

More Instant Messaging Interoperability
Internet-Draft
Intended status: Informational
Expires: 23 April 2026

R. Mahy
Rohan Mahy Consulting Services
20 October 2025

Room Policy for the More Instant Messaging Interoperability (MIMI)
Protocol
draft-ietf-mimi-room-policy-02

Abstract

This document describes a set of concrete room policies for the More Instant Messaging Interoperability (MIMI) Working Group. It describes several independent properties and policy attributes which can be combined to model a wide range of chat and multimedia conference types.

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://ietf-wg-mimi.github.io/mimi-room-policy/draft-ietf-mimi-room-policy.html>. Status information for this document may be found at <https://datatracker.ietf.org/doc/draft-ietf-mimi-room-policy/>.

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/ietf-wg-mimi/mimi-room-policy>.

Status of This Memo

This Internet-Draft is submitted in full conformance with the provisions of BCP 78 and BCP 79.

Internet-Drafts are working documents of the Internet Engineering Task Force (IETF). Note that other groups may also distribute working documents as Internet-Drafts. The list of current Internet-Drafts is at <https://datatracker.ietf.org/drafts/current/>.

Internet-Drafts are draft documents valid for a maximum of six months and may be updated, replaced, or obsoleted by other documents at any time. It is inappropriate to use Internet-Drafts as reference material or to cite them other than as "work in progress."

This Internet-Draft will expire on 23 April 2026.

Copyright Notice

Copyright (c) 2025 IETF Trust and the persons identified as the document authors. All rights reserved.

This document is subject to BCP 78 and the IETF Trust's Legal Provisions Relating to IETF Documents (<https://trustee.ietf.org/license-info>) in effect on the date of publication of this document. Please review these documents carefully, as they describe your rights and restrictions with respect to this document. Code Components extracted from this document must include Revised BSD License text as described in Section 4.e of the Trust Legal Provisions and are provided without warranty as described in the Revised BSD License.

Table of Contents

1. Introduction	3
2. Conventions and Definitions	4
2.1. Moderation Terms	4
2.2. Room Capabilities	5
3. Role-Based Access Control	5
4. Preauthorized Users	8
5. Base Room policy format syntax	10
5.1. Membership-related policy	10
5.2. Delivery and Read notifications, Pseudonyms	11
5.3. Link, Logging, History, and Bot policies	12
5.4. Link policies	13
5.5. Asset policies	14
5.6. Logging policies	14
5.7. Chat history policies	14
5.8. Chat bot policies	14
6. Operational policy	15
6.1. Some MLS-related policy that could be tied to a room	15
6.2. Not relevant to MIMI (between client and its provider)	18
6.3. Areas for future works	18
7. Role Capabilities	18
7.1. Membership Capabilities	19
7.2. Adjust metadata	22
7.3. Message Capabilities	22
7.4. Asset Capabilities	24
7.5. Real-time media	24

7.6. Disruptive Policy Changes	25
7.7. Reserved Capabilities	25
8. Extensibility of the policy format	26
9. Security Considerations	27
10. IANA Considerations	27
10.1. Preauthorized users MLS application component	27
10.2. Role definitions MLS application component	27
10.3. New MIMI Role Capabilities registry	27
11. References	29
11.1. Normative References	29
11.2. Informative References	30
Appendix A. Role examples	30
A.1. Cooperatively administered room	30
A.2. Strictly administered room	35
A.3. Moderated room	40
A.4. Multi-organization administered room	46
Appendix B. Complete TLS Presentation Language Syntax	53
Acknowledgments	56
Author's Address	56

1. Introduction

The MIMI architecture [I-D.ietf-mimi-arch] describes how each room has an associated policy. Providers offer a "policy envelope" of supported and allowed policy settings, from which the creator of a room selects a specific room policy. The room policy might further allow individual participants to make specific choices (for example, allowing but not requiring read-message notifications), while constraining other choices (for example, prohibiting self-deleting messages). Individual users can examine the room policy to determine if it is consistent with policies they accept either before or immediately on joining a room. Section 4.4 of [I-D.ietf-mimi-arch]

Making rooms interoperable across existing clients is challenging, as rooms and clients can support different policies and capabilities across vendors and providers. Our goal is to balance the policy and authorization goals of the room with the policy and authorization goals of the end user, so we can support a broad range of vendors and providers.

Each room is owned by one provider at a time. The owning provider controls the range of acceptable policies. The user responsible for the room can further choose among the acceptable policies. Users (regardless if on other providers) can either accept the policies of the room or not.

However we want to make it as easy as possible for clients from other providers to comply with the room policy primitives without enumerating specific features or requiring all clients implementations to present an identical user experience. An important aspect of this is the system of configurable Role-based access control with granular capabilities per role (described in Section 3). Each user in the participant list (defined in [I-D.ietf-mimi-protocol]) has exactly one role. By virtue of a user's credential, a user might also be `_preauthorized_` with a specific role as described in Section 4.

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.

***Room ID*:** An identifier which uniquely identifies a room.

***User ID*:** An internal identifier which uniquely identifies a user.

***Nickname*:** The identifier by which a user is referred inside a room. Depending on the context it may be a display name, handle, pseudonym, or temporary identifier. The nickname in one room need not correlate with the nickname for the same user in a different room.

***Client ID*:** An internal identifier which uniquely identifies one client/device instance of one user account.

***Persistent vs. Temporary rooms*:** A temporary room is no longer joinable once the last participant exits whereas a persistent room is not destroyed when the last participant exist. As MLS has no notion of a group with no members, a persistent room could consist of a sequence of distinct MLS groups, zero or one of which would exist at a time.

2.1. Moderation Terms

***Knock*:** To request entry into a room.

***Ban*:** To remove a user from a room such that the user is not allowed to re-enter the room (until and unless the ban has been removed). It is distinct from merely removing a user from a room.

***Kick*:** To temporarily remove a participant's clients from a room. The user is allowed to re-enter the room at any time.

Voice (noun): The privilege to send messages in a room.

Revoke Voice: To remove the permission to send messages in a room.

Grant Voice: To grant the permission to send messages in a room.

2.2. Room Capabilities

Membership-Approach: The overall approach of membership authorization in a room, which could be open, members-only (administrated), fixed-membership, or parent-dependent.

* *Open room*: Typically an open room can be joined by any non-banned user. It can be represented solely by a permissive set of roles as defined in Section 3.

* *Members-Only room*: A members-only room can only be joined by a user in the participant list, or who is pre-authorized. Authorized users can add or remove users to the room. In an enterprise context, it is also common (but not required) for users from a particular domain, group, or workgroup to be pre-authorized to add themselves to a Members-Only room. It can be represented solely by a set of less permissive roles as defined in Section 3.

* *Fixed-Membership room*: Fixed-membership rooms have the list of participants specified when they are created. Other users cannot be added. Ordinary users cannot leave or be removed, however a user can remove all its clients from the associated MLS group. The most common case of a fixed-membership room is a 1:1 conversation. This room membership style is used to implement Direct Message (DM) and Group DM features. Only a single fixed-membership room can exist for any unique set of participants.

* *Parent-dependent room*: In a parent-dependent room, the list participants of the room must be a strict subset of the participants of the parent room. If a user leaves or is removed from the parent room, that user is automatically removed from any parent-dependent rooms of that parent.

3. Role-Based Access Control

The Role-Based Access Control component contains a list of all the roles in the room, and the capabilities associated with them. It contains a `role_index`, which is used to refer to the role elsewhere. (Note that role indexes might not be contiguous.) The `role_index` zero is reserved to refer to a participant that does not (yet) or no longer appears (or will no longer appear) in the participant list.

The component also contains a `role_name` (a human-readable text string name for the role), and a `role_description` (another string, which can have zero length).

Each Role also can contain constraints on the minimum and maximum number of participants, and the minimum and maximum number of active participants. If the minimum number is zero, there is no minimum number of participants for that particular role. If there is no maximum number of participants for a particular role, that parameter is absent.

If the maximum number of active participants is zero, then no participants are allowed to have clients in the room's MLS group.

The `authorized_role_changes` field is used to provide fine-grained control about which transitions are allowed when adding and removing participants and when moving participants to new roles, including banning/unbanning, and promoting/demoting to or from roles with moderator or administrator privileges. A more detailed discussion is in the description of the specific capabilities in the next section.

This design results in each participant only having a single role at a time, with a single list of capabilities and an explicit list of allowed role transitions. It makes the authorization process for a verifier consistent regardless of the complexity of the set of authorization rules.

Some examples are provided in Appendix A.

RoleData is the format of the data field inside the ComponentData struct for the Role-Based Access Control component in the `app_data_dictionary` GroupContext extension defined in [I-D.ietf-mls-extensions].

```
/* See MIMI Capability Types IANA registry */
uint16 CapablityType;

struct {
    uint32 from_role_index;
    uint32 target_role_indexes<V>;
} SingleSourceRoleChangeTargets;

struct {
    uint32 role_index;
    opaque role_name<V>;
    opaque role_description<V>;
    CapabilityType role_capabilities<V>;
    uint32 minimum_participants_constraint;
    optional uint32 maximum_participants_constraint;
    uint32 minimum_active_participants_constraint;
    optional uint32 maximum_active_participants_constraint;
    SingleSourceRoleChangeTargets authorized_role_changes<V>;
} Role;

struct {
    Role roles<V>;
} RoleData;

RoleData RoleUpdate;
```

RoleUpdate (which has the same format as RoleData) is the format of the update field inside the AppDataUpdate struct in an AppDataUpdate Proposal for the Role-Based Access Control 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.

Note that in the MIMI environment, changing the definitions of roles is anticipated to be very rare over the lifetime of a room (for example changing a room which has grown dramatically from cooperatively managed by all participants to explicitly moderated or administered).

Changing Role definitions is sufficiently disruptive, that an update to this component is not valid if it appear in the same commit as any Participant List change.

4. Preauthorized Users

Preauthorized users are MIMI users and external senders that have authorization to adopt a role in a room by virtue of certain credential claims or properties, as opposed to being individually enumerated in the participant list. For example, a room for employee benefits might be available to join with the regular participant role to all full-time employees with a residence in a specific country; while anyone working in the human resources department might be able to join the same room as a moderator. This data structure is consulted in two situations: for external joins (external commits) and external proposals when the requester does not already appear in the participant list; and separately when an existing participant explicitly tries to change its `_own_` role.

Only consulting Preauthorized users in these cases prevents several attacks. For example, it prevents an explicitly banned user from rejoining a group based on a preauthorization.

`PreAuthData` is the format of the data field inside the `ComponentData` struct for the Preauthorized Participants component in the `application_data` `GroupContext` extension.

The individual `PreAuthRoleEntry` rules in `PreAuthData` are consulted one at a time. A `PreAuthRoleEntry` matches for a requester when every `Claim.claim_id` has a corresponding claim in the requester's `MLS Credential` which exactly matches the corresponding `claim_value`. When the rules in a Preauthorized users struct match multiple roles, the requesting client receives the first role which matches its claims.


```
struct {
    /* MLS Credential Type of the "claim" */
    CredentialType credential_type;
    /* the binary representation of an X.509 OID, a JWT claim name */
    /* string, or the CBOR map claim key in a CWT (an int or tstr) */
    opaque id<V>;
} ClaimId;

struct {
    ClaimId claim_id;
    opaque claim_value<V>;
} Claim;

struct {
    /* when all claims in the claimset are satisfied, the claimset */
    /* is satisfied */
    Claim claimset<V>;
    Role target_role;
} PreAuthRoleEntry;

struct {
    PreAuthRoleEntry preauthorized_entries<V>;
} PreAuthData;
```

PreAuthData PreAuthUpdate;

PreAuthUpdate (which has the same format as PreAuthData) is the format of the update field inside the AppDataUpdate struct in an AppDataUpdate Proposal for the Preauthorized Participants 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.

As with the definition of roles, in MIMI it is not expected that the definition of Preauthorized users would change frequently. Instead the claims in the underlying credentials would be modified without modifying the preauthorization policy.

Changing Preauthorized user definitions is sufficiently disruptive, that an update to this component is not valid if it appears in the same commit as any Participant List change, except for user removals.

Because the Preauthorized users component usually authorizes non-members, it is also a natural choice for providing concrete authorization for policy enforcing systems incorporated into or which run in coordination with the MIMI Hub provider or specific MLS Distribution Services. For example, a preauthorized role could allow the Hub to remove participants and to ban them, but not to add any users or devices. This unifies the authorization model for members and non-members.

5. Base Room policy format syntax

5.1. Membership-related policy

TODO: refactor `membership_style` to be constraints to the role-based access control system

The `membership_style` of a room can express some additional constraints on membership transitions in a room. It can have one of the following values:

- * ordinary (default)
- * fixed-membership
- * parent-dependent

```
enum {  
    reserved(0)  
    ordinary(1),  
    fixed-membership(2),  
    parent-dependent(3),  
    (255)  
} MembershipStyle;
```

An ordinary room has no constraints beyond those of the role-based access control system. A fixed-membership room (which can be used for DMs or Group DMs) has a participant list set once at creation time that cannot be added to. A parent-dependent room always has a strict subset of the participants of its parent room. A parent-dependent room is always hosted on the same Hub as the parent room.

If the `membership_style` is parent-dependent the `parent_room_uri` MUST be set with the room ID of the parent. Otherwise the field is zero-length.

If `multi_device` is true (the default), the MLS group may contain multiple clients per user. If false only a single client can be an MLS member at one time.

```
enum {
    false(0),
    true(1)
} bool;

struct {
    MembershipStyle membership_style;
    Uri parent_room_uri<V>;
    bool multi_device;
    bool persistent_room;
    ...
} BaseRoomPolicy;
```

If `persistent_room` is false, the room will be automatically inaccessible when the corresponding MLS group is destroyed (when there are no clients in the group). If `persistent_room` is true, the room policy will remain and a client whose user has appropriate authorization can create a new MLS group for the same room.

5.2. Delivery and Read notifications, Pseudonyms

```
enum {
    optional(0),
    required(1),
    forbidden(2)
} Optionality;

struct {
    ...
    Optionality delivery_notifications;
    Optionality read_receipts;
    bool pseudonymous_ids;
    ...
} RoomPolicy;
```

The `delivery_notifications` value can be set to "forbidden", "optional", or "required". If the value is set to "optional", the client uses its local configuration to determine if it should send delivery notifications in the group.

The `read_receipts` value can be set to "forbidden", "optional", or "required". If the value is set to "optional", the client uses its local configuration to determine if it should send read receipts in the group.

The format for delivery notifications and read receipts is described in Section 5.12 of [I-D.ietf-mimi-content].

If `pseudonymous_ids` is true, clients in the MLS group are free to use pseudonymous identifiers in their MLS credentials. Otherwise the policy of the room is that "real" long-term identifiers are required in MLS credentials in the room's corresponding MLS group.

5.3. Link, Logging, History, and Bot policies

```
struct {
    bool on_request;
    Uri join_link;
    bool multiuser;
    uint32 expiration;
    Uri link_requests;
} LinkPolicy;

enum {
    direct(0),
    hubProxy(1),
    ohttp(2),
    (255)
} DownloadPrivacyType;

enum {
    unspecified(0),
    localProvider(1),
    hub(2),
    (255)
} AssetUploadLocation;

struct {
    DownloadPrivacyType allowed_download_types<V>;
    DownloadPrivacyType forbidden_download_types<V>;
    DownloadPrivacyType default_download_type;
} DownloadPrivacy;

struct {
    AssetUploadLocation asset_upload_location;
    opaque upload_domain<V>;
    DownloadPrivacy download_privacy;
    uint64 max_image;
    uint64 max_audio;
    uint64 max_video;
    uint64 max_attachment;
} AssetPolicy;

struct {
    Optionality logging;
    Uri logging_clients<V>;
```

```
    Uri machine_readable_policy;
    Uri human_readable_policy;
} LoggingPolicy;

struct {
    Optionality history_sharing;
    Role who_can_share<V>;
    bool automatically_share;
    uint32 max_time_period;
} HistoryPolicy;

struct {
    opaque name<V>;
    opaque description<V>;
    Uri homepage;
    Role bot_role;
    bool can_read;
    bool can_write;
    bool can_target_message_in_group;
    bool per_user_content;
} Bot;

struct {
    ...
    bool discoverable;
    LinkPolicy link_policy;
    AssetPolicy asset_policy;
    LoggingPolicy logging_policy;
    HistoryPolicy history_sharing;
    Bot allowed_bots<V>;
    ...
} RoomPolicy;
```

5.4. Link policies

If discoverable is true, the room is searchable. Presumably this means the the only way to join the room in a client user interface is to be added by an administrator or to use a joining link. Inside the LinkPolicy are several fields that describe the behavior of links. If the on_request field is true, no joining link will be provided in the room policy; the client will need to fetch a joining link out-of-band or generate a valid one for itself. If present, the URI in link_requests can be used by the client to request an invite code. The value of join_link is empty and the other fields are ignored. If the on_request field is false, the join_link field will contain a joining link. If the link will work for multiple users, multiuser is true. The expiration field represents the time, in seconds after the start of the UNIX epoch (1-January-1970) when the link will expire.

The `link_requests` field can be empty.

5.5. Asset policies

Assets refer to attached files, images, audio files, and video files.

5.6. Logging policies

Inside the `LoggingPolicy`, the `logging` field can be forbidden, optional, or required. If logging is forbidden then the other fields are empty. If logging is required, the list of `logging_clients` needs to contain at least one logging URI. Each provider should have no more than one logging client at a time in a room. The `machine_readable_policy` and `human_readable_policy` fields optionally contain pointers to the owning provider's machine readable and human readable logging policies, respectively. If logging is optional and there is at least one `logging_client` then logging is active for the room.

5.7. Chat history policies

Inside the `HistoryPolicy`, if `history_sharing` is forbidden, this means that clients (including bots) are expected to not to share chat history with new joiners, in which case `who_can_share` is empty, `automatically_share` is false, and `max_time_period` is zero. Otherwise `who_can_share` is a list of roles that are authorized to share history (for example, only admins and owners can share). The values of none and outcast cannot be used in `who_can_share`. If `automatically_share` is true, clients can share history with new joiners without user initiation. The history that is shared is limited to `max_time_period` seconds worth of history.

5.8. Chat bot policies

Inside the `RoomPolicy` there is a list of `allowed_bots`. Each of which has several fields. The name, description, and homepage are merely descriptive. The `bot_role` indicates if the chat bot would be treated as a system-user, owner, admin, regular_user, or visitor. The `can_read` and `can_write` fields indicate if the chat bot is allowed to read messages or send messages in the MLS group, respectively. If `can_target_message_in_group` is true it indicates that the chat bot can send an MLS targeted message (see Section 2.2 of [I-D.ietf-mls-extensions]) or use a different conversation or out-of-band channel to send a message to specific individual users in the room. If `per_user_content` is true, the chat bot is allowed to send messages with distinct content to each member. (For example a poker bot could deal a different hand to each user in a chat). Users could set policies to reject or leave groups with bots rights that are

inconsistent with the user's privacy goals.

6. Operational policy

Section 7 of the [RFC9750] defines a set of operational policy considerations that influence interoperability of MLS clients. MIMI explicitly address a handful of the issues in the document by taking a position on ordering (Proposals referenced in a Commit need to be received before the Commit; the Commit entering a new epoch needs to be received before any other messages in that epoch), privacy of handshake messages (handshakes can be a PublicMessage or SemiPrivateMessage), and GroupInfo storage (committees need to provide a valid GroupInfo to the Hub). The rest of these issues are described here. Just because a topic is listed does not mean that a room needs to take a position; nor different rooms on a Hub need to have different policies for these items.

6.1. Some MLS-related policy that could be tied to a room

- * any mandatory or forbidden MLS extensions.
- * which proposals are valid to have in a commit, including but not limited to:
 - when, and under what circumstances, a reinitialization proposal is allowed.
 - when proposals from external senders are allowed and how to authorize those proposals.
 - when external joiners are allowed and how to authorize those external commits.
 - which other proposal types are allowed.
- * when members should commit pending proposals in a group.
- * when two credentials represent the same client.
- * how long to allow a member to stay in a group without updating its leaf keys before removing them.
- * When and how to pad messages.
- * When to send a reinitialization proposal.
- * How often clients should update their leaf keys.

- * Whether to prefer sending full commits or partial/empty commits.
- * Whether there should be a `required_capabilities` extension in groups.
- * minimum and maximum lifetime of `KeyPackages`
- * if last resort `KeyPackages` are allowed
- * how long to store resumption PSK (how much time and how many epochs)
- * minimum and maximum number past epochs to keep
- * how long to keep unused nonce and key pairs for a sender
- * maximum number of unused key pairs to keep
- * maximum number of steps that clients will move a secret tree ratchet forward in response to a single message before rejecting it
- * tolerance to out of order app messages
- * tolerance to out of order handshake messages
- * handshakes may be which of `PublicMessage`, `PrivateMessage`, or `SemiPrivateMessage`.
- * if external joiners are allowed
- * if external proposals are allowed
 - if so, who can submit
 - which member(s) are responsible for submitting pending proposals
- * how a joiner gets access to the `ratchet_tree`

The structure below describes provides a way to describe many of these parameters.


```
struct {
    ProtocolVersion versions<V>;
    CipherSuite cipher_suites<V>;
    ExtensionType extensions<V>;
    ProposalType proposals<V>;
    CredentialType credentials<V>;
    WireFormats wire_formats<V>;
    ComponentID component_ids<V>;
    ComponentID safe_aad_types<V>;
    MediaType media_types<V>;
    ContentType content_types<V>;
} ExtendedCapabilities;

enum {
    unspecified(0),
    immediateCommit(1),
    randomDelay(2),
    preferenceWheel(3),
    designatedCommitter(4),
    (255)
} PendingProposalStrategy;

struct {
    PendingProposalStrategy pending_proposal_strategy;
    uint64 minimumDelayMs;
    uint64 maximumDelayMs;
} PendingProposalPolicy;

struct {
    uint64 minimumTime;
    uint64 defaultTime;
    uint64 maximumTime;
} MinDefaultMaxTime;

struct {
    uint8 epoch_tolerance;
    uint16 pad_to_size;
    uint32 max_generations_skipahead;
} AppMessagePolicy;

struct {
    ExtendedCapabilities mandatory_capabilities;
    ExtendedCapabilities default_capabilities;
    ExtendedCapabilities forbidden_capabilities;
    WireFormats handshake_formats<V>;
    bool external_proposal_allowed;
    bool external_commit_allowed;
```

```
PendingProposalPolicy pending_proposal_policy;
MinDefaultMaxTime LeafNode_update_time;
AppMessagePolicy app_message_policy;
uint64 max_kp_lifetime;
uint64 max_credential_lifetime;
uint64 resumption_psk_lifetime;
MinDefaultMaxTime sender_nonce_keypair_lifetime;
uint32 max_keypairs;
MinDefaultMaxTime buffer_incoming_message_time;
uint32 max_buffered_messages;
} OperationalParameters;
```

6.2. Not relevant to MIMI (between client and its provider)

- * how many KPs to keep active
- * how group IDs are constructed

6.3. Areas for future works

How to protect and share the GroupInfo objects needed for external joins.

If an application wishes to detect and possibly discipline members that send malformed commits with the intention of corrupting a group's state, there must be a method for reporting and validating malformed commits. MLS requires the following parameters to be defined, which must be the same for two implementations to interoperate:

Application-level identifiers of public key material (specifically the application_id extension as defined in Section 5.3.3 of [RFC9420]).

7. Role Capabilities

As described in Section 3, each role has a list of capabilities, which in rare cases could be empty. When we say that the holder of a capability can take some action, we mean that whatever entity is taking the action (a participant, a potential future participant, or an external party) has a specific entry in the Participant List struct and a corresponding role--or is preauthorized to take action with a specific role via the Preauthorized Users struct--and that the role_capabilities list contains the relevant capability.

Unless otherwise specified, capabilities apply both to sending a set of consistent MLS proposals that could be committed by any member of the corresponding MLS group, and to sending an MLS commit containing a set of consistent MLS proposals.

7.1. Membership Capabilities

The membership capabilities below allow authorized holders to update the Participant list, or change the active participants (by removing and adding MLS clients corresponding to those participants), or both.

- * `canAddParticipant` - the holder of this capability can add another user, that is not already in the participant list, to the participant list. (This capability does not apply to the holder adding itself.) The `authorized_role_changes` list in the holder's role is consulted to authorize the added user's target role. The `authorized_role_changes` list MUST have an entry where the `authorized_role_changes.from_role_index` equals zero, and that entry's `target_role_indexes` list includes the target role. The proposed action is only authorized if the action respects both the `maximum_participants_constraint` (if present) and `maximum_active_participants_constraint` (if present) for the added user's target role. When the participant list addition for the target role is authorized, the holder is also authorized to add any MLS clients matching the added user to the room's MLS group .
- * `canAddOwnClient` - a holder of this capability that is in the participant list, can add its own client (via an external commit or external proposal); and can add other clients that share the same user identity (via Add proposals) if the holder's client is already a member of the corresponding MLS group.
- * `canAddSelf` - the holder of this capability can use an external commit or external proposal to add itself to the participant list. (The holder MUST NOT already appear in the participant list). Its usage differs slightly based on in which role it appears.
 - When `canAddSelf` appears on role zero, any user who is not already in the participant list can add itself, with certain provisions. The holder consults the `authorized_role_changes` list for an entry with `from_role_index` equal to zero. The holder can add itself with any non-zero `target_role_indexes` from that entry, if the action respects both the `maximum_participants_constraint` (if present) and `maximum_active_participants_constraint` (if present) for the added user's target role.

- When `canAddSelf` appears on a non-zero role, a client can only become the holder of this capability via the Preauthorized users mechanism. The `authorized_role_changes` list in the target role MUST have an entry where the `from_role_index` is zero and the `target_role_indexes` contains the target role. In addition, the action MUST respect both the `maximum_participants_constraint` (if present) and `maximum_active_participants_constraint` (if present) for the added user's target role.
- * `canUseJoinCode` - the holder of this capability can externally join a room using a join code for that room, provided the join code is valid, the join code refers to a valid target role, and both the `maximum_participants_constraint` (if present) and `maximum_active_participants_constraint` (if present) constraints are respected.
- * `canRemoveParticipant` - the holder of this capability can propose a) the removal of another user (excluding itself) from the participant list, and b) removal of all of that user's clients, as a single action. There MUST NOT be any clients of the removed user in the MLS group after the corresponding commit. A proposer holding this capability consults its role's `authorized_role_changes` entries for an entry where `from_role_index` matches the target user's current role; if the `target_role_indexes` for that entry contains zero, and the `minimum_participants_constraint` and `minimum_active_participants_constraint` are satisfied, the proposal is authorized.
- * `canRemoveOwnClient` - the holder of this capability can propose to remove its own client using an MLS Remove or SelfRemove proposal without changing the Participant list. Due to restrictions in MLS which insure the consistency of the group, this action cannot be committed by the leaving user. If the `minimum_active_participants_constraint` is satisfied, the proposal is authorized.

- * `canRemoveSelf` - the holder of this capability can propose to remove itself from (i.e. leave) the participant list; it MUST simultaneously propose to remove all of its remaining clients from the corresponding MLS group. Due to restrictions in MLS which insure the consistency of the group, this action cannot be committed by the leaving user. A proposer holding this capability consults its role's `authorized_role_changes` entries for an entry where `from_role_index` matches its current role; if the `target_role_indexes` for that entry contains zero, and the `minimum_participants_constraint` and `minimum_active_participants_constraint` are satisfied, the proposal is authorized.
- * `canKick` - the holder of this capability can propose removal of another participant's clients, without changing the Participant List. If the `minimum_active_participants_constraint` is satisfied, the proposal is authorized.
- * `canChangeUserRole` - the holder of this capability is authorized to change the role of another participant (but not itself), according to the holder's `authorized_role_changes` list, from a role represented by an entry where the target's current role matches `from_role_index` to any of the non-zero `target_role_indexes` in the same element of `authorized_role_changes`. The `minimum_participants_constraint` and `minimum_active_participants_constraint` for the target user's current role, and the `maximum_participants_constraint` (if present) and `maximum_active_participants_constraint` (if present) for the target user's target role must also be satisfied.
- * `canChangeOwnRole` - the holder of this capability is authorized to change its own role to the first non-zero role it matches in the Preauthorized users component (see Section 4). The `authorized_role_changes` list is not consulted. The `minimum_participants_constraint` and `minimum_active_participants_constraint` for the holder's original role, and the `maximum_participants_constraint` (if present) and `maximum_active_participants_constraint` (if present) for the holder's target role must also be satisfied.
- * `canBan` - the holder of this capability can propose to "ban" another user. Specifically, a successful ban changes the target user's role to a special "banned" role (if it exists), and removes all the banned user's clients. The "banned" role always has `role_index = 1` and `role_name = "banned"` (without quotes).

A "banned" role does not have to exist in a room, but to use the canBan and canUnban capabilities, the role needs to exist exactly as described above. While holding canChangeUserRole and canKick capabilities would allow the same action, it could potentially allow the holder other actions which might be undesirable in some contexts, such as kicking clients without banning.

A proposer holding this capability consults its role's `authorized_role_changes` entries for an entry where `from_role_index` matches the target user's current role; if the `target_role_indexes` for that entry contains the `role_index` 1; that `role_name` = "banned" for the role with `role_index` = 1, and the `minimum_participants_constraint` and `minimum_active_participants_constraint` are satisfied, the proposal is authorized.

- * `canUnban` - the holder of this capability can propose to "unban" another user. Specifically, a successful unban changes the target user's role from `role_index` = 1 to another non-zero `role_index` allowed by the holder's `authorized_role_changes` list. Adding clients for that unbanned user is `_not_` authorized by this capability. The authorization of this capability is identical to the `canChangeUserRole` capability, except that the `from_role_index` for the unbanned user MUST be 1, and the `role_name` of role 1 MUST be "banned".

7.2. Adjust metadata

The holder of each of the following capabilities is authorized to update the Room metadata defined in [I-D.ietf-mimi-protocol], changing the relevant field:

- * `canChangeRoomName`
- * `canChangeRoomDescription`
- * `canChangeRoomAvatar`
- * `canChangeRoomSubject`
- * `canChangeRoomMood`

7.3. Message Capabilities

The capabilities below refer to functionality related to the instant messages, for example sent using the MIMI content format [I-D.ietf-mimi-content].

- * `canSendMessage` - the holder can send instant messages to the room. Setting specific message fields may require additional capabilities.
- * `canReceiveMessage` - the holder can receive instant messages from the room.
- * `canCopyMessage` - the holder can copy content from a received instant message.
- * `canReportAbuse` - the holder can report a franked instant message as abusive.
- * `canReplyToMessage` - the holder can send a message replying to another message.
- * `canReactToMessage` - the holder can send a reaction, replying to another message, and using the "reaction" disposition.
- * `canDeleteOwnReaction` - the holder can retract (unlike) its own previous reaction.
- * `canDeleteOtherReaction` - the holder can delete the reaction of another user's previous reaction
- * `canEditOwnMessage` - the holder can edit the content of one of its own previously sent messages
- * `canDeleteOwnMessage` - the holder can retract one of its own previously sent messages
- * `canDeleteOtherMessage` - the holder can retract messages for other users.
- * `canStartTopic` - the holder can set the topic for a message
- * `canReplyInTopic` - the holder can send a message replying to a previous message, using the same topic as the original sender.
- * `canEditOwnTopic` - the holder can change the topic of a previously sent message
- * `canEditOtherTopic` - the holder can change the topic of a message previously sent by another user.
- * `canSendLink` - the holder can send an inline link

- * `canSendLinkPreview` - the holder can send an inline link with an associated preview.
- * `canFollowLink` - the holder can open a sent inline link.
- * `canCopyLink` - the holder can copy the URL of a sent inline link.

The Hub can enforce whether a member can send a message. It can also withhold fanout of application messages to clients of a user. The other capabilities in this section can only be enforced by other clients.

7.4. Asset Capabilities

- * `canUploadAttachment` - the holder can upload a file with the "attachent" disposition.
- * `canDownloadAttachment` - the holder can download a file with the "attachent" disposition.
- * `canUploadImage` - the holder can upload a file with the media type of "image" and the disposition of "render"
- * `canDownloadImage` - the holder can download a file with the media type of "image" and the disposition of "render"
- * `canUploadVideo` - the holder can upload a file with the media type of "video" and the disposition of "render"
- * `canDownloadVideo` - the holder can download a file with the media type of "video" and the disposition of "render"
- * `canUploadSound` - the holder can upload a file with the media type of "audio" and the disposition of "render"
- * `canDownloadSound` - the holder can download a file with the media type of "audio" and the disposition of "render"

7.5. Real-time media

The MIMI Working has not yet defined requirements for real-time media, however the capabilities below are widely representative of the permissions that would be required.

- * `canStartCall` - the holder can initiate a new real-time call/conference

- * `canJoinCall` - the holder can join an existing real-time call/conference
- * `canSendAudio` - the holder is authorized to contribute audio in a call/conference.
- * `canReceiveAudio` - the holder is authorized to receive audio in a call/conference.
- * `canSendVideo` - the holder is authorized to contribute video in a call/conference.
- * `canReceiveVideo` - the holder is authorized to receive video in a call/conference.
- * `canShareScreen` - the holder is authorized to contribute screen sharing in a call/conference
- * `canViewSharedScreen` - the holder is authorized to receive screen sharing in a call/conference

7.6. Disruptive Policy Changes

- * `canChangeRoomMembershipStyle` - the holder is authorized to modify the base room membership style.
- * `canChangeRoleDefinitions` - the holder is authorized to make changes to the definitions of the Roles component.
- * `canChangePreauthorizedUserList` - the holder is authorized to make changes to the Preauthorized Users component.
- * `canDestroyRoom` - the holder is authorized to completely destroy the room.
- * `canReinitGroup` - the holder is authorized to send an MLS ReInit proposal.

7.7. Reserved Capabilities

The following capability names are reserved for possible future use

- * `canCreateJoinCode`
- * `canKnock`
- * `canAcceptKnock`

- * canCreateSubgroup
- * canSendDirectMessage
- * canTargetMessage
- * canChangeOwnName
- * canChangeOwnPresence
- * canChangeOwnMood
- * canChangeOwnAvatar
- * canCreateRoom
- * canChangeMlsOperationalPolicies
- * canChangeOtherPolicyAttribute
- * MLS specific
 - update - update policy
 - PSK - psk policy
 - external proposal - general operational policy rules
 - external commit - general operational policy rules

8. Extensibility of the policy format

Finally, The extensibility mechanism allows for future addition of new room policies.

```
enum {
    null(0),
    boolean(1),
    number(2),
    string(3),
    jsonObject(4)
} ExtType;

struct {
    opaque name<V>;
    ExtType type;
    opaque value<V>;
} PolicyExtension;

struct {
    ...
    PolicyExtension policy_extensions<V>;
} RoomPolicy;
```

9. Security Considerations

This entire document focuses on authorization policy. TODO More Security

10. IANA Considerations

10.1. Preauthorized users MLS application component

TBC

10.2. Role definitions MLS application component

TBC

10.3. New MIMI Role Capabilities registry

Create a new registry with the following values assigned sequentially using the reference RFCXXXX.

```
canAddParticipant
canRemoveParticipant
canAddOwnClient
canRemoveSelf
canAddSelf
canCreateJoinCode - reserved for future use
canUseJoinCode
canBan
canUnBan
```

canKick
canKnock
canAcceptKnock
canChangeUserRole
canChangeOwnRole
canCreateSubgroup
canSendMessage
canReceiveMessage
canCopyMessage
canReportAbuse
canReactToMessage
canEditReaction
canDeleteReaction
canEditOwnMessage
canDeleteOwnMessage
canDeleteAnyMessage
canStartTopic
canReplyInTopic
canEditTopic
canSendDirectMessage
canTargetMessage
canUploadImage
canUploadVideo
canUploadAttachment
canDownloadImage
canDownloadVideo
canDownloadAttachment
canSendLink
canSendLinkPreview
canFollowLink
canCopyLink
canChangeRoomName
canChangeRoomDescription
canChangeRoomAvatar
canChangeRoomSubject
canChangeRoomMood
canChangeOwnName
canChangeOwnPresence
canChangeOwnMood
canChangeOwnAvatar
canStartCall
canJoinCall
canSendAudio
canReceiveAudio
canSendVideo
canReceiveVideo
canShareScreen
canViewSharedScreen

canChangeRoomMembershipStyle
canChangeRoleDefinitions
canChangePreauthorizedUserList
canChangeMlsOperationalPolicies
canDestroyRoom
canSendMLSReinitProposal

11. References

11.1. Normative References

[I-D.ietf-mimi-arch]

Barnes, R., "An Architecture for More Instant Messaging Interoperability (MIMI)", Work in Progress, Internet-Draft, draft-ietf-mimi-arch-01, 21 November 2024, <<https://datatracker.ietf.org/doc/html/draft-ietf-mimi-arch-01>>.

[I-D.ietf-mimi-content]

Mahy, R., "More Instant Messaging Interoperability (MIMI) message content", Work in Progress, Internet-Draft, draft-ietf-mimi-content-07, 7 July 2025, <<https://datatracker.ietf.org/doc/html/draft-ietf-mimi-content-07>>.

[I-D.ietf-mimi-protocol]

Barnes, R., Hodgson, M., Kohbrok, K., Mahy, R., Ralston, T., and R. Robert, "More Instant Messaging Interoperability (MIMI) using HTTPS and MLS", Work in Progress, Internet-Draft, draft-ietf-mimi-protocol-04, 7 July 2025, <<https://datatracker.ietf.org/doc/html/draft-ietf-mimi-protocol-04>>.

[I-D.ietf-mls-extensions]

Robert, R., "The Messaging Layer Security (MLS) Extensions", Work in Progress, Internet-Draft, draft-ietf-mls-extensions-08, 21 July 2025, <<https://datatracker.ietf.org/doc/html/draft-ietf-mls-extensions-08>>.

[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>>.

[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>>.

11.2. Informative References

- [RFC9750] Beurdouche, B., Rescorla, E., Omara, E., Inguva, S., and A. Duric, "The Messaging Layer Security (MLS) Architecture", RFC 9750, DOI 10.17487/RFC9750, April 2025, <<https://www.rfc-editor.org/rfc/rfc9750>>.

Appendix A. Role examples

A.1. Cooperatively administered room

This is an example set of role policies, which is suitable for friends and family rooms and small groups of peers in a workgroup or club.

- * no_role
 - role_index = 0
 - no capabilities
 - constraints
 - o minimum_participants_constraint = 0
 - o maximum_participants_constraint = null
 - o minimum_active_participants_constraint = 0
 - o maximum_active_participants_constraint = 0
 - o authorized_role_changes = []
- * banned
 - role_index = 1
 - no capabilities
 - constraints
 - o minimum_participants_constraint = 0
 - o maximum_participants_constraint = null
 - o minimum_active_participants_constraint = 0
 - o maximum_active_participants_constraint = 0

```
    o authorized_role_changes = []
* ordinary_user
- role_index = 2
- authorized capabilities
  o canAddParticipant
  o canRemoveParticipant
  o canAddOwnClient
  o canRemoveOwnClient
  o canRemoveSelf
  o canSendMessage
  o canReceiveMessage
  o canCopyMessage
  o canReportAbuse
  o canReplyToMessage
  o canReactToMessage
  o canDeleteOwnReaction
  o canEditOwnMessage
  o canDeleteOwnMessage
  o canStartTopic
  o canReplyInTopic
  o canEditOwnTopic
  o canUploadImage
  o canUploadVideo
  o canUploadSound
```

- o canUploadAttachment
- o canDownloadImage
- o canDownloadVideo
- o canDownloadSound
- o canDownloadAttachment
- o canSendLink
- o canSendLinkPreview
- o canFollowLink
- o canCopyLink
- o canChangeRoomName
- o canChangeRoomAvatar
- o canChangeRoomSubject
- o canChangeRoomMood
- o canChangeOwnName
- o canChangeOwnPresence
- o canChangeOwnMood
- o canChangeOwnAvatar
- constraints
 - o minimum_participants_constraint = 0
 - o maximum_participants_constraint = null
 - o minimum_active_participants_constraint = 0
 - o maximum_active_participants_constraint = null
 - o authorized_role_changes = [(0,[2]), (2,[0])]
- * group_admin

- role_index = 3
- authorized capabilities
 - o (include all the capabilities authorized for an ordinary_user)
 - o canBan
 - o canUnBan
 - o canKick
 - o canRevokeVoice
 - o canGrantVoice
 - o canChangeUserRole
 - o canDeleteOtherMessage
 - o canEditOtherTopic
 - o canChangeRoomDescription
- constraints
 - o minimum_participants_constraint = 1
 - o maximum_participants_constraint = null
 - o minimum_active_participants_constraint = 0
 - o maximum_active_participants_constraint = null
 - o authorized_role_changes = [(0,[1,2,3]), (1,[0,2,3]), (2,[0,1,3]), (3,[0,1,2])]
- * super_admin
 - role_index = 4
 - authorized capabilities
 - o (include all the capabilities authorized for a group_admin)
 - o canChangeRoomMembershipStyle

- o canChangePreauthorizedUserList
 - o canDestroyRoom
- constraints
 - o minimum_participants_constraint = 0
 - o maximum_participants_constraint = null
 - o minimum_active_participants_constraint = 0
 - o maximum_active_participants_constraint = null
 - o authorized_role_changes = [(0,[1,2,3,4]), (1,[0,2,3,4]), (2,[0,1,3,4]), (3,[0,1,2,4]), (4,[0,1,2,3])]
- * policy_enforcer
 - role_index = 5
 - capabilities
 - o (does not include any other capabilities)
 - o canRemoveParticipant
 - o canChangeUserRole
 - o canBan
 - o canUnban
 - o canChangeRoleDefinitions
 - o canChangePreauthorizedUserList
 - o canChangeMlsOperationalPolicies
 - o canDestroyRoom
 - o canSendMLSReinitProposal
 - constraints
 - o minimum_participants_constraint = 1
 - o maximum_participants_constraint = 2

- o minimum_active_participants_constraint = 0
- o maximum_active_participants_constraint = 0
- o authorized_role_changes = [(0,[1]), (1,[0]), (2,[0,1]),
(3,[0,1]), (4,[0,1])]
- Notes: can remove a banned user from the list (cleanup) but not
restore them

A.2. Strictly administered room

This is an example set of role policies, which is suitable for friends and family rooms and small groups of peers in a workgroup or club.

- * no_role
 - role_index = 0
 - authorized capabilities
 - o canUseJoinCode
 - constraints
 - o minimum_participants_constraint = 0
 - o maximum_participants_constraint = null
 - o minimum_active_participants_constraint = 0
 - o maximum_active_participants_constraint = 0
 - o authorized_role_changes = [(0,[2])]
- * banned
 - role_index = 1
 - no capabilities
 - constraints
 - o minimum_participants_constraint = 0
 - o maximum_participants_constraint = null

- o minimum_active_participants_constraint = 0
 - o maximum_active_participants_constraint = 0
 - o authorized_role_changes = []
- * ordinary_user
- role_index = 2
 - authorized_capabilities
 - o canAddOwnClient
 - o canAddSelf
 - o canRemoveOwnClient
 - o canRemoveSelf
 - o canChangeOwnRole
 - o canSendMessage
 - o canReceiveMessage
 - o canCopyMessage
 - o canReportAbuse
 - o canReactToMessage
 - o canDeleteOwnReaction
 - o canEditOwnMessage
 - o canDeleteOwnMessage
 - o canStartTopic
 - o canReplyInTopic
 - o canUploadImage
 - o canUploadVideo
 - o canUploadSound

- o canUploadAttachment
- o canDownloadImage
- o canDownloadVideo
- o canDownloadSound
- o canDownloadAttachment
- o canSendLink
- o canSendLinkPreview
- o canFollowLink
- o canCopyLink
- o canChangeOwnName
- o canChangeOwnPresence
- o canChangeOwnMood
- o canChangeOwnAvatar
- constraints
 - o minimum_participants_constraint = 0
 - o maximum_participants_constraint = null
 - o minimum_active_participants_constraint = 0
 - o maximum_active_participants_constraint = null
 - o authorized_role_changes = [(0,[2]), (2,[0])]
- * group_admin
 - role_index = 3
 - authorized capabilities
 - o (include all the capabilities authorized for an ordinary_user)
 - o canAddParticipant

- o canRemoveParticipant
- o canBan
- o canUnBan
- o canKick
- o canChangeUserRole
- o canCreateJoinCode - reserved for future use
- o canDeleteOtherReaction
- o canDeleteOtherMessage
- o canEditOwnTopic
- o canEditOtherTopic
- o canChangeRoomDescription
- o canChangeRoomName
- o canChangeRoomAvatar
- o canChangeRoomSubject
- o canChangeRoomMood
- constraints
 - o minimum_participants_constraint = 1
 - o maximum_participants_constraint = null
 - o minimum_active_participants_constraint = 0
 - o maximum_active_participants_constraint = null
 - o authorized_role_changes = [(0,[1,2,3]), (1,[0,2,3]), (2,[0,1,3]), (3,[0,1,2])]
- * super_admin
 - role_index = 4
 - authorized capabilities

- o (include all the capabilities authorized for a group_admin)
- o canChangeRoomMembershipStyle
- o canChangeRoleDefinitions
- o canChangePreauthorizedUserList
- o canDestroyRoom
- o canSendMLSReinitProposal
- constraints
 - o minimum_participants_constraint = 0
 - o maximum_participants_constraint = null
 - o minimum_active_participants_constraint = 0
 - o maximum_active_participants_constraint = null
 - o authorized_role_changes = [(0,[1,2,3,4]), (1,[0,2,3,4]), (2,[0,1,3,4]), (3,[0,1,2,4]), (4,[0,1,2,3])]
- * policy_enforcer
 - role_index = 5
 - capabilities
 - o (does not include any other capabilities)
 - o canRemoveParticipant
 - o canChangeUserRole
 - o canBan
 - o canUnban
 - o canChangeRoleDefinitions
 - o canChangePreauthorizedUserList
 - o canChangeMlsOperationalPolicies
 - o canDestroyRoom

- o canSendMLSReinitProposal
- constraints
 - o minimum_participants_constraint = 1
 - o maximum_participants_constraint = 2
 - o minimum_active_participants_constraint = 0
 - o maximum_active_participants_constraint = 0
 - o authorized_role_changes = [(0,[1]), (1,[0]), (2,[0,1]), (3,[0,1]), (4,[0,1])]
- Notes: can remove a banned user from the list (cleanup) but not restore them

A.3. Moderated room

- * no_role
 - role_index = 0
 - authorized capabilities
 - o canUseJoinCode
 - constraints
 - o minimum_participants_constraint = 0
 - o maximum_participants_constraint = null
 - o minimum_active_participants_constraint = 0
 - o maximum_active_participants_constraint = null
 - o authorized_role_changes = [(0,[2,3])]
- * banned
 - role_index = 1
 - no capabilities
 - constraints

- o minimum_participants_constraint = 0
 - o maximum_participants_constraint = null
 - o minimum_active_participants_constraint = 0
 - o maximum_active_participants_constraint = 0
 - o authorized_role_changes = []
- * guest
- role_index = 2
 - authorized_capabilities
 - o canRemoveSelf
 - o canReceiveMessage
 - o canCopyMessage
 - o canReactToMessage
 - o canDeleteOwnReaction
 - o canFollowLink
 - o canCopyLink
 - o canDownloadImage
 - o canDownloadVideo
 - o canDownloadSound
 - constraints
 - o minimum_participants_constraint = 0
 - o maximum_participants_constraint = null
 - o minimum_active_participants_constraint = 0
 - o maximum_active_participants_constraint = null
 - o authorized_role_changes = [(0,[2]), (2,[0])]

- * attendee
 - role_index = 3
 - authorized capabilities
 - o (include all the capabilities authorized for a guest)
 - o canAddOwnClient
 - o canAddSelf
 - o canRemoveOwnClient
 - o canChangeOwnRole
 - o canReportAbuse
 - o canReplyInTopic
 - o canDownloadAttachment
 - o canChangeOwnName
 - o canChangeOwnPresence
 - o canChangeOwnAvatar
 - constraints
 - o minimum_participants_constraint = 0
 - o maximum_participants_constraint = null
 - o minimum_active_participants_constraint = 0
 - o maximum_active_participants_constraint = null
 - o authorized_role_changes = [(0,[3]), (3,[0])]
- * speaker
 - role_index = 4
 - authorized capabilities
 - o (include all the capabilities authorized for a speaker)

- o canSendMessage
- o canEditOwnMessage
- o canDeleteOwnMessage
- o canStartTopic
- o canUploadImage
- o canUploadVideo
- o canUploadSound
- o canUploadAttachment
- o canSendLink
- o canSendLinkPreview
- constraints
 - o minimum_participants_constraint = 0
 - o maximum_participants_constraint = null
 - o minimum_active_participants_constraint = 0
 - o maximum_active_participants_constraint = null
 - o authorized_role_changes = [(0,[4]), (4,[0])]
- * moderator
 - role_index = 5
 - authorized capabilities
 - o (include all the capabilities authorized for an ordinary_user)
 - o canAddParticipant
 - o canRemoveParticipant
 - o canBan
 - o canUnBan

- o canKick
- o canChangeUserRole
- o canCreateJoinCode - reserved for future use
- o canDeleteOtherReaction
- o canDeleteOtherMessage
- o canEditOwnTopic
- o canEditOtherTopic
- o canChangeRoomName
- o canChangeRoomAvatar
- o canChangeRoomSubject
- o canChangeRoomMood
- constraints
 - o minimum_participants_constraint = 1
 - o maximum_participants_constraint = null
 - o minimum_active_participants_constraint = 0
 - o maximum_active_participants_constraint = null
 - o authorized_role_changes = [(0,[1,2,3,4,5]), (1,[0,2,3,4,5]), (2,[0,1,3,4,5]), (3,[0,1,2,4,5]), (4,[0,1,2,3,5]), (5,[0,1,2,3,4])]
- * super_admin
 - role_index = 6
 - authorized capabilities
 - o (include all the capabilities authorized for a moderator)
 - o canChangeRoomDescription
 - o canChangeRoomMembershipStyle

- o canChangeRoleDefinitions
- o canChangePreauthorizedUserList
- o canDestroyRoom
- o canSendMLSReinitProposal
- constraints
 - o minimum_participants_constraint = 0
 - o maximum_participants_constraint = null
 - o minimum_active_participants_constraint = 0
 - o maximum_active_participants_constraint = null
 - o authorized_role_changes = [(0,[1,2,3,4,5,6]),
(1,[0,2,3,4,5,6]), (2,[0,1,3,4,5,6]), (3,[0,1,2,4,5,6]),
(4,[0,1,2,3,5,6]), (5,[0,1,2,3,4,6]), (6,[0,1,2,3,4,5])]
- * policy_enforcer
 - role_index = 7
 - capabilities
 - o (does not include any other capabilities)
 - o canRemoveParticipant
 - o canChangeUserRole
 - o canBan
 - o canUnban
 - o canChangeRoleDefinitions
 - o canChangePreauthorizedUserList
 - o canChangeMlsOperationalPolicies
 - o canDestroyRoom
 - o canSendMLSReinitProposal

- constraints
 - o minimum_participants_constraint = 1
 - o maximum_participants_constraint = 2
 - o minimum_active_participants_constraint = 0
 - o maximum_active_participants_constraint = 0
 - o authorized_role_changes = [(0,[1]), (1,[0]), (2,[0,1]), (3,[0,1]), (4,[0,1]), (5, [0,1]), (6, [0,1])]
- Notes: can remove a banned user from the list (cleanup) but not restore them

A.4. Multi-organization administered room

In this example room policy, Alice from organization A is a super admin. There are per organization user and admin roles for orgs A, B, and C. Organizational admins can only move users to and from their org user role, their org admin role, the no_role; and can ban (but not unban) their own org users. The non-host orgs do not have the canChangeOwnRole and canAddSelf, and are limited to 3 admins per org.

* no_role

- role_index = 0
- no capabilities
- constraints
 - o minimum_participants_constraint = 0
 - o maximum_participants_constraint = null
 - o minimum_active_participants_constraint = 0
 - o maximum_active_participants_constraint = 0
 - o authorized_role_changes = []

* banned

- role_index = 1

- no capabilities
- constraints
 - o minimum_participants_constraint = 0
 - o maximum_participants_constraint = null
 - o minimum_active_participants_constraint = 0
 - o maximum_active_participants_constraint = 0
 - o authorized_role_changes = []
- * org_a_user
 - role_index = 2
 - authorized capabilities
 - o (all capabilities of org_b_user)
 - o canChangeOwnRole
 - o canAddSelf
 - constraints
 - o minimum_participants_constraint = 0
 - o maximum_participants_constraint = null
 - o minimum_active_participants_constraint = 0
 - o maximum_active_participants_constraint = null
 - o authorized_role_changes = [(0,[2]), (2,[0])]
- * org_b_user
 - role_index = 3
 - authorized capabilities
 - o canRemoveSelf
 - o canAddOwnClient

- o canRemoveOwnClient
- o canSendMessage
- o canReceiveMessage
- o canCopyMessage
- o canReportAbuse
- o canReplyInTopic
- o canReactToMessage
- o canDeleteOwnReaction
- o canEditOwnMessage
- o canSendLink
- o canSendLinkPreview
- o canFollowLink
- o canCopyLink
- o canDownloadImage
- o canDownloadVideo
- o canDownloadSound
- o canDownloadAttachment
- o canChangeOwnName
- o canChangeOwnPresence
- o canChangeOwnAvatar
- constraints
 - o minimum_participants_constraint = 0
 - o maximum_participants_constraint = null
 - o minimum_active_participants_constraint = 0

- o maximum_active_participants_constraint = null
 - o authorized_role_changes = [(0,[3]), (3,[0])]
- * org_c_user
 - role_index = 4
 - authorized capabilities
 - o (same capabilities as org_b_user)
 - constraints
 - o minimum_participants_constraint = 0
 - o maximum_participants_constraint = null
 - o minimum_active_participants_constraint = 0
 - o maximum_active_participants_constraint = null
 - o authorized_role_changes = [(0,[4]), (4,[0])]
- * org_a_admin
 - role_index = 5
 - authorized capabilities
 - o (all capabilities of org_b_admin)
 - o canChangeOwnRole
 - o canAddSelf
 - constraints
 - o minimum_participants_constraint = 0
 - o maximum_participants_constraint = null
 - o minimum_active_participants_constraint = 0
 - o maximum_active_participants_constraint = null
 - o authorized_role_changes = [(0,[2,5]), (2,[0,1,5]), (5,[0,1,2])]

- * org_b_admin
 - role_index = 6
 - authorized capabilities
 - o (all capabilities of org_b_user)
 - o canDeleteOwnMessage
 - o canStartTopic
 - o canUploadImage
 - o canUploadVideo
 - o canUploadSound
 - o canUploadAttachment
 - o canAddParticipant
 - o canRemoveParticipant
 - o canBan
 - o canKick
 - o canChangeUserRole
 - constraints
 - o minimum_participants_constraint = 1
 - o maximum_participants_constraint = 3
 - o minimum_active_participants_constraint = 1
 - o maximum_active_participants_constraint = null
 - o authorized_role_changes = [(0,[3,6]), (3,[0,1,6]), (6,[0,1,3])]
- * org_c_admin
 - role_index = 7
 - authorized capabilities

- o (all capabilities of org_b_admin)
- constraints
 - o minimum_participants_constraint = 1
 - o maximum_participants_constraint = 3
 - o minimum_active_participants_constraint = 1
 - o maximum_active_participants_constraint = null
 - o authorized_role_changes = [(0,[4,7]), (4,[0,1,7]), (7,[0,1,4])]
- * super_admin
 - role_index = 8
 - authorized capabilities
 - o (include all the capabilities authorized for org_a_admin)
 - o canUnBan
 - o canDeleteOtherReaction
 - o canDeleteOtherMessage
 - o canEditOwnTopic
 - o canEditOtherTopic
 - o canChangeRoomDescription
 - o canChangeRoomName
 - o canChangeRoomAvatar
 - o canChangeRoomSubject
 - o canChangeRoomMood
 - o canChangeRoomMembershipStyle
 - o canChangeRoleDefinitions
 - o canChangePreauthorizedUserList

- o canDestroyRoom
 - o canSendMLSReinitProposal
- constraints
 - o minimum_participants_constraint = 1
 - o maximum_participants_constraint = null
 - o minimum_active_participants_constraint = 1
 - o maximum_active_participants_constraint = null
 - o authorized_role_changes = [(0,[1,2,3,4,5,6,7,8]),
(1,[0,2,3,4,5,6,7,8]), (2,[0,1,5,8]), (3,[0,1,6]),
(4,[0,1,7]), (5,[0,1,2,8]), (6,[0,1,3]), (7,[0,1,4]),
(8,[0,1,2,5])]
- * policy_enforcer
 - role_index = 9
 - capabilities
 - o (does not include any other capabilities)
 - o canRemoveParticipant
 - o canChangeUserRole
 - o canBan
 - o canUnban
 - o canChangeRoleDefinitions
 - o canChangePreauthorizedUserList
 - o canChangeMlsOperationalPolicies
 - o canDestroyRoom
 - o canSendMLSReinitProposal
 - constraints
 - o minimum_participants_constraint = 1

- o maximum_participants_constraint = 2
- o minimum_active_participants_constraint = 0
- o maximum_active_participants_constraint = 0
- o authorized_role_changes = [(0,[1]), (1,[0]), (3,[0,1]), (4,[0,1]), (5,[0,1]), (6,[0,1]), (7,[0,1]), (8,[0,1])]
- Notes: can remove a banned user from the list (cleanup) but not restore them

Appendix B. Complete TLS Presentation Language Syntax

```
enum {
    false(0),
    true(1)
} bool;

struct {
    /* a valid Uniform Resource Identifier (URI) */
    opaque uri<V>;
} Uri;

enum {
    optional(0),
    required(1),
    forbidden(2)
} Optionality;

enum {
    reserved(0)
    ordinary(1),
    fixed-membership(2),
    parent-dependent(3),
    (255)
} MembershipStyle;

struct {
    Optionality logging;
    bool enabled;
    Uri logging_clients<V>;
    Uri machine_readable_policy;
    Uri human_readable_policy;
} LoggingPolicy;

struct {
    bool on_request;
```

```
    Uri join_link;
    bool multiuser;
    uint32 expiration;
    Uri link_requests;
} LinkPolicy;

struct {
    opaque name<V>;
    opaque description<V>;
    Uri homepage;
    Role bot_role;
    bool can_read;
    bool can_write;
    bool can_target_message_in_group;
    bool per_user_content;
} Bot;

struct {
    Optionality history_sharing;
    Role who_can_share<V>;
    bool automatically_share;
    uint32 max_time_period;
} HistoryPolicy;

enum {
    null(0),
    boolean(1),
    number(2),
    string(3),
    jsonObject(4)
} ExtType;

struct {
    opaque name<V>;
    ExtType type;
    opaque value<V>;
} PolicyExtension;

struct {
    ProtocolVersion versions<V>;
    CipherSuite cipher_suites<V>;
    ExtensionType extensions<V>;
    ProposalType proposals<V>;
    CredentialType credentials<V>;
    WireFormats wire_formats<V>;
    ComponentID component_ids<V>;
    ComponentID safe_aad_types<V>;
    MediaType media_types<V>;
```

```
    ContentType content_types<V>;
} ExtendedCapabilities;

enum {
    unspecified(0),
    immediateCommit(1),
    randomDelay(2),
    preferenceWheel(3),
    designatedCommitter(4),
    (255)
} PendingProposalStrategy;

struct {
    PendingProposalStrategy pending_proposal_strategy;
    uint64 minimum_delay_ms;
    uint64 maximum_delay_ms;
} PendingProposalPolicy;

struct {
    uint64 minimum_time;
    uint64 default_time;
    uint64 maximum_time;
} MinDefaultMaxTime;

struct {
    uint8 epoch_tolerance;
    uint16 pad_to_size;
    uint32 max_generations_skipahead;
} AppMessagePolicy;

struct {
    ExtendedCapabilities mandatory_capabilities;
    ExtendedCapabilities default_capabilities;
    ExtendedCapabilities forbidden_capabilities;
    WireFormats handshake_formats<V>;
    bool external_proposal_allowed;
    bool external_commit_allowed;
    PendingProposalPolicy pending_proposal_policy;
    MinDefaultMaxTime LeafNode_update_time;
    AppMessagePolicy app_message_policy;
    uint64 max_kp_lifetime;
    uint64 max_credential_lifetime;
    uint64 resumption_psk_lifetime;
    MinDefaultMaxTime sender_nonce_keypair_lifetime;
    uint32 max_keypairs;
    MinDefaultMaxTime buffer_incoming_message_time;
    uint32 max_buffered_messages;
```

```
} OperationalParameters;
```

```
struct {  
    MembershipStyle membership_style;  
    bool multi_device;  
    Uri parent_room_uri;  
    bool persistent_room;  
    Optionality delivery_notifications;  
    Optionality read_receipts;  
    bool semi_anonymous_ids;  
    bool discoverable;  
    LinkPolicy link_policy;  
    LoggingPolicy logging_policy;  
    HistoryPolicy history_sharing;  
    Bot allowed_bots<V>;  
    OperationalParameters operational_parameters;  
    PolicyExtension policy_extensions<V>;  
} RoomPolicy;
```

```
RoomPolicy room_policy;
```

Acknowledgments

TODO acknowledge.

Author's Address

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