle;

enum {
  reserved(0),
  full(1),
  partial(2),
  (255)
} GroupInfoRepresentation;

struct {
   GroupInfoRepresentation representation;
   select (representation) {
     case full:
       GroupInfo groupInfo;
     case partial:
       PartialGroupInfo partialGroupInfo;
   }
} GroupInfoOption;

struct {
  select (room.protocol) {
    case mls10:
      HandshakeBundle bundle;
  };
} UpdateRequest;
```

The semantics of GroupInfoRepresentation are as follows:

* full means that the entire GroupInfo will be included.

* partial means that a PartialGroupInfo struct will be shared and that the Distribution Service is expected to reconstruct the GroupInfo as described in [I-D.mahy-mls-ratchet-tree-options].

For example, in the first use case described in the Protocol Overview, Alice creates a Commit containing an AppSync proposal adding Bob (mimi://b.example/b/bob), and Add proposals for all Bob's MLS clients. Alice includes the Welcome message which will be sent for Bob, a GroupInfo object for the hub provider, and complete ratchet_tree extension.

A handshake message could be sent by the client as an MLS PublicMessage (which is visible to all providers), or as an MLS SemiPrivateMessage [I-D.mahy-mls-semiprivatemessage] encrypted for the members and the hub provider as the sole external_receiver. (The contents and sender of a SemiPrivateMessage would not be visible to other providers). The use of SemiPrivateMessage allows the Hub to accomplish its policy enforcement responsibilities without the other providers being aware of the membership of non-local users.

The response body is described below:

```
enum {
  success(0),
  wrongEpoch(1),
  notAllowed(2),
  invalidProposal(3),
  (255)
} UpdateResponseCode;

struct {
  UpdateResponseCode responseCode;
  string errorDescription;
  select (responseCode) {
    case success:
      /* the hub acceptance time (in milliseconds from the UNIX epoch) */
      uint64 accepted_timestamp;
    case wrongEpoch:
      /* current MLS epoch for the MLS group */
      uint64 currentEpoch;
    case invalidProposal:
      ProposalRef invalidProposals<V>;
  };
} UpdateRoomResponse
```

The semantics of the UpdatedResponseCode values are as follows:

* success indicates the UpdateRequest was accepted and will be distributed.

* wrongEpoch indicates that the hub provider is using a different epoch. The currentEpoch is provided in the response.

* notAllowed indicates that some type of policy or authorization prevented the hub provider from accepting the UpdateRequest.

* invalidProposal indicates that at least one proposal is invalid. A list of invalidProposals is provided in the response.

5.4. Submit a Message

End-to-end encrypted (application) messages are submitted to the hub for authorization and eventual fanout using an HTTP POST request.

POST /submitMessage/{roomId}

The request body is as follows:

```
struct {
  Protocol protocol;
  select(protocol) {
    case mls10:
      /* PrivateMessage containing an application message */
      MLSMessage appMessage;
      IdentifierURI sendingUri;
  };
} SubmitMessageRequest;
```

If the protocol is MLS 1.0, the request body (appMessage) is an MLSMessage with a WireFormat of PrivateMessage, and a content_type of application. The sendingUri is a valid URI of the sender and is an active participant in the room (a user URI in the participant list of the room).

The response indicates if the message was accepted by the hub provider. If a franking_tag was included in the frank_aad component Section 10.1 in the PrivateMessage Additional Authenticated Data (AAD) in the request, the server attempts to frank the message and includes the server_frank in a successful response (see the next subsection).

```
enum {
  accepted(0),
  notAllowed(1),
  epochTooOld(2),
  (255)
} SubmitResponseCode;

struct {
  uint8[32] server_frank;
  opaque franking_integrity_signature<V>;
} Frank;

struct {
  Protocol protocol;
  select(protocol) {
    case mls10:
      SubmitResponseCode statusCode;
      select (statusCode) {
        case success:
          /* the hub acceptance time
             (in milliseconds from the UNIX epoch) */
          uint64 accepted_timestamp;
          optional Frank frank;
        case epochTooOld:
          /* current MLS epoch for the MLS group */
          uint64 currentEpoch;
      };
  };
} SubmitMessageResponse;
```

The semantics of the SubmitResponseCode values are as follows:

* success indicates the SubmitMessageRequest was accepted and will be distributed.

* notAllowed indicates that some type of policy or authorization prevented the hub provider from accepting the UpdateRequest. This could include nonsensical inputs such as an MLS epoch more recent than the hub’s.

* epochTooOld indicates that the hub provider is using a new MLS epoch for the group. The currentEpoch is provided in the response.

  ISSUE: Do we want to offer a distinction between regular application messages and ephemeral applications messages (for example "is typing" notifications), which do not need to be queued at the target provider.

5.4.1. Message Franking

Franking is the placing of a cryptographic "stamp" on a message. In the MIMI context, the Hub is able to mark that it received a message without learning the message content. A receiver that decrypts the message can use a valid frank to prove it was received by the Hub and that the content was sent by a specific sender. Outsiders (including follower providers) never learn the content of the message, nor the sender.

Franking was popularized by Facebook and described in their whitepaper [SecretConversations] about their end-to-end encryption system. This franking mechanism is largely motivated by that solution with two significant changes as discussed in the final paragraph of this section.

MIMI franking relies on two new MLS application components. The first is the frank_aad Safe AAD component Section 4.9 of [I-D.ietf-mls-extensions]. The second is the franking_agent GroupContext component Section 4.6 of [I-D.ietf-mls-extensions]. The structure of franking_agent mirrors that of the ExternerSender struct described in Section 12.1.8.1 of [RFC9420]. It contains a single signature key.

```
struct {
  uint8[32] franking_tag
} FrankAAD;

FrankAAD frank_aad;

struct {
  SignaturePublicKey franking_signature_key;
  Credential credential;
} FrankingAgentData;

FrankingAgentData franking_agent;
FrankingAgentData FrankingAgentUpdate;
```

The signature algorithm of the franking_signature_key is the same as the signature scheme in the cipher suite of the MLS group.

FrankingAgentUpdate is the format of the update field inside the AppDataUpdate struct in an AppDataUpdate Proposal for the franking_agent component. The use of AppDataUpdate on the franking_agent is RECOMMENDED only when adding a new franking_agent. To avoid confusion about which signature key to use, when an MLS client rotates the franking_agent, the client SHOULD create a new MLS group by sending a ReInit proposal.

5.4.1.1. Client creation and sending

The franking mechanism requires any franked message format to contain a random salt, the sender URI, and the room URI. The section describes the exact way to extract these fields from the MIMI content format [I-D.ietf-mimi-content]. Other message formats would need to describe how to locate or derive these values.

The MIMI content format includes a per-message, mandatory, cryptographically random 128-bit salt generated by the sender. (An example mechanism to safely generate the salt is discussed in Section 8.2 of [I-D.ietf-mimi-content].)

The sender of a MIMI content message attaches to the message any fields the sender wishes to commit that are not otherwise represented in the content. For a MIMI content object, the sender includes the sender's user URI and room URI in the MIMI content extensions map (see definitions in Section 8.3 of [I-D.ietf-mimi-content]).

```
/ MIMI content extensions map /
{
  / sender_uri / 1: "mimi://b.example/u/alice"
  / room_uri   / 2: "mimi://hub.example/r/Rl33FWLCYWOwxHrYnpWDQg",
}
```

Note that these assertions do not vouch for the validity of these values, it just means that the sender is claiming it sent the values in the content, and cannot later deny to a receiver that it sent them.

Then the client calculates the franking_tag, as the HMAC SHA256 of the application_data (which includes the values in the extensions map above) using the salt in the MIMI content format:

```
franking_tag = HMAC_SHA256( salt, application_data)
```

The client includes the franking_tag in the Additional Authenticated Data of the MLS PrivateMessage using the frank_aad Safe AAD component. The client uses the MIMI submitMessage to send its message, and also asserts a sender identity to the Hub, which could be a valid pseudonym, and needs to match the sender URI value embedded in the message. If the message is accepted, the response includes the accepted timestamp and the server_frank (generated by the server).

5.4.1.2. Hub processing

When a valid franking_agent application component Section 10.2 is present in the GroupContext, the Hub can frank messages, and sign the frank with the Hub's franking_private_key corresponding to the franking_signature_key in the FrankingAgentData struct.

When the Hub receives an acceptable application message with the frank_aad Safe AAD component and a valid sender identity, it calculates a server frank for the message, and a franking_integrity_signature as follows:

```
context = sender_uri || room_uri || accepted_timestamp
server_frank = HMAC_SHA256(hub_key, franking_tag || context )
franking_integrity_signature =
  Signature.Sign(franking_private_key, server_frank || context)
```

hub_key is a secret symmetric key used on the Hub which the Hub can use to verify its own server_frank.

The franking_integrity_signature is used by receivers to verify that the values added by the Hub (the server_frank, and accepted_timestamp) were not modified by a follower provider, and that the sender_uri and room_uri match those provided by the sending client. The specific construction used is discussed in the Security Considerations in Section 9.1.

The Hub fans out the encrypted message (which includes the franking_tag), the server_frank, the accepted_timestamp, the room URI, and the franking_integrity_signature. Note that the sender_uri is encrypted in the application message, so the sender can remain anonymous with respect to follower providers.

5.4.1.3. Receiver verification of frank

When a client receives and decrypts an otherwise valid application message from a hub provider, the client looks for the existence of a frank (consisting of the franking_tag in the AAD, the server_frank and the franking_integrity_signature). If those fields are available, the client looks for the franking_agent application component in the GroupContext. It verifies the domain name in the franking_agent.credential corresponds to the domain of the Hub, and extracts the franking_signature_key.

Next it verifies the integrity of the server_frank, accepted_timestamp, sender_uri, and room_uri by calculating the signed_content and verifying the franking_integrity_signature as described below.

```
signed_content = server_frank ||
                   sender_uri || room_uri || accepted_timestamp
Signature.Verify(franking_signature_key, signed_content,
                            franking_integrity_signature)
```

Finally it verifies the construction of the franking_tag from the content format of the message (including the embedded salt), that the sender's identity in its credential in its MLS LeafNode matches the sender_uri asserted in the extensions map inside the MIMI Content, and that the room_uri asserted in the extensions map inside the MIMI Content matches the room ID in the received message.

The receiver needs to store the server_frank, franking_integrity_signature, and context fields with the decoded message, so they can be used later.

5.4.1.4. Comparison with the Facebook franking scheme

Unlike in the Facebook franking scheme [SecretConversations], the MIMI use case involves traffic which can transit multiple federated providers, any of which may be compromised or malicious. The MIMI franking scheme described here differs in the following ways.

Firstly, the sender includes its franking_tag application component as Additional Authenticated Data (AAD) inside the end-to-end encrypted message. This insures that the franking_tag is not tampered with by the sender's provider. According to [Grubbs2017], "... [Facebook's] franking scheme does not bind [the franking tag] to [the ciphertext] by including [the franking tag] in the associated data during encryption".

In MLS, the Hub cannot view the sender identity in an application message, so the sender sends its identity to the Hub. The hub never sends the identity of the sender to receivers, since this would be observed by follower providers. However, the receiver needs to verify that the sender identity provided by the sender's provider to the Hub matches the identity the receiver sees after it decrypts the message. Using the franking_integrity_signature generated by the Hub, receivers can verify the message context (sender identity, room id, and timestamp) and the server_frank with the franking_signature_key in the franking_agent component in the GroupContext. This second change provides two functions. It allows receivers to validate the sender_uri in the hub's context, without revealing the sender URI to follower providers. It also prevents a follower provider from modifying the server_frank, or any elements of the context (ex: modifying the accepted_timestamp) without detection. A valid signature ensures that follower providers did not tamper with the context or the server_frank.

5.5. Fanout Messages and Room Events

If the hub provider accepts an application or handshake message (proposal or commit) message, it forwards that message to all other providers with active participants in the room and all local clients which are active members. This is described as fanning the message out. One can think of fanning a message out as presenting an ordered list of MLS-protected events to the next "hop" toward the receiving client.

An MLS Welcome message is sent to the providers and local users associated with the KeyPackageRef values in the secrets array of the Welcome. In the case of a Welcome message, a RatchetTreeOption (see Section 3 of [I-D.mahy-mls-ratchet-tree-options]) is also included in the FanoutMessage.

The hub provider also fans out any messages which originate from itself (ex: MLS External Proposals).

> An external commit will invalidate certain pending proposals. For example, if the hub generates Remove proposals to remove a lost client or a deleted user, an external commit can invalidate pending Remove proposals. The hub would then be expected to regenerate any relevant, comparable proposals in the new epoch. To prevent a race condition where a member commit arrives before the regenerated proposals arrive, the hub can staple regenerated proposals to an external commit during the fanout process.

The hub can include multiple concatenated FanoutMessage objects relevant to the same room. This endpoint uses the HTTP POST method.

POST /notify/{roomId}

```
struct {
  /* the hub acceptance time (in milliseconds from the UNIX epoch) */
  uint64 timestamp;
  select (protocol) {
    case mls10:
      /* A PrivateMessage containing an application message, */
      /* a SemiPrivateMessage or PublicMessage containing a  */
      /* proposal or commit, or a Welcome message.           */
      MLSMessage message;
      select (message.wire_format) {
        case application:
           optional Frank frank;
        case welcome:
           RatchetTreeOption ratchetTreeOption;
        case proposal:
           /* a list of additional proposals, each represented */
           /* as either PublicMessage or SemiPrivateMessage    */
           MLSMessage moreProposals<V>;
        case commit:
           /* If this was an external commit, and any pending      */
           /* proposals were invalidated, staple the new epoch's   */
           /* replacement proposals (from the hub) to the commit   */
           /* commit */
           MLSMessage externalProposals<V>;
      };
  };
} FanoutMessage;
```

NOTE: Correctly fanning out Welcome messages relies on the hub and target providers storing the KeyPackageRef of claimed KeyPackages.

A client which receives a success to either an UpdateRoomResponse or a SubmitMessageResponse can view this as a commitment from the hub provider that the message will eventually be distributed to the group. The hub is not expected to forward the client's own message to the client or its provider. However, the client and its provider need to be prepared to receive the client's (effectively duplicate) message. This situation can occur during failover in high availability recovery scenarios.

Clients that are being removed SHOULD receive the corresponding Commit message, so they can recognize that they have been removed and clean up their internal state. A removed client might not receive a commit if it was removed as a malicious or abusive client, or if it was obviously deleted.

The moreProposals list in a FanoutMessage MUST be the same as the corresponding moreProposals list in the HandshakeBundle of an UpdateRequest.

The response to a FanoutMessage contains no body. The HTTP response code indicates if the messages in the request were accepted (201 response code), or if there was an error. The hub need not wait for a response before sending the next fanout message.

If the hub server does not contain an HTTP 201 response code, then it SHOULD retry the request, respecting any guidance provided by the server in HTTP header fields such as Retry-After. If a follower server receives a duplicate request to the /notify endpoint, in the sense of a request from the same hub server with the same request body as a previous /notify request, then the follower server MUST return a 201 Accepted response. In such cases, the follower server SHOULD process only the first request; subsequent duplicate requests SHOULD be ignored (despite the success response).

> *NOTE:* These deduplication provisions require follower servers to track which request bodies they have received from which hub servers. Since the matching here is byte-exact, it can be done by keeping a rolling list of hashes of recent messages.
>
> This byte-exact replay criterion might not be the right deduplication strategy. There might be situations where it is valid for the same hub server to send the same payload multiple times, e.g., due to accidental collisions.
>
> If this is a concern, then an explicit transaction ID could be introduced. The follower server would still have to keep a list of recently seen transaction IDs, but deduplication could be done irrespective of the content of request bodies.

5.6. Claim group key information

When a client joins an MLS group without an existing member adding the client to the MLS group, that is called an external join. This is useful a) when a new client of an existing user needs to join the groups of all the user’s rooms. It can also be used b) when a client did not have key packages available but their user is already in the participation list for the corresponding room, c) when joining an open room, or d) when joining using an external authentication joining code. In MIMI, external joins are accomplished by fetching the MLS GroupInfo for a room’s MLS group, and then sending an external commit incorporating the GroupInfo.

The GroupInfoRequest uses the HTTP POST method.

POST /groupInfo/{roomId}

The request provides an MLS credential proving the requesting client's real or pseudonymous identity. This user identity is used by the hub to correlate this request with the subsequent external commit. The credential may constitute sufficient permission to authorize providing the GroupInfo and later joining the group. Alternatively, the request can include an optional opaque joining code, which the requester could use to prove permission to fetch the GroupInfo, even if it is not yet a participant.

The request also provides a signature public key corresponding to the requester's credential. It also specifies a CipherSuite which merely needs to be one ciphersuite in common with the hub. It is needed only to specify the algorithms used to sign the GroupInfoRequest and GroupInfoResponse.

```
struct {
  Protocol protocol;
  select (protocol) {
    case mls10:
      CipherSuite cipher_suite;
      SignaturePublicKey requestingSignatureKey;
      Credential requestingCredential;
      HPKEPublicKey groupInfoPublicKey;
      optional opaque joiningCode<V>;
  };
} GroupInfoRequestTBS;

struct {
  Protocol protocol;
  select (protocol) {
    case mls10:
      CipherSuite cipher_suite;
      SignaturePublicKey requestingSignatureKey;
      Credential requestingCredential;
      HPKEPublicKey groupInfoPublicKey;
      opaque joiningCode<V>;
      /* SignWithLabel(requestingSignatureKey,          */
      /*    "GroupInfoRequestTBS", GroupInfoRequestTBS) */
      opaque signature<V>;
  };
} GroupInfoRequest;
```

If successful, the response body contains the GroupInfo and a way to get the ratchet_tree, both encrypted with the groupInfoPublcKey passed in the request.

```
   enum {
     reserved(0),
     success(1),
     notAuthorized(2),
     noSuchRoom(3),
     (255)
   } GroupInfoCode;

   struct {
     GroupInfo groupInfo;   /* without embedded ratchet_tree */
     RatchetTreeOption ratchetTreeOption;
   } GroupInfoRatchetTreeTBE;

   GroupInfoRatchetTreeTBE group_info_ratchet_tree_tbe;

   encrypted_groupinfo_and_tree = EncryptWithLabel(
     groupInfoPublicKey,
     "GroupInfo and ratchet_tree encryption",
     room_id,                          /* context */
     group_info_ratchet_tree_tbe)

   struct {
     Protocol version;
     opaque room_id<V>;
     GroupInfoCode status;
     select (protocol) {
       case mls10:
         CipherSuite cipher_suite;
         ExternalSender hub_sender;
         HPKECiphertext encrypted_groupinfo_and_tree;
     };
   } GroupInfoResponseTBS;

   struct {
     Protocol version;
     opaque room_id<V>;
     GroupInfoCode status;
     select (status) {
       case success:
         select (protocol) {
           case mls10:
             CipherSuite cipher_suite;
             ExternalSender hub_sender;
             HPKECiphertext encrypted_groupinfo_and_tree;
             /* SignWithLabel(hub_sender.signature_key,        */
             /*  "GroupInfoResponseTBS", GroupInfoResponseTBS) */
             opaque signature<V>;
         };
```

```
    default: struct{};
    };
} GroupInfoResponse;
```

The semantics of the GroupInfoCode are as follows:

* success indicates that GroupInfo and ratchet tree was provided as requested.

* notAuthorized indicates that the requester was not authorized to access the GroupInfo.

* noSuchRoom indicates that the requested room does not exist. If the hub does not want to reveal if a room ID does not exist it can use notAuthorized instead.

  TODO: Consider adding additional failure codes/semantics for joining codes (ex: code expired, already used, invalid)

ISSUE: What security properties are needed to protect a GroupInfo object in the MIMI context are still under discussion. It is possible that the requester only needs to prove possession of their private key. The GroupInfo in another context might be sufficiently sensitive that it should be encrypted from the end client to the hub provider (unreadable by the local provider).

5.7. Convey explicit consent

As discussed in Section 8, there are many ways that a provider could implicitly determine consent. This section describes a mechanism by which providers can explicitly request consent from a user of another provider, cancel such a request, convey that consent was granted, or convey that consent was revoked or preemptively denied.

Since they are not necessarily in the context of a room, consent requests are sent directly from the provider of the user requesting consent, to the provider of the target user. (There is no concept of a hub outside of the context of a room.)

```
POST /requestConsent/{targetDomain}
POST /updateConsent/{requesterDomain}
```

A requestConsent request is used by one provider to request explicit consent from a target user at another provider to fetch the target's KeyPackages (which is a prerequisite for adding the target to a group); or to cancel that request. The request body is a ConsentEntry, with a consentOperation of request or cancel respectively. It includes the URI of requesting user in the

requesterUri and the target user URI in the targetUri. If consent is only requested for a single room, the requester includes the roomId. The combination of the requesterUri, targetUri, and optional roomId represents the ConsentScope. A cancel MUST use the same ConsentScope as a previous request.

For a requestContent, the targetUri needs to be in one of the domains of the receiving provider, and the requesterUri needs to be in one of the domains of the sending provider.

The response to a requestConsent request is usually a 201 Accepted (indicating the requestConsent was received), optionally a 404 Not Found (indicating the targetUri is unknown), or a 500-class response. The 201 response code merely indicates that the request was received. A provider that does not wish to reveal if a user is not found can respond with a 201 Accepted. Likewise in response to a cancel which has no request matching the ConsentScope, a 201 Accepted is sent and no further action is taken.

```
enum {
  cancel(0),
  request(1),
  grant(2),
  revoke(3),
  (255)
} ConsentOperation;

struct {
  ConsentOperation consentOperation;
  IdentifierUri requesterUri;
  IdentifierUri targetUri;
  optional<RoomId> roomId;
  select (consentOperation) {
      case grant:
          KeyPackage clientKeyPackages<V>;
  };
} ConsentEntry;

struct {
  IdentifierUri requesterUri;
  IdentifierUri targetUri;
  optional<RoomId> roomId;
} ConsentScope;
```

An updateConsent request is used by one provider to provide explicit notice from a target user at one provider that consent for a specific "requester" was granted, revoked, or preemptively denied. In this context, the requester is the party that will later request

KeyPackages for the target. The request body is a ConsentEntry, with a consentOperation of grant (for a grant), or revoke for revocation or denial. Like a request, it includes the URI of the "requesting user" in the requesterUri and the target user URI in the targetUri. If consent is only granted or denied for a single room, the request includes the optional roomId.

A grant or revoke does not need to be in response to an explicit request, nor does the ConsentScope need to match a previous request for the same targetUri and requesterUri pair.

For example, in some systems there is a notion of a bilateral connection request. The party that initiates the connection request (for example Alice) would send a requestConsent for the target (ex: Bob), and send an unsolicited updateConsent with Bob as the "requestor" and itself (Alice) as the target.

In a grant, the sender includes a list of clientKeyPackages for the target user, which can be empty. For the case of a bilateral connection, a grant of consent with a matching ConsentScope often results in an immediate Add to a group. If the list is non-empty this reduces the number of messages which need to be sent.

For updateConsent the requesterUri needs to be in one of the domains of the receiving provider, and the targetUri needs to be in one of the domains of the sending provider.

The response to an updateConsent is usually a 201 Accepted (indicating the updateConsent was received), optionally a 404 Not Found (indicating the requesterUri is unknown), or a 500-class response. The response code merely indicates that the request was received. A provider that does not wish to reveal if a user is not found can respond with a 201 Accepted.

> *NOTE*: Revoking consent for a user might be privacy sensitive. If this is the case the target provider does not need to send a revoke to inform the requester provider.

5.8. Find internal address

The identifier query is to find the internal URI for a specific user on a specific provider. It is only sent from the local provider to the target provider (it does not transit a hub).

Note that this POST request is idempotent and safe in the sense defined by Section 9.2.2 of [RFC9110].

POST /identifierQuery/{domain}

Consider three users Xavier, Yolanda, and Zach all with accounts on provider XYZ. Xavier is a sales person and wants to be contactable easily by potential clients on the XYZ provider. He configures his profile on XYZ so that searching for his first or last name or handle will find his profile and allow Alice to send him a consent request (it is out of scope how Alice verifies she has found the intended Xavier and not a different Xavier or an impostor). Yolanda has her XYZ handle on her business cards and the email signature she uses with clients. She configures her profile so that a query for her exact handle will find her profile and allow Alice to send her a consent request. Zach does not wish to be queryable at all. He has configured his account so even an exact handle search returns no results. He could still send a join link out-of-band to Alice for her to join a room of Zach’s choosing.

The request body is described as below. Each request can contain multiple query elements, which all have to match for the request to match (AND semantics). For example matching both the OpenID Connect (OIDC) [OidcCore] given_name and family_name, or matching the OIDC given_name and the organization (from the vCard [RFC6350] ORG property).

```
enum {
  reserved(0),
  handle(1),
  nick(2),
  email(3),
  phone(4),
  partialName(5),
  wholeProfile(6),
  oidcStdClaim(7),
  vcardField(8),
  (255)
} SearchIdentifierType;

struct {
  SearchIdentifierType searchType;
  select(type) {
    case oidcStdClaim:
      opaque claimName<V>;
    case vcardField:
      opaque propertyName<V>;
  };
  opaque searchValue<V>;  /* a UTF8 string */
} QueryElement;

struct {
  QueryElement query_elements<V>;
} IdentifierRequest;
```

The semantics of the SearchIdentifierType values are as follows. handle means that the entire handle URI matches exactly (for example: im:alice.smith@a.example). nick means that the nickname or handle user part matches exactly (for example: alice.smith). The same account or human user may have multiple values which all match the nick field. email means the addr-spec production from [RFC5322] matches the query string exactly, for example (asmith@a.example). phone means the international format of a telephone number with the "+" prefix matches exactly (for example: +12125551212).

partialName means that the query string matches a case-insensitive substring of any field which contains the name of a (usually human) user. For example, mat would match first (given) or middle names Matt, Matthew, Mathias, or Mathieu and last (family) names of Mather and Matali. wholeProfile means that the query string matches a substring of any searchable field in a user’s profile.

oidcStdClaim means that the query string exactly matches the specified UserInfo Standard Claim (defined in Section 5.1 of [OidcCore]). vcardField means that the query string exactly matches the specified vCard property listed in the vCard Properties IANA registry.

As noted above, searches only return results for a user when the fields searched are searchable according the user’s and provider’s search policies.

The response body is described as an IdentifierResponse. It can contain multiple matches depending on the type of query and the policy of the target provider.

The response contains a code indicating the status of the query. success means that at least one result matched the query. notFound means that while the request was acceptable, no results matched the query. ambiguous means that a field (ex: handle) or combination of fields (ex: first and last name) need to match exactly for the provider to return any responses. forbidden means that use of this endpoint is not allowed by the provider or that an unspecified field or combination of fields is not allowed in an identifier query. unsupportedField means that the provider does not support queries on one of the fields queried.

```
enum {
  success(0),
  notFound(1),
  ambiguous(2),
  forbidden(3),
  unsupportedField(4),
  (255)
} IdentifierQueryCode;

enum {
  reserved(0),
  oidcStdClaim(7),
  vcardField(8),
  (255)
} FieldSource;

struct {
  FieldSource fieldSource;
  string fieldName;
  opaque fieldValue<V>;
} ProfileField;

struct {
  IdentifierUri stableUri;
  ProfileField fields<V>;
} UserProfile;

struct {
  IdentifierQueryCode responseCode;
  IdentifierUri uri<V>;
  UserProfile foundProfiles<V>;
} IdentifierResponse;
```

TODO: The format of specific identifiers is discussed in [I-D.mahy-mimi-identity]. Any specific conventions which are needed should be merged into this document.

5.9. Report abuse

Abuse reports are only sent to the hub provider. They are sent as an HTTP POST request.

```
POST /reportAbuse/{roomId}
```

The reportingUser optionally contains the identity of the user sending the abuseReport, while the allegedAbuserUri contains the URI of the alleged sender of abusive messages. The reasonCode is reserved to identify the type of abuse, and the note is a UTF8 human-readable string, which can be empty.

TODO: Find a standard taxonomy of reason codes to reference for the AbuseType. The IANA Messaging Abuse Report Format parameters are insufficient.

Finally, abuse reports can optionally contain a handful of allegedly AbusiveMessages, each of which contains an allegedly abusive message_content, its server_frank, its franking_integrity_signature, and its accepted_timestamp.

```
struct {
  /* the message content (ex: MIMI Content message) containing */
  /* allegedly abusive content                                 */
  opaque message_content<V>;
  Frank frank;
  uint64 accepted_timestamp;
} AbusiveMessage;

enum {
  reserved(0),
  (255)
} AbuseType;

struct {
  IdentifierUri reportingUser;
  IdentifierUri allegedAbuserUri;
  AbuseType reasonCode;
  opaque note<V>;
  AbusiveMessage messages<V>;
} AbuseReport;
```

There is no response body. The response code only indicates if the abuse report was accepted, not if any specific automated or human action was taken.

To validate an allegedly AbusiveMessage, the hub finds the salt, sender URI, and room URI inside the message_content and the accepted_timestamp to recalculate the franking_tag and context. Then the hub selects its relevant hub_key to regenerate the server_frank. Finally the hub verifies its franking_integrity_signature.

5.10. Download Files

IM systems make extensive use of inline images, videos, and sounds, and often include attached files, all of which will be referred to as "assets" in this section. Assets are stored (encrypted) on various external servers, and typically uploaded and fetched using HTTP [RFC9110] protected with TLS [RFC8446].

> In a MIMI content [I-D.ietf-mimi-content] message, the download URI is an https URL conveyed in the uri field in an ExternalPart in a MIMI content message.

Broadly, two approaches are possibly for storage of assets in federated IM systems. In the vast majority of deployed systems, the assets are uploaded to the local provider of the uploader. With the other approach, the assets are uploaded to the equivalent of the Hub system. MIMI is capable of asset downloads for both of these approaches.

The client consults the intersection of the room policy and its local policy to determine if and how to upload assets, where to upload them, and with which credentials. The details of the upload process are out of scope of this document.

The MIMI room policy [I-D.ietf-mimi-room-policy] in the group’s GroupContext includes an asset_policy component that specifies which domain name to use for asset download for each potential asset provider. This prevents the download proxy from using a host part associated with the same domain that is not used for the storage of MIMI assets.

For downloads, the clients can use any of three methods to download assets. The hub is not involved in the first and is required to implement support for the other two methods.

5.10.1. Direct download

Without any additional MIMI protocol mechanism, MIMI clients can download assets directly from the asset provider. Unfortunately this usually reveals sensitive private information about the client. In this case, the asset provider learns the IP address of the client, and timing information about when assets are downloaded, which is strongly linked with online presence. The asset provider can correlate other clients downloading the same assets and infer which clients are in which rooms. For this reason, direct client downloads of assets, especially from an asset provider which is not the download client’s provider, are NOT RECOMMENDED.

5.10.2. Download using a hub proxy

The hub offers the proxyDownload endpoint to proxy asset requests from downloading providers to known asset providers. This allows clients to hide their IP addresses from the asset provider (and the hub), although the hub and the downloading provider are still privy to the request.

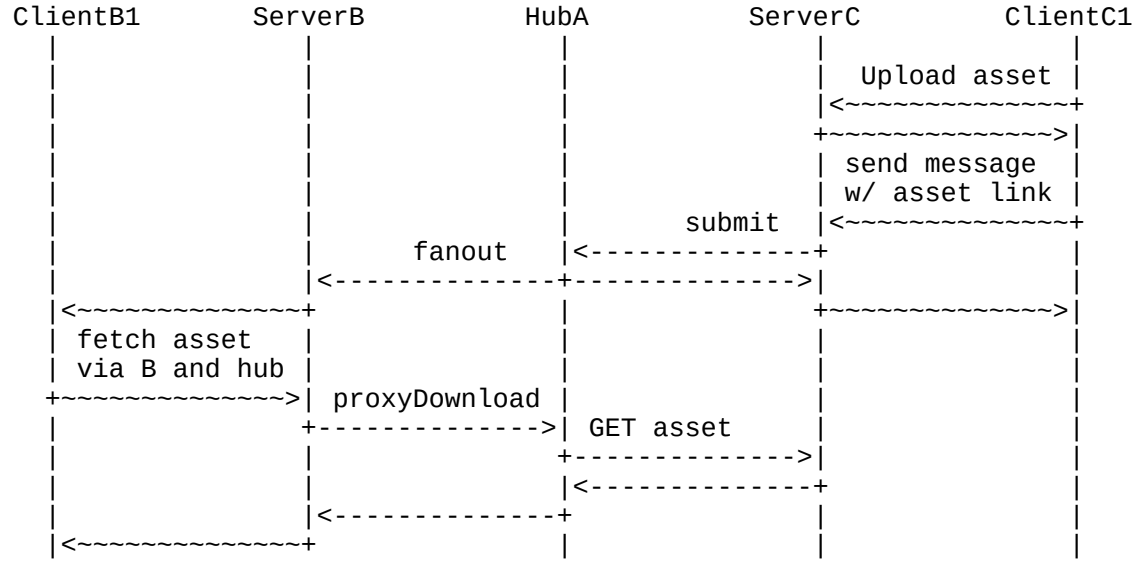
The proxyDownload endpoint includes the target asset URL, but does not include the specific room or requester. When the provider hosting the asset has several rooms hosted by the hub, the provider also does not directly learn with which room to associate a specific asset.

As with other MIMI protocol endpoints, the actual endpoint URL is discovered using the MIMI protocol directory Section 5.1.

GET /proxyDownload/{downloadUrl}

The downloading provider sends the request to the hub. The hub MUST verify that the host part of the downloadUrl is associated with the asset server domain for one of its peer providers. This prevents the hub from becoming an open proxy.

The overall flow is as shown below.



If the request succeeds, the response body contains the contents of the downloaded URL. The hub can reject proxyDownload requests by replying with standard HTTP error codes.

In terms of the privacy of this mechanism, the hub only sees that a specific provider requested an (opaque to the hub) URL from another provider. It does not know the room or the specific client. The asset provider sees requests for an (encrypted) asset coming from the hub provider. When caching is employed, the asset provider will only see periodic requests. The receiver's provider will see one request to the hub, among presumably many requests. It has no information about the sender, URL, or room associated with that request, however it can collect the domains of specific asset providers, and it can use timing analysis to correlate incoming messages (which are associated with a room) and outgoing proxyDownload requests.

Both the downloading provider and the hub can cache requests. This would tend to hide from the asset provider the number of downloading participants in a specific room, and make it more difficult for the hub to correlate a URL with a specific room based on the number of participants.

The proxy download method improves the privacy of the clients with respect to the asset provider, and to some extent with the hub, but it still allows the downloading provider to correlate URLs with downloading users and infer which URLs are sent in which rooms.

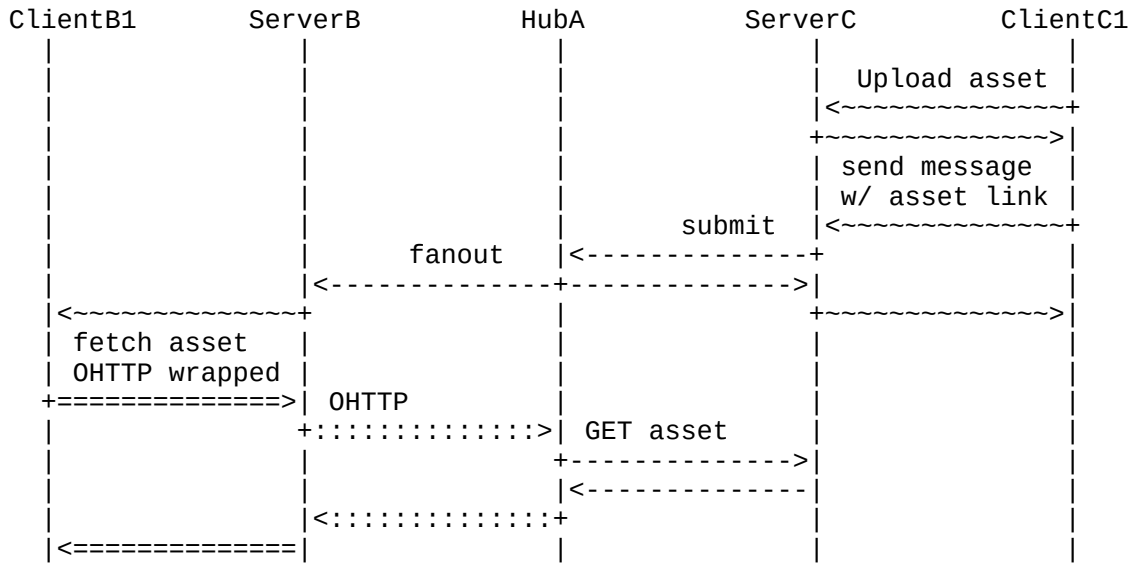
The hub is REQUIRED to implement this mechanism.

5.10.3. Download using Oblivious HTTP

Oblivious HTTP (OHTTP) [RFC9458] is a mechanism for the encapsulation of HTTP requests that provides a variety of desirable privacy properties. It provides the best privacy of the download mechanisms described here, but requires the client to implement an additional protocol.

Using OHTTP for asset download privacy works as follows. The clients sends an OHTTP request to the downloading provider's OHTTP Relay which forwards the request to the hub's OHTTP Gateway. The hub then sends a GET request for the asset to the asset provider (the Target)

The hub MUST only accept OHTTP requests to Targets which are configured as asset storage locations for one of its peer providers.



For the hub, an OHTTP Gateway is REQUIRED to implement. The hub SHOULD provide the OHTTP Gateway capability for its peer providers.

Other providers SHOULD implement the OHTTP Relay capability and SHOULD enable the relay to access

The main benefit of this mechanism is that the downloading provider does not learn the domains of every asset provider.

6. Minimal metadata rooms

The room state is visible to the hub and with it the room’s participant list, giving the hub access to a significant amount of user metadata.

To limit the amount of metadata the hub has access to, rooms can be created as _minimal metadata rooms_ (MMR). In an MMR the participant list and the credentials in the room’s underlying MLS group consist only of pseudonyms. The real identifiers are stored alongside the pseudonyms encrypted under a key known only to room participants, but not the hub.

MMRs requires some additional key management, which leads to restrictions in how the MMR can be joined and which users each participant can add to the room.

6.1. Credential encryption

Identifiers of participants and their clients occur in two locations in a room's state: the participant list and the credentials of the room's underlying MLS group. In an MMR, the real identifiers of clients and users are replaced by pseudonyms in the shape of random UUIDs qualified with the domain of the user's provider.

In MMRs, all leaves of the underlying group MUST contain PseudonymousCredentials.

```
struct {
  IdentifierUri client_pseudonym;
  IdentifierUri user_pseudonym;
  opaque signature_public_key;
  opaque identity_link_ciphertext<V>;
} PseudonymousCredential
```

* user_pseudonym: The pseudonym of the client's user in this group

* client_pseudonym: The pseudonym of the client identified by this credential

* signature_public_key: The signature public key used to authenticate MLS messages

* identity_link_ciphertext: A ciphertext containing a credential with the clients real identifier

In any given room, the user_pseudonym of a client MUST be the same across all clients of a user and it MUST be the same as the user's entry in the participant list.

```
struct {
  IdentifierUri client_pseudonym;
  IdentifierUri user_pseudonym;
  opaque signature_public_key;
} PseudonymousCredentialTBS

struct {
  /* SignWithLabel(., "PseudonymousCredentialTBS",
    PseudonymousCredentialTBS) */
  opaque pseudonymous_credential_signature<V>;
  Credential client_credential;
} IdentityLinkTBE
```

The identity_link_ciphertext is created by encrypting the IdentityLinkTBE. The IdentityLinkTBE contains the client's real credential, and a signature over the PseudonymousCredentialTBS signed with the client credential's signature_public_key.

TODO: Specify a key management scheme that ideally - is efficient - allows the basic MIMI flows - ensures that all participants can learn the identities of all other participants at all times - provides FS and PCS w.r.t. metadata hiding

There are several options that represent different trade-offs, but are not yet fully specified. They will be added at a later date.

7. Relation between MIMI state and cryptographic state

7.1. Room state

The state of a room consists of its room ID, its base policy, its participant list (including the role and participation state of each participant), and the associated end-to-end protocol state (its MLS group state) that anchors the room state cryptographically.

While all parties involved in a room agree on the room's state during a specific epoch, the Hub is the arbiter that decides if a state change is valid, consistent with the room's then-current policy. All state-changing events are sent to the Hub and checked for their validity and policy conformance, before they are forwarded to any follower servers or local clients.

As soon as the Hub accepts an event that changes the room state, its effect is applied to the room state and future events are validated in the context of that new state.

The room state is thus changed based on events, even if the MLS proposal implementing the event was not yet committed by a client. Note that this only applies to events changing the room state.

7.2. Cryptographic room representation

Each room is represented cryptographically by an MLS group. The Hub that manages the room also manages the list of group members, i.e. the list of clients belonging to users currently in the room. Application state that is stored in the MLS GroupContext is stored as application components in the app_data_dictionary extension, as described in Section 4.6 of [I-D.ietf-mls-extensions].

7.3. Proposal-commit paradigm

The MLS protocol follows a proposal-commit paradigm. Any party involved in a room (follower server, Hub or clients) can send proposals (e.g. to add/remove/update clients of a user or to re-initialize the group with different parameters). However, only clients can send commits, which contain all valid previously sent proposals and apply them to the MLS group state.

The MIMI usage of MLS ensures that the Hub, all follower servers and the clients of all active participants agree on the group state, which includes the client list and the key material used for message encryption (although the latter is only available to clients). Since the group state also includes a copy of the room state at the time of the most recent commit, it is also covered by the agreement.

7.4. Authenticating proposals

MLS requires that MLS proposals from the Hub and from follower servers (external senders in MLS terminology) be authenticated using key material contained in the external_senders extension of the MLS group. Each MLS group associated with a MIMI room MUST therefore contain an external_senders extension. That extension MUST contain at least the Certificate of the Hub.

When a user from a follower server becomes a participant in the room, the Certificate of the follower server MAY be added to the extension. When the last participant belonging to a follower server leaves the room, the certificate of that user MUST be removed from the list. Changes to the external_senders extension only take effect when the MLS proposal containing the event is committed by a MIMI commit.

7.5. Participant List

The participant list is a list of "users" in a room. Within a room, each user is assigned exactly one _role_ (expressed with a role_index and described in [I-D.ietf-mimi-room-policy] at any given time (specifically within any MLS epoch). In a room that has multiple MLS clients per "user", the identifier for each user in participants.user is the same across all that user’s clients in the room. Note that each user has a single role at any point in time, and therefore all clients of the same user also have the same role.

The participant list may include inactive participants, which currently do not have any clients in the corresponding MLS group, for example if their clients do not have available KeyPackages or if all of their clients are temporarily "kicked" out of the group. The participant list can also contain participants that are explicitly banned, by assigning them a suitable role which does not have any capabilities.

```
struct {
  opaque user<V>;
  uint32 role_index;
} UserRolePair;

struct {
  UserRolePair participants<V>;
} ParticipantListData;

ParticipantListData participant_list;
```

ParticipantListData is the format of the data field inside the ComponentData struct for the Participant list Metadata component in the app_data_dictionary GroupContext extension.

```
struct {
  uint32 user_index;
  uint32 role_index;
} UserindexRolePair;

struct {
  UserindexRolePair changedRoleParticipants<V>
  uint32 removedIndices<V>;
  UserRolePair addedParticipants<V>;
} ParticipantListUpdate;
```

ParticipantListUpdate is the contents of an AppDataUpdate Proposal with the component ID for the participant list. The index of the participants vector in the current ParticipantListData struct is referenced as the user_index when making changes. First the changedRoleParticipants list contains UserindexRolePairs with the index of a user who changed roles and their new role. Next is the removedIndices list which has a list of users to remove completely from the participant list. Finally there is a list of addedParticipants (which contains a user and role) that is appended to the end of the ParticipantListData.

Each of these actions (modifying a user's role, removing a user, and adding a user) is authorized separately according to the rules specified in [I-D.ietf-mimi-room-policy]. If all the changes are authorized, the ParticipantListData is modified accordingly.

A single commit is not valid if it contain any combination of Participant list updates that operate on (add, remove, or change the role of) the same user in the participant list more than once.

7.6. Room Metadata

The Room Metadata component contains data about a room which might be displayed as human-readable information for the room, such as the name of the room and a URL pointing to its room image/avatar.

It can contain a list of room_descriptions, each of which can have a specific language_tag and media_type along with the description_content. An empty media_type implies text/plain;charset=utf-8.

RoomMetaData is the format of the data field inside the ComponentData struct for the Room Metadata component in the app_data_dictionary GroupContext extension.

```
/* a valid URI (ex: MIMI URI) */
struct {
  opaque uri<V>;
} Uri;

/* a sequence of valid UTF8 without nulls */
struct {
  opaque string<V>;
} UTF8String;

struct {
  /* an empty media_type is equivalent to text/plain;charset=utf-8 */
  opaque media_type<V>;
  opaque language_tag<V>;
  opaque description_content<V>;
} RichDescription;

struct {
  Uri room_uri;
  UTF8String room_name;
  RichDescription room_descriptions<V>;
  /* an https URI resolving to an avatar image */
  Uri room_avatar;
  UTF8String room_subject;
  UTF8String room_mood;
} RoomMetaData;

RoomMetaData room_metadata;

RoomMetaData RoomMetaUpdate;
```

RoomMetaUpdate (which has the same format as RoomMetaData) is the format of the update field inside the AppDataUpdate struct in an AppDataUpdate Proposal for the Room Metadata component. If the contents of the update field are valid and if the proposer is authorized to generate such an update, the value of the update field completely replaces the value of the data field.

Only a single Room metadata update is valid per commit.

8. Consent

Most instant messaging systems have some notion of how a user consents to be added to a room, and how they manipulate this consent.

In the connection-oriented model, once two users are connected, either user can add the other to any number of rooms. In other systems (often with many large and/or public rooms), a user needs to consent individually to be added to a room.

The MIMI consent mechanism supports both models and allows them to coexist. It allows a user to request consent, grant consent, revoke consent, and cancel a request for consent. Each of these consent operations can indicate a specific room, or indicate any room.

A connection grant or revoke does not need to specify a room if a connection request did, or vice versa. A connection grant or revoke does not even need to follow a connection request.

For example, Alice could ask for consent to add Bob to a specific room. Bob could send a connection grant for Alice to add him to any room, or a connection revoke preventing Alice from adding him to any room. Similarly, Alice might have sent a connection request to add Bob for any room (as a connection request), which Bob ignored or did not see. Later, Bob wants to join a specific room administered by Alice. Bob sends a connection grant for the specific room for Alice and sends a Knock request to Alice asking to be added. Finally, Cathy could send a connection grant for Bob (even if Bob did not initiate a connection request to Cathy), and Alice could recognize Cathy on the system and send a connection revoke for her preemptively.

> *NOTE*: Many providers use additional factors to apply default consent within their service such as a user belonging to a specific workgroup or employer, participating in a related room (ex: WhatsApp "communities"), or presence of a user in the other user’s contact list. MIMI does not need to provide a way to replicate or describe these supplemental mechanisms, since they are strongly linked to specific provider policies.

Consent requests have sensitive privacy implications. The sender of a consent request should receive an acknowledgement that the request was received by the provider of the target user. For privacy reasons, the requestor should not know if the target user received or viewed the request. The original requestor will obviously find out if the target grants consent, but a consent revocation/rejection is typically not communicated to the revoked/rejected user (again for privacy reasons).

Consent operations are only sent directly between the acting provider (sending the request, grant, revoke, or cancel) and the target provider (the object of the consent). In other words, the two providers must have a direct peering relationship.

In our example, Alice requests consent from Bob for any room. Later, Bob sends a grants consent to Alice to add him to any room. At the same time as sending the consent request, Alice grants consent to Bob to add her to any room.

9. Security Considerations

TODO: Add MIMI threat model, and great expand this section.

The MIMI protocol incorporates several layers of security.

Individual protocol actions are protected against network attackers with mutually-authenticated TLS, where the TLS certificates authenticate the identities that the protocol actors assert at the application layer.

Messages and room state changes are protected end-to-end using MLS. The protection is "end-to-end" in the sense that messages sent within the group are confidentiality-protected against all servers involved in the delivery of those messages, and in the sense that the authenticity of room state changes is verified by the end clients involved in the room. The usage of MLS ensures that the servers facilitating the exchange cannot read messages in the room or falsify room state changes, even though they can read the room state change messages.

Each room has an authorization policy that dictates which protocol actors can perform which actions in the room. This policy is enforced by the hub server for the room. The actors for whom the policy is being evaluated authenticate their identities to the hub server using the MLS PublicMessage signed object format, together with the identity credentials presented in MLS. This design means that the hub is trusted to correctly enforce the room’s policy, but this cost is offset by the simplicity of not having multiple policy enforcement points.

9.1. Franking

TBD.

10. IANA Considerations

This document registers the following four MLS application components per Section 7.6 of [I-D.ietf-mls-extensions].

10.1. frank_aad app component

* Value: TBD1
* Name: frank_aad
* Where: AD
* Recommended: Y
* Reference: RFCXXXX

10.2. franking_signature_key app component

* Value: TBD2
* Name: franking_signature_key
* Where: GC
* Recommended: Y
* Reference: RFCXXXX

10.3. participant_list app component

* Value: TBD3
* Name: participant_list
* Where: GC
* Recommended: Y
* Reference: RFCXXXX

10.4. room_metadata app component

* Value: TBD4
* Name: room_metadata
* Where: GC
* Recommended: Y
* Reference: RFCXXXX

11. References

11.1. Normative References

[I-D.barnes-mimi-arch]
Barnes, R., "An Architecture for More Instant Messaging Interoperability (MIMI)", Work in Progress, Internet-Draft, draft-barnes-mimi-arch-03, 4 March 2024, <https://datatracker.ietf.org/doc/html/draft-barnes-mimi-arch-03>.

[I-D.barnes-mls-appsync]
Jo谷l, Barnes, R., Mahy, R., and M. Mularczyk, "A Safe Application Interface to Messaging Layer Security", Work in Progress, Internet-Draft, draft-barnes-mls-appsync-01, 12 December 2024, <https://datatracker.ietf.org/doc/html/draft-barnes-mls-appsync-01>.

[I-D.ietf-mimi-content]
Mahy, R., "More Instant Messaging Interoperability (MIMI) message content", Work in Progress, Internet-Draft, draft-ietf-mimi-content-06, 28 February 2025, <https://datatracker.ietf.org/doc/html/draft-ietf-mimi-content-06>.

[I-D.ietf-mimi-room-policy]
Mahy, R., "Room Policy for the More Instant Messaging Interoperability (MIMI) Protocol", Work in Progress, Internet-Draft, draft-ietf-mimi-room-policy-01, 2 March 2025, <https://datatracker.ietf.org/doc/html/draft-ietf-mimi-room-policy-01>.

[I-D.ietf-mls-extensions]
Robert, R., "The Messaging Layer Security (MLS) Extensions", Work in Progress, Internet-Draft, draft-ietf-mls-extensions-07, 3 July 2025, <https://datatracker.ietf.org/doc/html/draft-ietf-mls-extensions-07>.

[I-D.mahy-mls-ratchet-tree-options]
Mahy, R., "Ways to convey the Ratchet Tree in Messaging Layer Security", Work in Progress, Internet-Draft, draft-mahy-mls-ratchet-tree-options-02, 22 April 2025, <https://datatracker.ietf.org/doc/html/draft-mahy-mls-ratchet-tree-options-02>.

[I-D.mahy-mls-semiprivatemessage]
Mahy, R., "Semi-Private Messages in the Messaging Layer Security (MLS) Protocol", Work in Progress, Internet-Draft, draft-mahy-mls-semiprivatemessage-05, 21 April 2025, <https://datatracker.ietf.org/doc/html/draft-mahy-mls-semiprivatemessage-05>.

[OidcCore] Sakimura, N., Bradley, J., Jones, M. B., Medeiros, B. de., and C. Mortimore, "OpenID Connect Core 1.0 incorporating errata set 2", 15 December 2023, <https://openid.net/specs/openid-connect-core-1_0.html>.

[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/rfc/rfc2119>.

[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/rfc/rfc3986>.

[RFC5322] Resnick, P., Ed., "Internet Message Format", RFC 5322, DOI 10.17487/RFC5322, October 2008, <https://www.rfc-editor.org/rfc/rfc5322>.

[RFC6125] Saint-Andre, P. and J. Hodges, "Representation and Verification of Domain-Based Application Service Identity within Internet Public Key Infrastructure Using X.509 (PKIX) Certificates in the Context of Transport Layer Security (TLS)", RFC 6125, DOI 10.17487/RFC6125, March 2011, <https://www.rfc-editor.org/rfc/rfc6125>.

[RFC6350] Perreault, S., "vCard Format Specification", RFC 6350, DOI 10.17487/RFC6350, August 2011, <https://www.rfc-editor.org/rfc/rfc6350>.

[RFC7231] Fielding, R., Ed. and J. Reschke, Ed., "Hypertext Transfer Protocol (HTTP/1.1): Semantics and Content", RFC 7231, DOI 10.17487/RFC7231, June 2014, <https://www.rfc-editor.org/rfc/rfc7231>.

[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/rfc/rfc8174>.

[RFC8446] Rescorla, E., "The Transport Layer Security (TLS) Protocol Version 1.3", RFC 8446, DOI 10.17487/RFC8446, August 2018, <https://www.rfc-editor.org/rfc/rfc8446>.

[RFC9110] Fielding, R., Ed., Nottingham, M., Ed., and J. Reschke, Ed., "HTTP Semantics", STD 97, RFC 9110, DOI 10.17487/RFC9110, June 2022, <https://www.rfc-editor.org/rfc/rfc9110>.

[RFC9420] Barnes, R., Beurdouche, B., Robert, R., Millican, J., Omara, E., and K. Cohn-Gordon, "The Messaging Layer Security (MLS) Protocol", RFC 9420, DOI 10.17487/RFC9420, July 2023, <https://www.rfc-editor.org/rfc/rfc9420>.

[RFC9458] Thomson, M. and C. A. Wood, "Oblivious HTTP", RFC 9458, DOI 10.17487/RFC9458, January 2024, <https://www.rfc-editor.org/rfc/rfc9458>.

11.2. Informative References

[Grubbs2017]
Grubbs, P., Lu, J., and T. Ristenpart, "Message Franking via Committing Authenticated Encryption", 2017, <https://eprint.iacr.org/2017/664.pdf>.

[I-D.mahy-mimi-identity]
Mahy, R., "More Instant Messaging Interoperability (MIMI) Identity Concepts", Work in Progress, Internet-Draft, draft-mahy-mimi-identity-03, 3 March 2025, <https://datatracker.ietf.org/doc/html/draft-mahy-mimi-identity-03>.

[InvisibleSalamanders]
Dodis, Y., Grubbs, P., Ristenpart, T., and J. Woodage, "Fast Message Franking: From Invisible Salamanders to Encryptment", 2018, <https://link.springer.com/content/pdf/10.1007/978-3-319-96884-1_6.pdf>.

[RFC8555] Barnes, R., Hoffman-Andrews, J., McCarney, D., and J. Kasten, "Automatic Certificate Management Environment (ACME)", RFC 8555, DOI 10.17487/RFC8555, March 2019, <https://www.rfc-editor.org/rfc/rfc8555>.

[SecretConversations]
Facebook, "Messenger Secret Conversations: Technical Whitepaper, Version 2.0", 18 May 2017, <https://about.fb.com/wp-content/uploads/2016/07/messenger-secret-conversations-technical-whitepaper.pdf>.

Acknowledgments

Thanks to Paul Grubs, Jon Millican, and Julia Len for their reviews of the franking mechanism and suggested changes. Thanks to Felix Linker for his preliminary formal methods analysis (https://github.com/felixlinker/mimi-franking-tamarin/) of MIMI franking.

Authors' Addresses

Richard L. Barnes
Cisco
Email: rlb@ipv.sx

Matthew Hodgson
The Matrix.org Foundation C.I.C.
Email: matthew@matrix.org

Konrad Kohbrok
Phoenix R&D
Email: konrad.kohbrok@datashrine.de

Rohan Mahy
Rohan Mahy Consulting Services
Email: rohan.ietf@gmail.com

Travis Ralston
The Matrix.org Foundation C.I.C.
Email: travisr@matrix.org

Raphael Robert
Phoenix R&D
Email: ietf@raphaelrobert.com