Workgroup: More Instant Messaging Interoperability
Internet-Draft: draft-ralston-mimi-linearized-matrix-03
Published: 10 July 2023
Intended Status: Standards Track
Expires: 11 January 2024
Authors: T. Ralston
         The Matrix.org Foundation C.I.C.
         M. Hodgson
         The Matrix.org Foundation C.I.C.

Abstract

This document specifies Linearized Matrix (LM). LM is an extensible
protocol for interoperability between messaging providers, using
Matrix's (matrix.org) decentralized room model. LM simplifies the
Directed Acyclic Graph (DAG) persistence of Matrix while maintaining
compatibility with non-linearized servers within a room. It does this
by using a doubly-linked list of events/messages per room with hub
and spoke fanout.

LM's extensibility enables a wide range of transport protocol and
end-to-end encryption possibilities. This document uses Matrix's room
access control semantics supported by Messaging Layer Security (MLS),
transported via HTTPS and JSON. The details of which server-to-server
transport to use and what a is put over MLS are replaceable.

The threat model of LM does not place trust in a central owning
server for each conversation. Instead, it defines a hub server which
handles maintaining linearized room history for other servers in the
room. This model permits transparent interconnection between LM
servers and Matrix servers, in the same 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://
turt2live.github.io/ietf-mimi-linearized-matrix/draft-ralston-mimi-
linearized-matrix.html. Status information for this document may be
found at https://datatracker.ietf.org/doc/draft-ralston-mimi-
linearized-matrix/.

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/.
Source for this draft and an issue tracker can be found at https://github.com/turt2live/ietf-mimi-linearized-matrix.

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 11 January 2024.

Copyright Notice

Copyright (c) 2023 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
2. Conventions and Definitions
3. Architecture
   3.1. Server Names
3.2. Rooms
   3.2.1. Room Versions
3.3. Users
3.4. Devices
3.5. Events
   3.5.1. Linearized PDU
   3.5.2. State Events
   3.5.3. Event Types
4. MLS Considerations
   4.1. Device Credentials
   4.2. Group Creation
   4.3. Updating Group State
   4.4. Key Packages
   4.5. Room Event Types
      4.5.1. m.mls.commit
      4.5.2. m.room.encrypted
5. Processing Events
   5.1. Receiving Events/PDUs
   5.2. Authorization Rules
      5.2.1. Auth Events Selection
      5.2.2. Calculating Power Levels
      5.2.3. Auth Rules Algorithm
6. Signing
   6.1. Signing Events
   6.2. Signing Arbitrary Objects
   6.3. Checking Signatures
7. Canonical JSON
8. Event Redactions
9. Hashes
   9.1. Content Hash Calculation
   9.2. Reference Base64
10. Unpadded Base64
11. Hub Selection
   11.1. Hub Transfers
12. Transport
   12.1. TLS Certificates
   12.2. API Standards
      12.2.1. Requests and Responses
      12.2.2. Errors
      12.2.3. Unsupported Endpoints
      12.2.4. Malformed Requests
      12.2.5. Transaction Identifiers
      12.2.6. Rate Limiting
   12.3. Resolving Server Names
      12.3.1. GET /.well-known/matrix/server
   12.4. Request Authentication
      12.4.1. Retrieving Server Keys
   12.5. Sending Events
      12.5.1. PUT /matrix/federation/v2/send/:txnId
   12.6. Event and State APIs
      12.6.1. GET /matrix/federation/v2/event/:eventId
      12.6.2. GET /matrix/federation/v1/state/:roomId
      12.6.3. GET /matrix/federation/v1/state_ids/:roomId
      12.6.4. GET /matrix/federation/v2/backfill/:roomId
   12.7. Room Membership
      12.7.1. Make and Send Handshake
      12.7.2. Invites
1. Introduction

Linearized Matrix derives from Matrix (matrix.org). Matrix is an interoperable decentralized communications protocol with an open specification. Matrix uses a Directed Acyclic Graph (DAG) to persist rooms, synchronizing the DAG between servers. Linearized Matrix uses an append-only doubly-linked list. A hub server is present in each room to persist the linked list, and to handle linearization of events (user/application messages, as well as room configuration changes) from other servers.

The hub server in a room does not act as an owner for the room. All rooms support the doubly-linked list and a DAG at the same time. The precise details of DAG and linked list interconnection are not covered by this document; they are out of scope for the More Instant Messaging Interoperability (MIMI) working group. Full details are available within the Matrix specification process as [MSC3995]. Where applicable, this document covers the mandatory components.

As rooms support these two representations, this permits a variable threat model. Messaging providers who want to deliver all their messages/don't trust other servers to deliver their messages would choose the DAG representation, at the cost of implementation complexity. Providers using the Linearized Matrix representation place trust in the hub server to deliver the messages. Those providers attach verifiable hashes and signatures to each event as a safeguard against the hub server modifying the events.
The primary exports of Linearized Matrix are the room model and the rules which govern how a room accepts events. This document specifies a server-centric approach, where one or more servers perform access control. With some changes, it can become client-centric too. This document also specifies a transport for synchronizing the doubly-linked list to other servers. More efficient and scalable transport methods should replace this example.

In a similar fashion, this document specifies how Messaging Layer Security (MLS) [I-D.ietf-mls-protocol] [I-D.ietf-mls-architecture] could run over a Linearized Matrix room. User messages use MLS, while state events (room configuration information) are plain-text in this iteration of the document. Clients synchronize room and MLS group membership, while servers verify those memberships. This ensures that users who are not in the room are unable to gain access to encrypted messages.

A key component for interoperability is consistent access control semantics. Where a single server 'owns' a room, it can establish arbitrary measures. For example, an owning provider might decide that a different kind of password is required to join a room. This diminishes the user experience on providers who do not (or can not) support those custom measures. With decentralization, all participating servers must use the same consistent set of access controls. Linearized Matrix uses a decentralized room model for this reason, with linear room history for ease of implementation.

Linearized Matrix also supports transferring a hub to a different server in the room. This enables two major features: a hub server that wishes to leave the room can do so, as it is no longer responsible for handling events for the room, and because the hub could change at any time, access control semantics must remain consistent. As a bonus, hub transfers make it possible for a (non-linearized) Matrix server to take hub status. When it becomes the hub, it takes responsibility for linearizing the room's DAG for other Linearized Matrix servers in the room. The DAG linearization algorithm is not specified in this document; it is out of scope for the MIMI working group.

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.

This document uses [I-D.ralston-mimi-terminology] for the majority of terminology.
This document additionally defines:

**Rejection**: An event which is "rejected" is not relayed to any local clients and is not appended to the room in any way.

**Soft Failure**: An event which is "soft-failed" **SHOULD NOT** be relayed to any local clients, nor be used as in auth_events if possible. The event is otherwise appended to the room as per usual.

Further terms are introduced in-context within this document.

**TODO**: We should move/copy those definitions up here anyways.

### 3. Architecture

For a given conversation/room:

In this diagram, events (**Section 3.5**) are objects which carry information about the room as well as messages between users within that room.

Server A is acting as a hub for the other two servers in the diagram. Servers B and C do not converse directly when sending events to the room: those events are instead sent to the hub which then distributes them back out to all participating servers. Servers communicate with each other using the API surface described by **Section 12**.
Clients are shown in the diagram here for demonstrative purposes only. No Client-Server API or other requirements of clients are specified in this document.

This leads to two distinct roles:

* **Hub server**: the server responsible for holding conversation history on behalf of other servers in the room.

* **Participant server**: any non-hub server. This server **MAY** persist conversation history or rely on the hub server instead.

**OPEN QUESTION**: Should we support having multiple hubs for increased trust between participant and hub? (participant can pick the hub it wants to use rather than being forced to use a single hub)

**OPEN QUESTION**: Should we instead require that each server persist events they create, with the hub being responsible for purely distribution/fetching? It'd be in the hub's best interest to store all history, but equally if it can rebuild the room by reaching out to origin servers then it might be fine enough (it'd only be storing event IDs rather than whole events).

### 3.1. Server Names

Each messaging provider is referred to as a "server" and has a "domain name" or "server name" to uniquely identify it. This server name is then used to namespace user IDs, room IDs/aliases, etc.

A server name **MUST** be compliant with Section 2.1 of [RFC1123]. Improper server names **MUST** be considered "uncontactable" by a server.

A server **MUST NOT** use a literal IPv4 or IPv6 address as a server name. Doing so reduces the ability for the server to move to another internet address later, and IP addresses are generally difficult to acquire a certificate for (required in Section 12). Additionally, servers **SHOULD NOT** use an explicit port in their server name for similar portability reasons.

The approximate ABNF [RFC5234] grammar for a server name is:

```plaintext
server_name = hostname [ "":" port ]
port = 1*5DIGIT
hostname = 1*255dns-char
dns-char = DIGIT / ALPHA / "." / "."
```

Server names **MUST** be treated as case sensitive (eXaMpLe.OrG, example.org, and EXAMPLE.ORG are 3 different servers). Server names **SHOULD** be lower case (example.org) and **SHOULD NOT** exceed 230 characters for ease of use. The 230 characters specifically gives
room for a suitably long localpart while being within the 255 allowable characters from Section 2.1 of [RFC1123].

Examples:

*example.org (DNS host name)

*example.org:5678 (DNS host name with explicit port)

3.2. Rooms

Rooms hold the same definition under [I-D.ralston-mimi-terminology]: a conceptual place where users send and receive events. All users with sufficient access to the room receive events sent to that room.

The different chat types are represented by rooms through state events (Section 3.5.2), which ultimately change how the different algorithms in the room version (Section 3.2.1) behave.

Rooms have a single internal "Room ID" to identify them from another room. The ABNF [RFC5234] grammar for a room ID is:

```
room_id = "!" room_id_localpart ":" server_name
room_id_localpart = 1*opaque
opaque = DIGIT / ALPHA / "-" / "." / "~" / "_"

server_name is inherited from Section 3.1.
```

room_id MUST NOT exceed 255 characters and MUST be treated as case sensitive.

Example: !abc:example.org.

The server_name for a room ID does NOT indicate the room is "hosted" or served by that domain. The domain is used as a namespace to prevent another server from maliciously taking over a room. The server represented by that domain may no longer be participating in the room.

The entire room ID after the ! sigil MUST be treated as opaque. No part of the room ID should be parsed, and cannot be assumed to be human-readable.

3.2.1. Room Versions

TODO: We should consider naming this something else.

Each room declares which room version it's using, and each room version (identified by a single string) describes the specific algorithms a server needs to follow in order to participate in rooms
with that version. Room versions are immutable once specified, as otherwise a change in algorithms could cause a split-brain between servers participating in affected rooms.

Room versions prefixed with I. **MUST** only be used within the IETF specification process. Room versions consisting solely of 0-9 and . **MUST** only be used by the Matrix protocol.

This document as a whole describes I.1 as a room version.

Servers **MUST** implement support for I.1 at a minimum. Servers **SHOULD** use I.1 when creating new rooms.

**Implementation note:** Currently I.1 is not a real thing. Use org.matrix.i-d.ralston-mimi-linearized-matrix.02 when testing against other Linearized Matrix implementations. This string may be updated later to account for breaking changes.

**Implementation note:** org.matrix.i-d.ralston-mimi-linearized-matrix.00 also exists in the wild, defining a set of algorithms which exist in a prior version of this document (00 and 01).

**TODO:** Remove implementation notes.

There is no implicit ordering or hierarchy to room versions. Future room versions, such as an I.2, can choose to build upon I.1's algorithms or start completely from scratch if they prefer.

A room version has the following algorithms defined:

* Event authorization - Rules which govern when events are accepted, rejected, or soft-failed by a server. For I.1, this is Section 5.2.

* Redaction - Description of which fields to keep on an event during redaction. Redaction is used by the signing and hash algorithms, meaning they need to be consistent across implementations. For I.1, this is Section 8.

* Event format - Which fields are expected to be present on an event, and the schema for each. For I.1, this is Section 3.5.

* Canonical JSON - Specific details about how to canonicalize an event as JSON. This is used by the signing algorithm and must be consistent between implementations. For I.1, this is Section 7.

* Hub selection - Rules around picking the hub server and transferring to a new hub. For I.1, this is Section 11.
Identifier grammar - All identifiers (room IDs, user IDs, event IDs, etc) can change grammar within a room version. As such, they SHOULD generally be treated as opaque as possible over a transport. For I.1, these details are described in Section 3.1, Section 3.2, Section 3.3, Section 3.4, and Section 3.5.

The transport between servers is decoupled from the algorithms above. For example, events are treated as blobs with no specific format over the wire but have strict schema in the context of a room or room version. Endpoints MUST be designed with this distinction in mind.

Each room version has a "stable" or "unstable" designation. Stable room versions SHOULD be used in production by messaging providers. Unstable room versions might contain bugs or are not yet fully specified and SHOULD NOT be used in production by messaging providers.

I.1 shall be considered "stable".

Implementation note: org.matrix.i-d.ralston-mimi-linearized-matrix.02 is considered "unstable".

TODO: Remove implementation notes.

TODO: Matrix considers a version as stable once accepted through FCP. When would be the process equivalent for the IETF?

The ABNF [RFC5234] grammar for a room version is:

\[
\text{room_version} = 1*128\text{room_version_char}
\]

\[
\text{room_version_char} = \text{DIGIT} / %x61-7A ; a-z / "." / "."
\]

Examples:

*1

*I.1

*org.example.my-room-version

Room versions not formally specified SHOULD be prefixed using reverse domain name notation, creating a sort of namespace. org.example.my-room-version is an example of this.

3.3. Users

As described by [I-D.ralston-mimi-terminology], a user is typically a human which operates a client. Each user has a distinct user ID.
The ABNF [RFC5234] grammar for a user ID is:

```abnf
user_id = "@" user_id_localpart ":" server_name
user_id_localpart = 1*user_id_char
user_id_char = DIGIT
    / %x61-7A ; a-z
    / "." / "." / ":=
    / ":" / "/" / "+"
```

server_name is inherited from Section 3.1.

user_id **MUST NOT** exceed 255 characters and **MUST** be treated as case sensitive.

Examples:

* alice@example.org

* watch/for/slashes:example.org

user_id_localpart **SHOULD** be human-readable and notably **MUST NOT** contain uppercase letters.

server_name denotes the domain name (Section 3.1) which allocated the ID, or would allocate the ID if the user doesn't exist yet.

Identity systems and messaging providers **SHOULD NOT** use a phone number in a localpart. Using a phone number would mean that when a human operator's changes their phone number then they'd lose access to their existing chats and potentially gain access to any chats the new number is participating in. Providers which use phone numbers **SHOULD** ensure the new user ID is not in chats belonging to a different logical user. To prevent this case, a GUID (scoped to the allocating server) or an account ID is recommended as a localpart, allowing users to change their phone number without losing chats.

This document does not define how a user ID is acquired. It is expected that an identity specification under MIMI will handle resolving email addresses, phone numbers, names, and other common queries to user IDs.

User IDs are sometimes informally referenced as "MXIDs", short for "Matrix User IDs".

### 3.4. Devices

Each user can have zero or more devices/active clients. These devices are intended to be members of the MLS group and thus have their own key package material associated with them.
Because device IDs are used as "key versions" in a key ID (Section 6), they have a compatible ABNF [RFC5234] grammar:

```
device_id = 1*key_version_char
device_id_char = ALPHA / DIGIT / "_"
```

3.5. Events

All data exchanged over Linearized Matrix is expressed as an "event". Each client action (such as sending a message) correlates with exactly one event. All events have a type to distinguish them, and use reverse domain name notation to namespace custom events (for example, org.example.appname.eventname).

Event types using m. as a prefix MUST only be used by the protocol.

When events are traversing a transport to another server they are referred to as a Persistent Data Unit or PDU. Structurally, an event and PDU are the same.

An event has the following minimum fields:

* `room_id` (string; required) - The room ID for where the event is being sent. This MUST be a valid room ID (Section 3.2).

* `type` (string; required) - A UTF-8 [RFC3629] string to distinguish different data types being carried by events. Event types are case sensitive. This MUST NOT exceed 255 characters.

* `state_key` (string; optional) - A UTF-8 [RFC3629] string to further distinguish an event as a state event (see Section 3.5.2). Can be an empty string. This MUST NOT exceed 255 characters.

Each event type specifies its own state key requirements. For m.room.member (Section 3.5.3.3), the state key is the user ID (Section 3.3) for which the membership applies to. For m.room.join_rules (Section 3.5.3.2), this is an empty string. For a custom event type this may be an opaque string such as a UUID or randomly generated string.

* `sender` (string; required) - The user ID which is sending this event. This MUST be a valid user ID (Section 3.3).

* `origin_server_ts` (64-bit integer; required) - The milliseconds since the unix epoch for when this event was created.

* `hub_server` (string; optional) - When a hub server is converting an LPDU (Section 3.5.1) to a formal event, it MUST specify its own server name (Section 3.1) here. The value MUST be a valid server name.
To support interconnection with non-linearized Matrix, as discussed in Section 1, events created outside of a hub server MUST NOT populate this field.

*content (object; required) - The event content. The specific schema depends on the event type. Clients and servers processing an event MUST NOT assume the content is safe or accurately represented. Malicious clients and servers are able to send payloads which don't comply with a given schema, which may cause unexpected behaviour on the receiving side. For example, a field marked as "required" might be missing.

*hashes (object; required) - The content hashes (Section 9.1) for the event. There is a special lpdu key to contain the LPDU (partial PDU schema; see Section 3.5.1) hashes, which is itself keyed by hash algorithm and has the encoded hash as the value. The hashes object, outside of lpdu, similarly is keyed by hash algorithm with encoded hash values.

Events which specify a hub_server MUST additionally contain an lpdu hash. All other events MUST NOT contain lpdu hashes. This is to support interconnection with non-linearized Matrix, as discussed in Section 1.

*signatures (object; required) - Keyed first by domain name then by key ID, the signatures for the event.

*auth_events (array of strings; required) - The event IDs which prove the sender is able to send this event in the room. Which specific events are put here are defined by the auth events selection algorithm (Section 5.2.1).

*prev_events (array of strings; required) - This field is to support interconnection with non-linearized Matrix, discussed in Section 1. Events which specify a hub_server are expected to have exactly 1 entry in this array, while other events MAY have 1 or more entries.

Note that an event ID is not specified on the schema. Event IDs are calculated to ensure accuracy and consistency between servers. To determine the ID for an event, calculate the reference hash (Section 9.2) then encode it using URL-safe Unpadded Base64 (Section 10) and prefix that with the event ID sigil, $. For example, $nKHVqt3iyLA_HEE8lT1yUaaVjjBRR-fAqpN4t7opadc.

Using a reference hash in the event ID prevents other servers from changing what that event ID represents. If an event ID was just a UUID (or similar), any server along the send or receive path could swap the actual event payload out for a different event. The hashes and signatures in this case do nothing to protect the actual event,
as a malicious server can simply generate legal hashes/signatures as part of their modification. If we add a namespace condition where the event ID must reference the same server name as the sender ($uuid:example.org) then the sending server is free to re-assign the event ID to a different payload during the sending of that event, though all other servers along the send/receive path cannot change the event. This is particularly problematic if the event's sender is the hub for the room: the hub can send one copy of an event to half the room, and a very different copy to the other half while still maintaining the same event ID for both halves. To fix this, a hash of the event itself would need to become part of the event ID such that any modification or reassignment of the event ID is made obvious to recipient servers. This could be a tuple of (uuid, hash), though for simplicity this document uses just the reference hash on its own.

With event IDs containing hashes of the events it's theoretically possible for a sufficiently motivated person to build a database of event IDs. With that database, they could eventually determine enough information to identify parts of the event payload itself. There is not enough anchoring information on an event (or sent over the transport) for this to be efficient or effective, however.

The ABNF [RFC5234] for an event ID is:

```
event_id = "$" reference_hash
reference_hash = 1*urlsafe_unpadded_base64_char
urlsafe_unpadded_base64_char = ALPHA / DIGIT / "-" / "_"
```

If both the sender and receiver are implementing the algorithms correctly, the event ID will be the same. When different, the receiver will have issues accepting the event (none of the auth_events will make sense, for example). The sender and receiver will need to review that their implementation matches the specification in this case.

Events are treated as JSON [RFC8259] within the protocol, but can be encoded and represented by any binary-compatible format. Additional overhead may be introduced when converting between formats, however.

An example event is:
An event/PDU MUST NOT exceed 65536 bytes when formatted using Canonical JSON (Section 7). Note that this includes all signatures on the event.

Fields have no size limit unless specified above, other than the maximum 65536 bytes for the whole event.

3.5.1. Linearized PDU

All events generated by participant servers are routed through the hub, but the participant servers themselves are unable to populate fields like prev_events because they can't guarantee order and those fields contribute to the event ID, signatures, and overall validity. To fix this, participant servers send the hub server a "Linearized PDU" or "LPDU" which does not include the fields they cannot set while still ensuring integrity of the event contents themselves.
The participant server **MUST NOT** populate the following fields on events (LPDUs) they are sending to the hub:

*auth_events - the participant cannot reliably determine what allows it to send the event.
*prev_events - the participant cannot reliably know what event precedes theirs.
*hashes (except hashes.lpdu) - top-level hashes cover the above two fields.

The participant server **MUST** populate the hashes.lpdu object, covering a content hash ([Section 9.1](#)) of the partial event, giving authenticity to the sender's contents. The participant server additionally signs this partial event before sending it to the hub.

The participant server will receive an echo of the fully-formed event from the hub once appended to the room.

### 3.5.2. State Events

State events track metadata for the room, such as name, topic, and members. State is keyed by a tuple of type and state_key. The state "at" an event is the set of state events which have the most recent (in terms of event ordering, not timestamp) state tuple.

For example, consider the following (simplified) room history:
The state at index 2 consists of Alice's m.room.member event (Section 3.5.3.3) and the m.room.create event (Section 3.5.3.1) from the room. The m.room.encrypted event itself is not a state event and therefore does not get appended to the state "at" any particular event.

The state at index 4 would have Alice's new m.room.member event, Bob's m.room.member event, and the m.room.create event from before. Alice's old membership event is overridden due to having the same type and state_key as the previous event. Note however that the state at index 3 still contains the older membership event, as the new event happens later with respect to event ordering.

"Current state" is the state at the most recent event in the room. Calculating the state at a given event is needed for the authorization rules (Section 5.2) and event visibility (Section 3.5.3.5.1) algorithms. Clients additionally need to know current state to show accurate room names, topics, avatars, etc.

### 3.5.2.1. Stripped State

Stripped state event are extremely simplified state events to provide context to a user for an invite (Section 12.7.2) or knock (Section 12.7.4). Servers and clients have no ability to verify the events outside of the context for a room, so all such fields are
removed. Servers and clients **MUST NOT** rely on the events being accurate because they cannot independently verify them.

When generating stripped state for an invite or knock, the following events **SHOULD** be included if present in the current room state itself:

- *m.room.create* ([Section 3.5.3.1](#))
- *m.room.name* ([TODO](#): Link)
- *m.room.avatar* ([TODO](#): Link)
- *m.room.topic* ([TODO](#): Link)
- *m.room.join_rules* ([Section 3.5.3.2](#))
- *m.room.canonical_alias* ([TODO](#): Link)

Servers **MAY** include other event types/state keys. The above set gives users enough context to determine if they'd like to knock/join the room, as features such as the name and avatar are generally key pieces of information for a user.

Stripped state events **MUST** only have sender, type, state_key, and content from the event schema ([Section 3.5](#)).

Example:

```json
{
  "type": "m.room.create",
  "sender": "@alice:example.org",
  "state_key": "",
  "content": {
    "room_version": "I.1"
  }
}
```

### 3.5.3. Event Types

Linearized Matrix defines the following event types. The section headers are the event type.

#### 3.5.3.1. m.room.create

The very first event in the room. It **MUST NOT** have any auth_events or prev_events, and the domain of the sender **MUST** be the same as the domain in the room_id. The state_key **MUST** be an empty string.
The content for a create event **MUST** have at least a `room_version` field to denote what set of algorithms the room is using.

These conditions are checked as part of the event authorization rules ([Section 5.2](#)).

### 3.5.3.2. m.room.join_rules

Defines whether users can join without an invite and other similar conditions. The `state_key` **MUST** be an empty string. Any other state key, including lack thereof, serve no meaning and are treated as though they were a custom event.

The content for a join rules event **MUST** have at least a `join_rule` field to denote the join policy for the room. Allowable values are:

* `public` - anyone can join without an invite.
* `knock` - users must receive an invite to join, and can request an invite (knock) too.
* `invite` - users must receive an invite to join.

**TODO**: Describe restricted (and knock_restricted) rooms?

**TODO**: What's the default?

### 3.5.3.3. m.room.member

Defines the membership for a user in the room. If the user does not have a membership event then they are presumed to be in the leave state.

The `state_key` **MUST** be a non-empty string denoting the user ID the membership is affecting.

The content for a membership event **MUST** have at least a membership field to denote the membership state for the user. Allowable values are:

* `leave` - not participating in the room. If the `state_key` and sender do not match, this was a kick rather than voluntary leave.
* `join` - participating in the room.
* `knock` - requesting an invite to the room.
* `invite` - invited to participate in the room.
*ban - implies kicked/not participating. Cannot be invited or join the room without being unbanned first (moderator sends a kick, essentially).

These conditions are checked as part of the event authorization rules (Section 5.2), as are the rules for moving between membership states.

The content for a membership event MAY additionally have a reason field containing a human-readable (and usually human-supplied) description for why the membership change happened. For example, the reason why a user was kicked/banned or why they are requesting an invite by knocking.

3.5.3.4. m.room.power_levels

Defines what given users can and can't do, as well as which event types they are able to send. The enforcement of these power levels is determined by the event authorization rules (Section 5.2).

The state_key MUST be an empty string.

The content for a power levels event SHOULD have at least the following:

*ban (integer) - the level required to ban a user. Defaults to 50 if unspecified.

*kick (integer) - the level required to kick a user. Defaults to 50 if unspecified.

*invite (integer) - the level required to invite a user. Defaults to 0 if unspecified.

*redact (integer) - the level required to redact an event sent by another user. Defaults to 50 if unspecified.

*events (map) - keyed by event type string, the level required to send that event type to the room. Defaults to an empty map if unspecified.

*events_default (integer) - the level required to send events in the room. Overridden by the events map. Defaults to 0 if unspecified.

*state_default (integer) - the level required to send state events in the room. Overridden by the events map. Defaults to 50 if unspecified.

*users (map) - keyed by user ID, the level of that user. Defaults to an empty map if unspecified.
*users_default (integer) - the level for users. Overridden by the users map. Defaults to 0 if unspecified.

**TODO:** Include notifications for at-room here too?

Note that if no power levels event is specified in the room then the room creator (sender of the m.room.create state event) has a default power level of 100.

These conditions are checked as part of the event authorization rules (Section 5.2).

3.5.3.5. **m.room.history_visibility**

**TODO:** Describe.

3.5.3.5.1. **Calculating Event Visibility**

**TODO:** Describe. (when can a server see an event?). Mention that m.mls.commit is exempt.

**TODO:** This section feels like it's in the wrong place. Bring it closer to on-receive-event sections.

3.5.3.6. **TODO:** Other events

**TODO:** m.room.name, m.room.topic, m.room.avatar, m.room.canonical_alias

4. **MLS Considerations**

**TODO:** We should consider running [I-D.robert-mimi-delivery-service] over LM instead. Using something like [I-D.mahy-mimi-group-chat] would be good for more of a client-side representation of the LM room model.

The MIMI working group is chartered to use Messaging Layer Security (MLS) [I-D.ietf-mls-protocol] [I-D.ietf-mls-architecture] for encryption in chats, and this document specifies no different. Each room has a single MLS Group associated with it, both identified by the room ID (Section 3.2).

Rooms additionally track membership at a per-user level while MLS tracks group membership at a per-device level. With this consideration, commits to the MLS Group **MUST** use PublicMessage, giving the hub server an ability to inspect MLS group membership changes for illegal joins and leaves.

Encryption can only be enabled at the time the room is created. This prevents the room having encryption disabled or downgraded without an
entirely new room being created. The exact ciphersuite and other algorithmic details are contained in the content for the \texttt{m.room.create} event (\texttt{Section 3.5.3.1}):

```json
{
    "encryption": {
        "algorithm": "m.mls.v1.dhkemx25519-aes128gcm-sha256-ed25519"
    }
}
```

algorithm denotes which specific algorithm clients \textbf{MUST} use for sending and receiving encrypted events in the room. If a received event is encrypted using a different algorithm, it \textbf{MUST} be treated as undecryptable (even if the client has sufficient key information to decrypt it).

\texttt{m.mls.v1} as a prefix describes the behaviour for encrypted clients, with the remainder of the algorithm string covering the exact ciphersuite. This document uses the same mandatory ciphersuite as MLS: MLS\textsubscript{128} DHKEMX25519 AES128GCM SHA256 Ed25519. Thus, this is encoded as \texttt{m.mls.v1.dhkemx25519-aes128gcm-sha256-ed25519}. Other ciphersuites can be represented similarly, though are considered to be entirely new encryption algorithms for the purposes of this document.

Custom or non-standard encryption algorithms are possible with this approach, however out of scope for MIMI. If such an algorithm is used, it \textbf{SHOULD} be prefixed using reverse domain name notation. For example, org.example.my-encryption.

Mentioned in the introduction (\texttt{Section 1}), this document does not explore the details for what is needed to interconnect Linearized Matrix and Matrix's existing room model. However, for interconnection to be successful, extensions to MLS are needed to support decentralization. One possible extension is "Decentralised MLS" [DMLS].

### 4.1. Device Credentials

Under Section 5.3 of [I-D.ietf-mls-protocol], each MLS group member (a device, \texttt{Section 3.4}) has a "credential" or signing key associated with it. These are published to each client's local server and available over federation (\texttt{Section 12.10}).

This document relies upon out-of-band verification and therefore uses basic credentials. The format for the credential is:
struct {
    opaque user_id<V>;
    opaque device_id<V>;
    opaque signature_key<V>;
} BasicCredential;

user_id is as described by Section 3.3, and device_id is as described by Section 3.4. signature_key is from MLS.

The device then constructs the following object, signs it using each of the listed keys, and publishes it through its local server (Section 12.10.1):

```
{
    "device_id": "ABCDEF",
    "user_id": "@alice:example.org",
    "algorithms": [
        "m.mls.v1.dhkemx25519-aes128gcm-sha256-ed25519"
    ],
    "keys": {
        "m.mls.v1.credential.ed25519:ABCDEF":
            "<unpadded base64 BasicCredential>"
    },
    "signatures": {
        "@alice:example.org": {
            "m.mls.v1.credential.ed25519:ABCDEF":
                "<unpadded base64 signature>"
        }
    }
}
```

device_id is the client's device ID (Section 3.4). user_id is the user ID (Section 3.3) to which the device ID belongs. algorithms are the encryption algorithms the device supports, and SHOULD contain at least m.mls.v1.dhkemx25519-aes128gcm-sha256-ed25519.

When a device supports m.mls.v1.dhkemx25519-aes128gcm-sha256-ed25519, it MUST specify its basic credential with the m.mls.v1.credential.ed25519 key algorithm.

keys is an object containing each algorithm-specific key (or keys) for the device. The fields for the object form a key ID, with the device ID representing the "key version", as per Section 6.

All top-level fields in the object above MUST be supplied.

For each of the device's keys, a valid signature MUST be produced. If there is a missing signature from any of the keys, or from the user_id, the device information is considered invalid. Invalid
devices **MUST NOT** be members of the MLS group, and are removed if already members prior to the device information becoming invalid.

### 4.2. Group Creation

After the `m.room.create` event and other initial state events for the room are sent, the room creator **MUST** establish the appropriate MLS group. This is sent as an `m.mls.commit` event ([Section 4.5.1](#)). Afterwards, the remaining devices are added as normal ([Section 4.3](#)).

Ideally, the `m.room.create` event would also contain the initial public group state, however doing so would mean either tracking an independent MLS group ID or allowing the client to specify the room ID. While servers **MAY** allow the client to specify the room ID, servers usually have better context for which localparts ([Section 3.2](#)) are already claimed by other rooms. Having independent group IDs and room IDs can lead to confusion and a similar sort of namespacing issue (a room creator can create a conflicting group ID). Instead, the server (usually) creates the room on behalf of the client, allowing the client to then send the initial public group state to the room for other MLS members.

### 4.3. Updating Group State

This document does not provide a way to send proposals to the MLS group, meaning all commits **MUST** only contain proposals which are sent by the same member ([Section 12](#) of [I-D.ietf-mls-protocol]).

All commits are encoded as `m.mls.commit` events ([Section 4.5.1](#)) and are sent to the room. These commits are additionally encoded using `PublicMessage`, giving servers visibility on the contents of the commits. Upon receiving the event ([Section 5.1](#)), the hub server **MUST** additionally validate that any membership changes match what is possible with the room membership:

*Devices can only be added to the group if they belong to a user which is joined to the room, or if the room is "world readable" ([Section 3.5.3.5.1](#)). It is generally not enough to be invited, knocking, etc on the room - the user ID must usually be in the join state.*

*Devices can be removed in two ways:

- A device can remove another device if they both belong to the same user ID.

- A device can be removed by anyone if the user ID to which it belongs is no longer in the join state. This condition is required to satisfy a case in MLS where a device cannot self-remove itself from the group.*
If this validation fails, the hub server **MUST** reject the request if it's shaped as an LPDU ([Section 3.5.1](#)) and soft-fail the event if it's a PDU ([Section 3.5](#)).

Welcome messages are sent to devices over to-device messaging ([Section 12.10.2](#)). The message_type for the message is m.room.encrypted [**TODO**: Rename to avoid confusion with room event?] and message of:

```
{
  "algorithm": "m.mls.v1.welcome.[ciphersuite]",
  "ciphertext": "<unpadded base64 encoded welcome message>",
  "commit_event_id": "<event ID of the m.mls.commit event>"
}
```

algorithm is the ciphersuite, dhhkemx25519-aes128gcm-sha256-ed25519, prefixed with m.mls.v1.welcome.

The remaining fields are as described in the example. See [Section 10](#) for "unpadded base64".

All fields **MUST** be supplied. Note that the sender's user ID and device ID are made available over the to-device messaging endpoints ([Section 12.10.2](#)).

In all cases, a device remembers the event ID (either from the m.mls.commit event or commit_event_id from a to-device message) after decryption to associate it with the MLS epoch. The device can then do a reverse lookup of epoch to event ID to MLS group state. Note that a client **always** has access to m.mls.commit events, even when hidden by history visibility ([Section 3.5.3.5.1](#)).

**TODO**: Is it correct to say all commits are visible as "shared"?

**TODO**: We may need to store the group state in the media repo if it gets to be too big, or otherwise allow oversized events.

**TODO**: The server also likely needs to prevent devices being added to the group which don't support the ciphersuite/algorithm.

### 4.4. Key Packages

Clients "claim" another device's key package through their server ([Section 12.10.3](#)). Clients will typically generate several key packages and upload them to their server, making them available even if the client goes offline.

The algorithm for a key package is m.mls.v1.key_package.dhhkemx25519-aes128gcm-sha256-ed25519 and is combined with a device-generated key version, forming a key ID described by [Section 6](#). The key version
**SHOULD** be generated based upon the key package itself rather than using an unrelated string, such as a hash or the public key of the key package.

### 4.5. Room Event Types

**TODO:** Should these be defined with the other event types rather than here?

This document describes the following event types for use with MLS-encrypted rooms. The section headers are the event type. See [Section 3.5](#) for more information on events.

These event types are non-state events, also called "room events".

#### 4.5.1. m.mls.commit

Represents an MLS commit, which may be rejected by the hub server. The content for the event **MUST** contain at least the following example:

```json
{
  "message": "<unpadded base64 encoded PublicMessage>",
  "public_group_state": "<unpadded base64 encoded public group state>
}
```

As mentioned, message is a PublicMessage from MLS. public_group_state is to enable external joins.

An optional field, prev_commit_event_id, **SHOULD** be specified when a parent commit exists. This is to enable clients to find the commit they have keys for upon joining the room, as the most recent one may not be decryptable to them. The client can then work forwards from where they can decrypt the message.

**TODO:** Should we use the RatchetTree extension? It might make the group state massive...

**TODO:** Which fields from this need to be protected by [Section 8](#)?

#### 4.5.2. m.room.encrypted

Represents an encrypted MLS application message. The sender first encrypts the message per the content format then **MUST** send an event with content matching:
Within this document, algorithm will be m.mls.v1.dhkemx25519-aes128gcm-sha256-ed25519. The other fields are as described in the example.

Clients **SHOULD** treat m.room.encrypted events which are improperly structured as undecryptable events.

**TODO**: Which fields from this need to be protected by Section 8?

5. Processing Events

An event has several authenticity properties:

*Content hashes ([Section 9.1](#)) to cover the LPDU ([Section 3.5.1](#)) and event ([Section 3.5](#)) contents.

*Reference hashes ([Section 9.2](#)) which double as the event ID, covering the redacted event.

*Signatures from the direct senders (server name of the sender and the hub_server if present), ensuring the entities did actually produce that event.

**TODO**: Does the hub's signature actually guard anything?

These properties are validated throughout this document. Each property has different behaviour when violated. For example, a difference in content hash ultimately causes the event to be stored as a redacted copy.

5.1. Receiving Events/PDUs

**TODO**: This section conflates sending and receiving a bit more than it should. Split sending out to its own section.

When a hub receives an LPDU from a participant it **MUST** add the missing fields to create a fully formed PDU then **MUST** send that PDU back out to all participants, including the original sender.

A server is considered to have "received" an event when it does not recognize the event ID. This may be because the event has not yet been persisted, or the server is not persisting anything (in the case of a participant server). This includes when the server asks another server for an event it might be missing.
When a server (hub or participant) receives an event, it **MUST**:

1. Verify the event matches the schema for the room version ([Section 3.5](#)), without considering type-specific schemas applied to content. If an event fails to meet this requirement, it is dropped/ignored.

2. Ensure the required signatures are present and that they are valid ([Section 6.3](#)). If the event has a `hub_server` field, the event **MUST** be signed by that server. The event **MUST** also be signed by the server implied by the sender, noting that this will be an LPDU if `hub_server` is present. All other signatures **MUST NOT** be considered for signature validation, regardless of their individual validity. If the event fails to meet this requirement, it is dropped/ignored.

3. Ensure the event has a valid content hashes ([Section 9.1](#)). If the event has a `hub_server` field, it **MUST** have a content hash which covers the LPDU. If either the LPDU or PDU content hash doesn't match what the receiving server calculations, the event is redacted before further processing. The server will ultimately persist the redacted copy.

Additionally, a hub server **MUST** complete the following checks. Participant servers **SHOULD** also perform the following checks to validate that the hub server is acting in a compliant manner. If the hub is not acting appropriately (for example, by sending the participant an event which never should have been accepted), the participant server **MAY** choose to warn its local users that the room history may have been tampered with.

   1. The constraints described by [Section 4.3](#) validated, if the room is encrypted.

**5.2. Authorization Rules**

These are the rules which govern whether an event is accepted into the room, depending on the state events surrounding that event. A given event is checked against multiple different sets of state.

**5.2.1. Auth Events Selection**

The `auth_events` on an event **MUST** consist of the following state events, with the exception of an `m.room.create` event which has no `auth_events`.

1. The `m.room.create` state event.

2. The current `m.room.power_levels` state event, if any.
3. The sender's current `m.room.member` state event, if any.

4. If the type is `m.room.member`:
   1. The target's (state_key) current `m.room.member` state event, if any.
   2. If `content.membership` is `join` or `invite`, the current `m.room.join_rules` state event, if any.

    **TODO**: Talk about restricted room joins here?

5.2.2. Calculating Power Levels

A requirement of the authorization rules is being able to determine the current/future "power level" for a user. All power levels are calculated with reference to the content of an `m.room.power_levels` state event ([Section 3.5.3.4](#section-3.5.3.4)).

To calculate a user's current power level:

1. If users is present, use the power level for the user ID, if present.

2. If users is not present, or the user ID is not present in users, use users_default.

3. If users_default is not present, use 0.

To calculate the required power level to do an action:

1. If the action (kick, ban, invite, or redact) is present, use that power level.

2. If not present, use the default for the action (50 for kick, ban, and redact, 0 for invite).

To calculate the required power level to send an event:

1. If events is present, use the power level for the event type, if present.

2. If events is not present, or the event type is not present in events:
   1. If state_key is present (including empty), use state_default.
      1. If state_default is not specified, use 50.
2. If state_key is not present, use events_default.
   1. If events_default is not specified, use 0.

5.2.3. Auth Rules Algorithm

With consideration for default/calculated power levels (Section 5.2.2), each event **MUST** pass the following rules. "Current state" (and "current membership", etc) are the state of the room before the event being checked is applied.

TODO: should we reference m.federate?

1. Events must be signed (Section 6.3) by the server denoted by the sender field. Note that this may be an LPDU if the hub_server is specified and not the same server. If the event is improperly signed, reject the event.

2. If hub_server is present on the event, the event must be signed (Section 6.3) by that server. If it is improperly signed, reject the event.

3. If the event's type is m.room.create:
   1. If it has any prev_events, reject the event.
   2. If the domain of the room_id is not the same domain as the sender, reject the event.
   3. If content.room_version is not I.1, reject the event.
   4. Otherwise, allow the event.

4. Considering the event's auth_events:

   TODO: Does this check make sense for Linearized Matrix? We already removed what was #3 because it talked about rejecting events which reference rejected events.

   1. If there are duplicate entries for a given type and state_key pair, reject the event.
   2. If there are entries whose type and state_key do not match those specified by the auth events selection algorithm (Section 5.2.1), reject the event.
   3. If there is no m.room.create event among the entries, reject.
5. If the event's type is m.room.member:

   1. If there is no state_key property, or no membership in content, reject the event.

   2. If the event content's membership field is join:

      1. If the previous event is an m.room.create event and the state_key is the creator, allow the event.

      2. If sender does not match state_key, reject the event.

      3. If the sender is banned, reject the event.

      4. If the join_rule for m.room.join_rules is invite or knock, then allow the event if the current membership state is invite or join.

      5. If the join_rule for m.room.join_rules is public, allow the event.

      6. Otherwise, reject the event.

   3. If the event content's membership field is invite:

      1. If the sender's current membership state is not join, reject the event.

      2. If the target user's (state_key) membership is join or ban, reject the event.

      3. If the sender's power level is greater than or equal to the power level needed to send invites, allow the event.

      4. Otherwise, reject the event.

   4. If the event content's membership field is leave:

      1. If the sender matches the state_key, allow the event if and only if that user's current membership state is knock, join, or invite.

      2. If the sender's current membership state is not join, reject the event.

      3. If the target user's (state_key) current membership state is ban, and the sender's power level is less than the power level needed to ban other users, reject the event.
4. If the sender's power level is greater than or equal to the power level needed to kick users, and the target user's (state_key) power level is less than the sender's, allow the event.

5. Otherwise, reject the event.

5. If the event content's membership field is ban:

   1. If the sender's current membership state is not join, reject the event.

   2. If the sender's power level is greater than or equal to the power level needed to ban users, and the target user's (state_key) power level is less than the sender's power level, allow the event.

   3. Otherwise, reject the event.

6. If the event content's membership field is knock:

   1. If the join_rule for m.room.join_rules is anything other than knock, reject the event.

   2. If the sender does not match the state_key, reject the event.

   3. If the sender's current membership state is not ban or join, allow the event.

   4. Otherwise, reject the event.

7. Otherwise, the membership is unknown. reject the event.

6. If the sender's current membership state is not join, reject the event.

7. If the event type's required power level to send it is greater than the sender's power level, reject the event.

8. If the event has a state_key which starts with an @ and does not match the sender, reject the event.

   TODO: Do we care? This is theoretically to allow for owned state events, but in practice nothing which uses this concept makes it this far into the auth rules (membership events are validated above).
9. If the event's type is `m.room.power_levels`:

1. If any of the fields `users_default`, `events_default`, `state_default`, `ban`, `redact`, `kick`, or `invite` in content are present and not an integer, reject the event.

2. If `events` in content is present and not an object with values that are integers, reject the event.

3. If the `users` in content is present and not an object with valid user IDs as keys and integers as values, reject the event.

4. If there is no previous `m.room.power_levels` event in the room, allow the event.

5. For the fields `users_default`, `events_default`, `state_default`, `ban`, `redact`, `kick`, and `invite`, check if they were added, changed, or removed. For each found alteration:

   1. If the current value is higher than the sender's current power level, reject the event.

   2. If the new value is higher than the sender's current power level, reject the event.

6. For each entry being changed in or removed from `events`:

   1. If the current value is higher than the sender's current power level, reject the event.

7. For each entry being added to or changed in `events`:

   1. If the new value is greater than the sender's current power level, reject the event.

8. For each entry being changed in or removed from `users`, other than the sender's own entry:

   1. If the current value is higher than the sender's current power level, reject the event.

9. For each entry being added to or changed in `users`:

   1. If the new value is greater than the sender's current power level, reject the event.

10. Otherwise, allow the event.

10. Otherwise, allow the event.
There are some consequences to these rules:

* Unless you are already a member of the room, the only permitted operations (aside from the initial create/join) are being able to join public rooms, accept invites to rooms, and reject invites to rooms.

* To unban another user, the sender must have a power level greater than or equal to both the kick and ban power levels, and greater than the target user's power level.

**TODO:** If we want to enforce a single hub in a room, we'd do so here with auth rules.

6. Signing

All servers, including hubs and participants, publish an ed25519 [RFC8032] signing key to be used by other servers when verifying signatures. These keys can then be fetched over the transport as needed (Section 12.4.1).

**TODO:** Verify RFC reference. We might be using a slightly different ed25519 key today? See https://hdevalence.ca/blog/2020-10-04-its-25519am

Each key ID consists of an algorithm name and version. Signing keys **MUST** use an algorithm of ed25519 (and therefore **MUST** be an ed25519 key). The key version **MUST** be valid under the following ABNF [RFC5234]:

```
key_version = 1*key_version_char
key_version_char = ALPHA / DIGIT / "_"
```

An algorithm and version combined is a "key ID", delimited by : as per the following ABNF [RFC5234]:

```
key_id = key_algorithm "::" key_version
key_algorithm = "ed25519"
```

Additional key algorithms may be supported by future documents.

6.1. Signing Events

To sign an event:

1. Redact it (Section 8).

2. Sign the result as an arbitrary object (Section 6.2).
6.2. Signing Arbitrary Objects

To sign an object:

1. Remove signatures if present.

2. Encode the result with Canonical JSON (Section 7).

3. Using the relevant ed25519 signing key (usually the server's), sign the object.

4. Encode that signature under signatures using unpadded base64 (Section 10).

Note that signatures is an object with keys being the entity which did the signing and value being the key ID to encoded signature pair. See Section 3.5 for details on the signatures structure for events specifically.

6.3. Checking Signatures

If the signatures field is missing, doesn't contain the entity that is expected to have done the signing (usually a server name), doesn't have a known key ID, or is otherwise structurally invalid then the signature check fails.

If decoding the base64 fails, the check fails.

If the object is an event, redact (Section 8) it before continuing.

If removing the signatures property, canonicalizing the JSON (Section 7), and verifying the signature fails, the check fails. Note that to verify the signature the server may need to fetch another server's key first (Section 12.4.1).

Otherwise, the check passes.

TODO: Which specific signatures are required? If a server has multiple signing keys, possibly a combination of new and old, do we require all or some of them to sign?

7. Canonical JSON

When signing a JSON object, such as an event, it is important that the bytes be ordered in the same way for everyone. Otherwise, the signatures will never match.

To canonicalize a JSON object, use [RFC8785].
**TODO**: Matrix currently doesn't use RFC8785, but it should (or similar).

8. Event Redactions

All fields at the top level except the following are stripped from the event:

* `type`
* `room_id`
* `sender`
* `state_key`
* `content`
* `origin_server_ts`
* `hashes`
* `signatures`
* `prev_events`
* `auth_events`
* `hub_server`

Additionally, some event types retain specific fields under the event's content. All other fields are stripped.

* `m.room.create` retains all fields in content.
* `m.room.member` retains membership.
* `m.room.join_rules` retains `join_rule`.
* `m.room.power_levels` retains `ban`, `events`, `events_default`, `kick`, `redact`, `state_default`, `users`, `users_default`, and `invite`.
* `m.room.history_visibility` retains `history_visibility`.

9. Hashes

An event is covered by two hashes: a content hash and a reference hash. The content hash covers the unredacted event to ensure it was not modified in transit. The reference hash covers the essential fields of the event, including content hashes, and serves as the event's ID.
9.1. Content Hash Calculation

TODO: Describe what this protects, and why it matters.

1. Remove any existing signatures field.
   1. If calculating an LPDU's (Section 3.5.1) content hash, remove any existing hashes field as well.
   2. If not calculating an LPDU's content hash, remove any existing fields under hashes except for lpdu.

2. Encode the object using canonical JSON.

3. Hash the resulting bytes with SHA-256 [RFC6234].

4. Encode the hash using unpadded base64 (Section 10).

9.2. Reference Hash Calculation

TODO: Describe what this protects, and why it matters.

1. Redact the event.

2. Remove signatures field.

3. Encode the object using canonical JSON.

4. Hash the resulting bytes with SHA-256 [RFC6234].

5. Encode the hash using URL-safe unpadded base64 (Section 10).

10. Unpadded Base64

Throughout this document, "unpadded base64" is used to represent binary values as strings. Base64 is as specified by Section 4 of [RFC4648], and "unpadded base64" simply removes any = padding from the resulting string.

Implementations SHOULD accept input with or without padding on base64 values, where possible.

Section 5 of [RFC4648] describes URL-safe base64. The same changes are adopted here. Namely, the 62nd and 63rd characters are replaced with - and _ respectively. The unpadded behaviour is as described above.

11. Hub Selection

TODO: Describe impacts of hub transfers
The hub server for a room is the server denoted by the sender of the m.room.create event (Section 3.5.3.1). Note that this is effectively the same as the server name contained in the room ID (Section 3.2) currently, however is deliberately not defined as such. In a future scenario where hub transfers are possible, the room ID does not change when the hub server does.

11.1. Hub Transfers

TODO: This section, if we want a single canonical hub in the room. Some expected problems in this area are: who signs the transfer event? who sends the transfer event? how does a transfer start?

TODO: Likely to be done with an m.room.hub state event in the room, where the "current hub" is either the sender of m.room.hub or m.room.create if no hub state event is present. Auth rules would govern what makes for a legal m.room.hub event.

TODO: [I-D.kohbrok-mimi-portability] may be of help here.

12. Transport

This document specifies a wire transport which uses JSON [RFC8259] over HTTPS [RFC9110]. Servers MUST support a minimum of HTTP/2 [RFC9113] and TLS 1.3 [RFC8446].

TODO: This transport doesn't scale, and doesn't use RESTful endpoints. This example transport is heavily inspired by Matrix's existing Server-Server API, largely acting as a starting point for testing interoperability of the access control semantics. A better option might be gRPC, which might change how events are structured but keep the overall semantics the same. [I-D.rosenberg-mimi-protocol] might have ideas here as well for REST APIs.

12.1. TLS Certificates

Servers MUST provide a TLS certificate signed by a known Certificate Authority. Requesting servers are ultimately responsible for the Certificate Authorities they place trust in, however servers SHOULD trust authorities which would commonly be trusted by an operating system or web browser.

12.2. API Standards

12.2.1. Requests and Responses

All HTTP POST and PUT endpoints require the sending server to supply a (potentially empty) JSON object as the request body. Requesting
servers SHOULD supply a Content-Type header of application/json for such requests.

All endpoints which require a server to respond with a JSON object MUST include a Content-Type header of application/json.

All JSON data, in requests or responses, MUST be encoded using UTF-8 [RFC3629].

All endpoints in this document do not support trailing slashes on them. When such a request is encountered, it MUST be handled as an unknown endpoint (Section 12.2.3). Examples include:

*https://example.org/_matrix/path* - valid.

*https://example.org/_matrix/path/* - unknown/invalid.

*https://example.org//_matrix/path* - unknown/invalid (domain also can't have a trailing slash).

*https://example.org//_matrix/path/* - doubly unknown/invalid.

Servers (both hub and participants) MUST implement all endpoints unless otherwise specified.

Most endpoints have a version number as part of the path. This version number is that endpoint's version, allowing for breaking changes to be made to the schema of that endpoint. For clarity, the version number is not representative of an API version.

12.2.2. Errors

All errors are represented by an error code defined by this document and an accompanied HTTP status code. It is possible for a HTTP status code to map to multiple error codes, and it's possible for an error code to map to multiple HTTP status codes.

When a server is returning an error to a caller, it MUST use the most appropriate error response defined by the endpoint. If no appropriate error response is specified, the server SHOULD use M_UNKNOWN as the error code and 500 Internal Server Error as the HTTP status code.

Errors are represented as JSON objects, requiring a Content-Type: application/json response header:

```json
{
  "errcode": "M_UNKNOWN",
  "error": "Something went wrong."
}
```
errcode is required and denotes the error code. error is an optional human-readable description of the error. error can be as precise or vague as the responding server desires - the strings in this document are suggestions.

Some common error codes are:

*M_UNKNOWN - An unknown error has occurred.

*M_FORBIDDEN - The caller is not permitted to access the resource. For example, trying to join a room the user does not have an invite for.

*M_NOT_JSON - The request did not contain valid JSON. Must be accompanied by a 400 Bad Request HTTP status code.

*M_BAD_JSON - The request did contain valid JSON, but it was missing required keys or was malformed in another way. Must be accompanied by a 400 Bad Request HTTP status code.

*M_LIMIT_EXCEEDED - Too many requests have been sent. The caller should wait before trying the request again.

*M_TOO_LARGE - The request was too large for the receiver to handle.

12.2.3. Unsupported Endpoints

If a server receives a request for an unsupported or otherwise unknown endpoint, the server MUST respond with an HTTP 404 Not Found status code and M_UNRECOGNIZED error code. If the request was for a known endpoint, but wrong HTTP method, a 405 Method Not Allowed HTTP status code and M_UNRECOGNIZED error code (Section 12.2.2).

12.2.4. Malformed Requests

If a server is expecting JSON in the request body but receives something else, it MUST respond with an HTTP status code of 400 Bad Request and error code M_NOT_JSON (Section 12.2.2). If the request contains JSON, and is for a known endpoint, but otherwise missing required keys or is malformed, the server MUST respond with an HTTP status code of 400 Bad Request and error code M_BAD_JSON (Section 12.2.2). Where possible, error for M_BAD_JSON should describe the missing keys or other parsing error.

12.2.5. Transaction Identifiers

Where endpoints use HTTP PUT, it is typical for a "transaction ID" to be specified in the path parameters. This transaction ID MUST ONLY be used for making requests idempotent - if a server receives two (or
more) requests with the same transaction ID, it **MUST** return the same response for each and only process the request body once. It is assumed that requests using the same transaction ID also contain the same request body between calls.

A transaction ID only needs to be unique per-endpoint and per-sending server. A server’s transaction IDs do not affect requests made by other servers or made to other endpoints by the same server.

### 12.2.6. Rate Limiting

Servers **SHOULD** implement rate limiting semantics to reduce the risk of being overloaded. Endpoints which support being rate limited are annotated in this document.

If a rate limit is encountered, the server **MUST** respond with an HTTP 429 Too Many Requests status code and M_LIMIT_EXCEEDED error code ([Section 12.2.2](#)). If applicable, the server should additionally include a retry_after_ms integer field on the error response to denote how long the caller should wait before retrying, in milliseconds.

```json
{
  "errcode": "M_LIMIT_EXCEEDED",
  "error": "Too many requests. Try again later."
  "retry_after_ms": 10254
}
```

The exact rate limit mechanics are left as an implementation detail. A potential approach may be to prevent repeated requests for the same resource at a high rate and ensuring a remote server does not request more than a defined number of resources at a time.

### 12.3. Resolving Server Names

Before making an API request, the caller **MUST** resolve a server name ([Section 3.1](#)) to an IP address and port, suitable for HTTPS ([RFC9110](#)) traffic.

A server **MAY** change the IP/port combination used for API endpoints using SRV DNS records ([RFC2782](#)). Servers **MAY** additionally change which TLS certificate is presented by using .well-known delegation.

 well-known delegation (step 3 below) is recommended for its ease of configuration over SRV DNS records.

The target server **MUST** present a valid TLS certificate ([Section 12.1](#)) for the name described in each step. Similarly, the requesting server **MUST** use an HTTP Host header matching the description in each step.
Server developers should note that many of the DNS requirements for the steps below are typically handled by the software language or library implicitly. It is rare that a DNS A record needs to be resolved manually, for example.

Per Section 3.1, a server name consists of <hostname>[:<port>]. The steps to convert that server name to an IP address and port are:

1. If <hostname> has an explicit <port> is present, resolve <hostname> to an IP address using CNAME [RFC1034] [RFC2181], AAAA [RFC3596], or A [RFC1035] DNS records. Requests are made to the resolved IP address and port number.

   TLS certificate: <hostname> (always without port)

   Host header: <hostname>:<port>

2. A regular (non-Matrix) HTTPS request is made to https://<hostname>/._matrix._tcp./.well-known/matrix/server, expecting the schema defined by Section 12.3.1. If the response is invalid (bad/not JSON, missing properties, non-200 response, etc), skip to Step 4. If the response is valid, the m.server property is parsed as <delegated_hostname>[:<delegated_port>].

   1. If <delegated_hostname> has an explicit <delegated_port> is present, resolve <delegated_hostname> to an IP address using CNAME, AAAA, or A DNS records. Requests are made to the resolved IP address and port number.

      TLS certificate: <delegated_hostname> (always without port)

      Host header: <delegated_hostname>:<delegated_port>

   2. An SRV DNS record is resolved for _matrix._tcp.<delegated_hostname>. This may result in another hostname and port to be resolved using AAAA or A DNS records. Requests are made to the resolved IP address and port number.

      TLS certificate: <delegated_hostname>

      Host header: <delegated_hostname> (without port)

3. If no SRV record is found, an IP address is resolved for <delegated_hostname> is resolved using CNAME, AAAA, or A DNS records. Requests are made to the resolved IP address with port number 8448.

   TLS certificate: <delegated_hostname>
3. If the .well-known call from Step 3 resulted in an invalid response, an SRV DNS record is resolved for `_matrix._tcp.<hostname>`. This may result in another hostname and port to be resolved using AAAA or A DNS records. Requests are made to the resolved IP address and port number.

   TLS certificate: <hostname> (always without port)

   Host header: <hostname> (without port)

4. If the .well-known call from Step 3 resulted in an invalid response, and the SRV record from Step 4 was not found, and IP address is resolved using CNAME, AAAA, or A DNS records. Requests are made to the resolved IP address and port 8448.

   TLS certificate: <hostname> (always without port)

   Host header: <hostname> (without port)

We require `[[delegated_]hostname]` rather than `<srv_hostname>` in Steps 3.3 and 4 for the following reasons:

1. DNS is largely insecure (not all domains use DNSSEC [RFC9364]), so the target of the SRV record must prove it is a valid delegate/target for `<[[delegated_]hostname]]` via TLS.

2. Section 6.2.1 of [RFC6125] recommends this approach, and is consistent with other applications which use SRV records (such as Section 13.7.2.1 of [RFC6120]/XMPP).

Server implementations and owners should additionally note that the target of a SRV record **MUST NOT** be a CNAME, as per RFC 2782 [RFC2782]:

   the name **MUST NOT** be an alias (in the sense of RFC 1034 or RFC 2181)

[RFC1034] [RFC2181]

12.3.1. GET /.well-known/matrix/server

Used by the server name resolution approach to determine a delegated hostname for a given server. 30x HTTP redirection **MUST** be followed, though loops **SHOULD** be avoided. Normal X.509 certificate validation is applied to this endpoint (not the specific validation required by the server name resolution steps) [RFC5280].

This endpoint **MAY** be implemented by servers (it is optional).
Rate-limited: No.

Authentication required: No.

This HTTP endpoint does not specify any request parameters or body.

200 OK response:

```
{
  "m.server": "delegated.example.org:8448"
}
```

m.server is a required response field. Responses SHOULD have a Content-Type HTTP header of application/json, however servers parsing the response should assume that the body is JSON regardless of Content-Type header. Failures in parsing the JSON or otherwise invalid data that prevents parsing MUST NOT result in discovery failure. Instead, the caller is expected to move on to the next step of the name resolution approach.

Cache control headers SHOULD be respected on a 200 OK response. Callers SHOULD impose a maximum cache time of 48 hours, regardless of cache control headers. A default of 24 hours SHOULD be used when no cache control headers are present.

Error responses (non-200) SHOULD be cached for no longer than 1 hour. Callers SHOULD exponentially back off (to a defined limit) upon receiving repeated error responses.

### 12.4. Request Authentication

Most endpoints in this document require authentication to prove which server is making the request. This is done using public key digital signatures.

The request method, target, and body are represented as a JSON object, signed, and appended as an HTTP Authorization header with an auth scheme of X-Matrix.

The object to be signed is:

```
{
  "method": "GET",
  "uri": "/path/to/endpoint?with_qs=true",
  "origin": "requesting.server.name.example.org",
  "destination": "target.server.name.example.org",
  "content": {"json_request_body": true}
}
```
method is the HTTP request method, capitalized. uri is the full request path, beginning with the leading slash and containing the query string (if present). uri does not contain the https: scheme or hostname.

**TODO:** Define an ordering algorithm for the query string (if we need to?).

origin and destination are the sender and receiver server names (Section 3.1), respectively.

content is the JSON-encoded request body. When a request doesn't contain a body, such as in GET requests, use an empty JSON object. That object is then signed (Section 6.2) by the requesting server. The resulting signature is appended as an Authentication HTTP header on the request:

```
GET /path/to/endpoint?with_qs=true
Authorization: X-Matrix origin="requesting.server.name.example.org",
   destination="target.server.name.example.org",
   key="ed25519:0",
   sig="<unpadded base64 encoded signature>"
Content-Type: application/json
{"json_request_body": true}
```

Linebreaks within Authorization are for clarity and are non-normative.

The format of the Authorization header matches Section 11.4 of [RFC9110]. The header begins with an authorization scheme of X-Matrix, followed by one or more spaces, followed by an (unordered) comma-separated list of parameters written as name=value pairs. The names are case insensitive, though the values are. The values must be enclosed in quotes if they contain characters which are not allowed in a token, as defined by Section 5.6.2 of [RFC9110]. If a value is a valid token it may not be enclosed in quotes. Quoted values MAY contain backslash-escaped characters. When parsing the header, the recipient must unescape the characters.

The exact parameters are:

*origin - The name of the sending server. **MUST** match the origin in the signed JSON.

*destination - The name of the receiving server. **MUST** match the destination in the signed JSON.
key - The ID, including algorithm name, of the sending server's signing key used to sign the request.

signature - The unpadded base64 (Section 10) encoded signature from step 2.

Unknown parameters are ignored and MUST NOT result in authentication errors.

A receiving server validates the Authorization header by composing the JSON object represented above and checking the sender's signature (Section 6.3). Note that to comply with Section 6.3 the receiver may need to append a signatures field to the JSON object manually. All signatures MUST use an unexpired key at the time of the request (Section 12.4.1.1).

A server with multiple signing keys SHOULD include an Authorization header for each signing key.

If an endpoint requires authentication, servers MUST:

*Validate all presented Authorization headers.
*Ensure at least one Authorization header is present.

If either fails (lack of headers, or any of the headers fail validation), the request MUST be rejected with an HTTP 401 Unauthorized status code and M_FORBIDDEN error code (Section 12.2.2):

```
{
  "errcode": "M_FORBIDDEN",
  "error": "Signature error on request."
}
```

If an endpoint does not require authentication, Authorization headers are ignored entirely.

Responses from a server are authenticated using TLS and do not have additional signing requirements.

12.4.1. Retrieving Server Keys

TODO: Explain what notaries are and what they do, if we keep this section at all.

A server's signing keys are published under /_matrix/key/v2/server (Section 12.4.1.2) and can be queried through notary servers in two ways: Section 12.4.1.3 and Section 12.4.1.4. Notary servers implicitly call /_matrix/key/v2/server when queried, signing and
caching the response for some time. This allows the target server to offline without affecting their previously sent events.

The approach used here is borrowed from the Perspectives Project [PerspectivesProject], modified to cover the server's ed25519 keys and to use JSON instead of XML. The advantage of this system is it allows each server to pick which notaries it trusts, and can contact multiple notaries to corroborate the keys returned by any given notary.

Servers SHOULD attempt to contact the target server directly before using a notary server.

Note that these endpoints operate outside the context of a room: a server does not need to participate in any shared rooms to be used as a notary by another server, and does not need to use the hub as a notary.

12.4.1.1. Validity

A server's keys are only valid for a short time, denoted by valid_until_ts. Around the valid_until_ts timestamp, a server would re-fetch the server's keys to discover any changes. In the vast majority of cases, only valid_until_ts changes between requests (keys are long-lived, but validated frequently).

valid_until_ts MUST be handled as the lesser of valid_until_ts and 7 days into the future, preventing attackers from publishing long-lived keys that are unable to be revoked. Servers SHOULD use a timestamp approximately 12 hours into the future when responding with their keys.

TODO: What does it mean to require events have an origin_server_ts which is less than that of valid_until_ts? Do we reject the event, soft-fail it, or do something else? Do we only do this on the hub?

12.4.1.2. GET `/matrix/key/v2/server`

Retrieves the server's signing keys. The server can have any number of active or inactive keys at a time, but SHOULD have at least 1 active key at all times.

Rate-limited: No.

Authentication required: No.

This HTTP endpoint does not specify any request parameters or body.

200 OK response:
server_name **MUST** be the name of the server ([Section 3.1](#)) which is returning the keys.

valid_until_ts is the integer timestamp (milliseconds since Unix epoch) for when the server's keys should be re-fetched. See [Section 12.4.1.1](#).

m.linearized is an optional boolean, but **SHOULD** be set to true. Semantics for false and not being present apply to contexts outside of this document.

verify_keys are the current signing keys for the server, keyed by key ID ([Section 6](#)). The object value for each key ID under verify_keys is simply the key, consisting of the unpadded base64 encoded public key matching that algorithm and version.

old_verify_keys are similar to verify_keys, but have an additional required expired_ts property to denote when the key ceased usage. This overrides valid_until_ts for the purposes of [Section 12.4.1.1](#) at an individual key level.

**TODO**: What about events sent with old_verify_keys?

For request authentication ([Section 12.4](#)), only keys listed under verify_keys are honoured. If another key is referenced by the Authorization headers, the request fails authentication.
Notaries SHOULD cache a 200 OK response for half of its lifetime to avoid serving stale values. Responding servers SHOULD avoid returning responses which expire in less than an hour to avoid repeated requests. Requesting servers SHOULD limit how frequently they query for keys to avoid flooding a server with requests.

If the server fails to respond to this request, notaries SHOULD continue to return the last response they received from the server so that the signatures of old events can still be checked, even if that response is no longer considered valid (Section 12.4.1.1).

Servers are capable of rotating their keys without populating old_verify_keys, though this can cause reliability issues if other servers don't see both keys. Notaries SHOULD cache responses with distinct key IDs indefinitely. For example, if a server has ed25519:0 and ed25519:1 on its first response, and a later response returns ed25519:1 and ed25519:2, the notary should cache both responses. This gives servers an ability to validate ed25519:0 for old events in a room.

12.4.1.3. GET /_matrix/key/v2/query/:serverName

This is one of two endpoints for querying a server's keys through another server. The notary (receiving) server will attempt to refresh its cached copy of the target server's keys through /_matrix/key/v2/server, falling back to any cached values if needed.

Rate-limited: No.

Authentication required: No.

Path parameters:

*:serverName - the target server's name (Section 3.1) to retrieve keys for.

Query parameters:

*minimum_valid_until_ts (integer; optional) - The time in milliseconds since the Unix epoch the target server's keys will need to be valid until to be useful to the caller. If not specified the notary server's current time will be used.

Request body: None applicable.

200 OK response:
server_keys is the array of keys (see Section 12.4.1.2 response format) for the target server. If the target server could not be reached and the notary has no cached keys, this array is empty. If the keys do not meet minimum_valid_until_ts per Section 12.4.1.1, they are not included.

The notary server **MUST** sign each key returned in server_keys by at least one of its own signing keys. The calling server **MUST** validate all signatures on the objects.

### 12.4.1.4. POST /_matrix/key/v2/query

A bulk version of /_matrix/key/v2/query/:serverName (Section 12.4.1.3). The same behaviour applies to this endpoint.

- Rate-limited: No.
- Authentication required: No.

Path parameters: None applicable.

Query parameters: None applicable.

Request body:

```json
{
    "server_keys": {
        "example.org": {
            "ed25519:0": {
                "minimum_valid_until_ts": 1686783382189
            }
        }
    }
}
```

server_keys is required and is the search criteria. The object value is first keyed by server name which maps to another object keyed by Key ID, mapping to the specific criteria. If no key IDs are given in the request, all of the server's known keys are queried. If no servers are given in the request, the response **MUST** contain an empty server_keys array.

minimum_valid_until_ts holds the same meaning as in Section 12.4.1.3.
200 OK response:

Same as Section 12.4.1.3 with the following added detail:

Responding servers SHOULD only return signed key objects for the key IDs requested by the caller, however servers MAY respond with more keys than requested. The caller is expected to filter the response if needed.

12.5. Sending Events

Events accepted into the room by a hub server must be sent to all other servers in that room. Similarly, participant servers need a way to send partial events through the hub server, as mentioned by Section 3.5.1.

A single endpoint is used for all rooms on either server, and can contain both fully-formed PDUs (Section 3.5) or Linearized PDUs (partial events; Section 3.5.1) depending on the server's role in the applicable room.

A typical event send path will be:

PUT /send/:txnId is shorthand for Section 12.5.1.
Hubs which generate events would skip to the point where they create a fully-formed PDU and send it out to all other participants.

When a hub is broadcasting events to participant servers, it **MUST** include the following targets:

* The server implied by the sender for a kick or ban m.room.member (Section 3.5.3.3) event, up to the point of that kick or ban.

* All servers which have at least 1 user which is joined to the room.

12.5.1. PUT /_matrix/federation/v2/send/:txnId

Sends (L)PDUs (Section 3.5, Section 3.5.1) to another server. The sending server **MUST** wait for a 200 OK response from the receiver before sending another request with a different :txnId.

**Implementation note**: Currently this endpoint doesn't actually exist. Use PUT /_matrix/federation/unstable/org.matrix.i-d.ralston-mimi-linearized-matrix.02/send/:txnId when testing against other Linearized Matrix implementations. This string may be updated later to account for breaking changes.

**TODO**: Remove implementation notes.

**Rate-limited**: No.

**Authentication required**: Yes.

Path parameters:

* :txnId - the transaction ID (Section 12.2.5) for the request.

Query parameters: None applicable.

Request body:

```json
{
  "edus": [
    {/* EDU */}
  ],
  "pdus": [
    {/* Either an LPDU or PDU */}
  ]
}
```

edus are the Ephemeral Data Units (Section 12.9) to send. If no EDUs are being sent, this field **MAY** be excluded from the request body. There **MUST NOT** be more than 100 entries in edus.
PDU are the events/PDUs (Section 3.5) and LPDUs (Section 3.5.1) to send to the server. Whether it's an LPDU or PDU depends on the sending server's role in that room: if they are a non-hub server, it will be an LPDU. There MUST NOT be more than 50 entries in pdus.

Each event in the pdus array gets processed as such:

1. Identify the room ID for the event. The exact format of the event can differ between room versions, however currently this would be done by extracting the room_id property.
   1. If that room ID is invalid/not found, the event is rejected.
   2. If the server is not participating in the room, the event is dropped/skipped.
2. If the event is an LPDU and the receiving server is the hub, the additional PDU fields are appended before continuing.
3. If the event is an LPDU and the receiving server is not the hub, the event is dropped/skipped.
4. The checks defined by Section 5.1 are performed.
5. If the event still hasn't been dropped/rejected, it is appended to the room. For participant servers, this may mean it's queued for sending to local clients.

Server implementation authors should note that these steps can be condensed, but are expanded here for specification purposes. For example, an LPDU's signature can/will fail without ever needing to append the PDU fields first - the server can skip some extra work this way.

200 OK response:

```json
{
   "failed_pdus": {
      "$eventid": {
         "error": "Invalid event format"
      },
      "$eventid": {
         "error": "@alice:example.org cannot send m.room.power_levels"
      }
   }
}
```
The receiving server **MUST NOT** send a 200 OK response until all events have been processed. Servers **SHOULD NOT** block responding to this endpoint on sending accepted events to local clients or other participant servers, as doing so could lead to a lengthy backlog of events waiting to be sent.

Sending servers **SHOULD** apply/expect a timeout and retry the exact same request with the same transaction ID until they see a 200 OK response. If the sending server attempts to send a different transaction ID from the one already in flight, the receiving server **MUST** respond with a 400 Bad Request HTTP status code and M_BAD_STATE error code ([Section 12.2.2](#)). Receiving servers **SHOULD** continue processing requests to this endpoint event after the sender has disconnected/timed out, but **SHOULD NOT** process the request multiple times due to the transaction ID ([Section 12.2.5](#)).

failed_pdu is an object mapping event ID ([Section 3.5](#)) to error string. Event IDs are based upon the received object, not the final/complete object. For example, if an LPDU is sent, gets its PDU fields appended, and fails event authorization, then the error would be for the event ID of the LPDU, not the fully-formed PDU. This is to allow the sender to correlate what they sent with errors.

The object for each event ID **MUST** contain an error string field, representing the human-readable reason for an event being rejected.

Events which are dropped/ignored or accepted do **not** appear in failed_pdu.

**TODO:** Should we also return fully-formed PDUs for the LPDUs we received?

### 12.6. Event and State APIs

When a participant in the room is missing an event, or otherwise needs a new copy of it, it can retrieve that event from the hub server. Similar mechanics apply for getting state events, current state of a room, and backfilling scrollbar in a room.

All servers are required to implement all endpoints ([Section 12.2.1](#)), however only hub servers are guaranteed to have the full history/state for a room. While other participant servers might have history, they **SHOULD NOT** be contacted due to the high likelihood of a Not Found-style error.

#### 12.6.1. GET /_matrix/federation/v2/event/:eventId

Retrieves a single event.
Implementation note: Currently this endpoint doesn't actually exist. Use GET /matrix/federation/unstable/org.matrix.i-d.ralston-mimi-linearized-matrix.02/event/:eventId when testing against other Linearized Matrix implementations. This string may be updated later to account for breaking changes.

TODO: Remove implementation notes.

Rate-limited: Yes.

Authentication required: Yes.

Path parameters:

*:eventId - the event ID (Section 3.5) to retrieve. Note that event IDs are typically reference hashes (Section 9.2) of the event itself, which includes the room ID. This makes event IDs globally unique.

Query parameters: None applicable.

Request body: None applicable.

200 OK response:

```json
{
  /* the event */
}
```

The response body is simply the event (Section 3.5) itself, if the requesting server has reasonable visibility of the event (Section 3.5.3.5.1). When the server can see an event but not the contents, the event is served redacted (Section 8) instead.

If the event isn't known to the server, or the requesting server has no reason to know that the event even exists, a 404 Not Found HTTP status code and M_NOT_FOUND error code (Section 12.2.2) is returned.

The returned event MUST be checked before being used by the requesting server (Section 5.1). This endpoint MUST NOT return LPDUs (Section 3.5.1), instead treating such events as though they didn't exist.

12.6.2. GET /matrix/federation/v1/state/:roomId

Retrieves a snapshot of the room state (Section 3.5.2) at the given event. This is typically most useful when a participant server prefers to store minimal information about the room, but still needs to offer context to its clients.
Rate-limited: Yes.

Authentication required: Yes.

Path parameters:

*:roomId - the room ID (Section 3.2) to retrieve state in.

Query parameters:

*event_id (string; required) - The event ID (Section 3.5) to retrieve state at.

Request body: None applicable.

200 OK response:

```json
{
  "auth_chain": [
    {/* event */}
  ],
  "pdus": [
    {/* event */}
  ]
}
```

The returned room state is in two parts: the pdus, consisting of the events which represent "current state" (Section 3.5.2) prior to considering state changes induced by the event in the original request, and auth_chain, consisting of the events which make up the auth_events (Section 5.2.1) for the pdus and the auth_events of those events, recursively.

The auth_chain will eventually stop recursing when it reaches the m.room.create event, as it cannot have any auth_events.

TODO: Do we actually need to recurse auth events to get the full auth chain here? What are participant servers expected to do with this information? (Do they even care about it?)

For example, if the requested event ID was an m.room.power_levels event, the returned state would be as if the new power levels were not applied.

Both auth_chain and pdus contain event objects (Section 3.5).

If the requesting server does not have reasonable visibility on the room (Section 3.5.3.5.1), or either the room ID or event ID don't exist, a 404 Not Found HTTP status code and M_NOT_FOUND error code
(Section 12.2.2) is returned. The same error is returned if the event ID doesn't exist in the requested room ID.

Note that the requesting server will generally always have visibility of the auth_chain and pdu events, but may not be able to see their contents. In this case, they are redacted (Section 8) before being served.

The returned events MUST be checked before being used by the requesting server (Section 5.1). This endpoint MUST NOT return LPDUs (Section 3.5.1), instead treating such events as though they didn't exist.

If the receiving server is not the hub server for the room ID, an HTTP status code of 400 Bad Request and error code M_WRONG_SERVER (Section 12.2.2) is returned.

12.6.3. GET /_matrix/federation/v1/state_ids/:roomId

This performs the same function as Section 12.6.2 but returns just the event IDs instead.

Rate-limited: Yes.

Authentication required: Yes.

Path parameters:

*:roomId - the room ID (Section 3.2) to retrieve state in.

Query parameters:

*event_id (string; required) - The event ID (Section 3.5) to retrieve state at.

Request body: None applicable.

200 OK response:

```{  
"auth_chain_ids": ["$event1", "$event2"],  
"pdu_ids": ["$event3", "$event4"]
}
```

See Section 12.6.2 for behaviour. Note that auth_chain becomes auth_chain_ids when using this endpoint, and pdus becomes pdu_ids.
12.6.4. GET /_matrix/federation/v2/backfill/:roomId

Retrieves a sliding window history of previous events in a given room.

Implementation note: Currently this endpoint doesn't actually exist. Use GET /_matrix/federation/unstable/org.matrix.i-d.ralston-mimi-linearized-matrix.02/backfill/:roomId when testing against other Linearized Matrix implementations. This string may be updated later to account for breaking changes.

TODO: Remove implementation notes.

Rate-limited: Yes.

Authentication required: Yes.

Path parameters:

*:roomId - the room ID (Section 3.2) to retrieve events from.

Query parameters:

*v* (string; required) - The event ID (Section 3.5) to start backfilling from.

*limit* (integer; required) - The maximum number of events to return, including v.

Request body: None applicable.

200 OK response:

```
{
    "pdus": [
        { /* event */ }
    ]
}
```

The number of returned pdus **MUST NOT** exceed the limit provided by the caller. limit **SHOULD** have a maximum value imposed by the receiving server. pdus contains the events (Section 3.5) preceeding the requested event ID (v), including v. pdus is ordered from oldest to newest.

If the requesting server does not have reasonable visibility on the room (Section 3.5.3.5.1), or either the room ID or event ID don't exist, a 404 Not Found HTTP status code and M_NOT_FOUND error code (Section 12.2.2) is returned. The same error is returned if the event ID doesn't exist in the requested room ID.
If the requesting server does have visibility on the returned events, but not their contents, they are redacted (Section 8) before being served.

The returned events **MUST** be checked before being used by the requesting server (Section 5.1). This endpoint **MUST NOT** return LPDUs (Section 3.5.1), instead treating such events as though they didn't exist.

### 12.7. Room Membership

When a server is already participating in a room, it can simply send m.room.member (Section 3.5.3.3) events with the /send API (Section 12.5.1) to other servers/the hub directly. When a server is not already participating however, it needs to be welcomed in by the hub server.

A typical invite flow would be:
POST /invite is shorthand for Section 12.7.2.1. Similarly, GET /make_join is Section 12.7.3.1 and POST /send_join is Section 12.7.3.2.

If the user decided to reject the invite, the TargetServer would use GET /make_leave (Section 12.7.2.2.1) and POST /send_leave (Section 12.7.2.2.2) instead of make/send_join.
12.7.1. Make and Send Handshake

When a server is already participating in a room, it can use m.room.member (Section 3.5.3.3) events and the /send API (Section 12.5.1) to directly change membership. When the server is not already involved in the room, such as when being invited for the first time, the server needs to "make" an event and "send" it through the hub server to append it to the room.

The different processes which use this handshake are:

* Rejecting Invites (Section 12.7.2.2)

* Joins (Section 12.7.3)

* Knocks (Section 12.7.4)

The "make" portion of the endpoints take the shape of GET /_matrix/federation/v1/make_CHANGE/:roomId/:userId, where CHANGE is leave, join, or knock (respective to the list above). This endpoint will return a partial LPDU (Section 3.5.1) which needs to be turned into a full LPDU and signed before being sent using POST /_matrix/federation/v3/send_CHANGE/:txnId.

The flow for this handshake appears as such:
Note that the send_CHANGE step re-checks the event against the auth rules: any amount of time could have passed between the make_CHANGE and send_CHANGE calls.

**TODO:** Describe how the external server is meant to find the hub. Invites work by (usually) trying to contact the server which sent the invite, but knocking is a guess.

### 12.7.2. Invites

When inviting a user belonging to a server already in the room, senders **SHOULD** use m.room.member (**Section 3.5.3.3**) events and the /
send API (Section 12.5.1). This section's endpoints SHOULD only be used when the target server is not participating in the room already.

Note that being invited does not count as the server "participating" in the room. This can mean that while a server has a user with a pending invite in the room, this section's endpoints are needed to send additional invites to other users on the same server.

The full invite sequence is:
POST /invite is shorthand for Section 12.7.2.1.

What causes a user to be considered "ineligible" for an invite is left as an implementation detail. See Section 13 and Section 14 for suggestions on handling user-level privacy controls and spam invites.
12.7.2.1. POST /_matrix/federation/v3/invite/:txnId

Sends an invite event to a server. If the sender is a participant server, the receiving server (the hub) will convert the contained LPDU (Section 3.5.1) to a fully-formed event (Section 3.5) before sending that event to the intended server.

**Implementation note:** Currently this endpoint doesn't actually exist. Use POST /_matrix/federation/unstable/org.matrix.i-d.ralston-mimi-linearized-matrix.02/invite/:txnId when testing against other Linearized Matrix implementations. This string may be updated later to account for breaking changes.

**TODO:** Remove implementation notes.

**Rate-limited:** Yes.

**Authentication required:** Yes.

Path parameters:

`:txnId` - the transaction ID (Section 12.2.5) for the request. The event ID (Section 3.5) of the contained event may be a good option as a readily-available transaction ID.

Query parameters: None applicable.

Request body:

```
{
  "event": {/* the event */},
  "invite_room_state": [/* stripped state events */],
  "room_version": "I.1"
}
```

invite_room_state are the stripped state events (Section 3.5.2.1) for the room's current state. invite_room_state **MAY** be excluded from the request body.

room_version is the room version identifier (Section 3.2.1) the room is currently using. This will be retrieved from the m.room.create (Section 3.5.3.1) state event.

event is the event (LPDU or PDU; Section 3.5) representing the invite for the user. It **MUST** meet the following criteria, in addition to the requirements of an event:

* type **MUST** be m.room.member.
* membership in content **MUST** be invite.
When the hub server receives a request from a participant server, it **MUST** populate the event fields before sending the event to the intended recipient. This means running the event through the normal event authorization steps (**Section 5.2**). If the invite is not allowed under the auth rules, the server responds with a 403 Forbidden HTTP status code and M_FORBIDDEN error code (**Section 12.2.2**).

The intended recipient of the invite can be identified by the state_key on the event.

If the invite event is valid, the hub server sends its own POST /_matrix/federation/v3/invite/:txnId request to the target server (if the target server is not itself) with the fully-formed event. The transaction ID does not need to be the same as the original inbound request.

All responses from the target server **SHOULD** be proxied verbatim to the original requesting server through the hub. The hub **SHOULD** discard what appears to be excess data before sending a response to the requesting server, such as extra or large fields. If the target server does not respond with JSON, an error response (**Section 12.2.2**) **SHOULD** be sent by the hub instead.

The target server then ensures it can support the room version. If it can't, it responds with an HTTP status code of 400 Bad Request and error code of M_INCOMPATIBLE_ROOM_VERSION (**Section 12.2.2**).

Then, the target server runs any implementation-specific checks as needed, such as those implied by **Section 13** and **Section 14**, rejecting/erroring the request as needed.

Finally, the target server signs the event and returns it to the hub. The hub server appends this signed event to the room and sends it out to all participants in the room. The signed event is additionally returned to the originating participant server, though it also receives the event through the /send API (**Section 12.5.1**).

200 OK response:

```
{
  "pdu": {/* signed fully-formed event */}
}
```

Note that by the time a response is received, the event is signed 2-3 times:

1. The LPDU signature from the participant server (**Section 3.5.1**).
2. The hub's signature on the PDU (**Section 3.5**).
3. The target server's signature on the PDU.

These signatures are to satisfy the auth rules (Section 5.2).

**TODO**: Do we ever validate the target server's signature? Do we need to?

### 12.7.2.2. Rejecting Invites and Leaves

Rejecting an invite is done by making a membership transition of invite to leave through the user's `m.room.member` (Section 3.5.3.3) event. The membership event **should** be sent directly when it can and use the "make and send" handshake (Section 12.7.1) described here otherwise.

This same approach is additionally used to retract a knock (Section 12.7.4).

#### 12.7.2.2.1. GET `/matrix/federation/v1/make_leave/:roomId/:userId`

Requests an event template from the hub server for a room. The requesting server will have already been checked to ensure it supports the room version as part of the invite process prior to making a call to this endpoint.

**Rate-limited**: Yes.

**Authentication required**: Yes.

Path parameters:

*:roomId - the room ID (Section 3.2) to get a template for.

*:userId - the user ID (Section 3.3) attempting to leave.

Query parameters: None applicable.

Request body: None applicable.

200 OK response:

```json
{
  "event": {/* partial LPDU */},
  "room_version": "I.1"
}
```

The response body's event **must** be a partial LPDU (Section 3.5.1) with at least the following fields:

* `type` of `m.room.member`. 
The sending server SHOULD remove all other fields before using the event in a send_leave (Section 12.7.2.2).

If the receiving server is not the hub server for the room ID, an HTTP status code of 400 Bad Request and error code M_WRONG_SERVER (Section 12.2.2) is returned. If the room ID is not known, 404 Not Found is used as an HTTP status code and M_NOT_FOUND as an error code (Section 12.2.2).

If the user does not have permission to leave under the auth rules (Section 5.2), a 403 Forbidden HTTP status code is returned alongside an error code of M_FORBIDDEN (Section 12.2.2). For example, if the user does not have a pending invite, is not a member of the room, or is banned.

If the sending server does not recognize the returned room_version, it SHOULD NOT attempt to populate the template or use the send_leave (Section 12.7.2.2) endpoint.

### 12.7.2.2.2. POST /_matrix/federation/v3/send_leave/:txnId

Sends a leave membership event to the room through a hub server.

**Implementation note**: Currently this endpoint doesn't actually exist. Use POST /_matrix/federation/unstable/org.matrix.i-d.ralston-mimi-linearized-matrix.02/send_leave/:txnId when testing against other Linearized Matrix implementations. This string may be updated later to account for breaking changes.

**TODO**: Remove implementation notes.

**Rate-limited**: Yes.

**Authentication required**: Yes.

**Path parameters**:

*:txnId - the transaction ID (Section 12.2.5) for the request. The event ID (Section 3.5) of the contained event may be a good option as a readily-available transaction ID.

**Query parameters**: None applicable.

**Request body**: 

*state_key of :userId from the path parameters.*

*sender of :userId from the path parameters.*

*content of {"membership": "leave"}.*
The errors responses from /make_leave (Section 12.7.2.2.1) are copied here. Servers should note that room state MAY change between a /make_leave and /send_leave, potentially in a way which prevents the user from leaving the room suddenly. For example, the invited user may have been banned from the room.

12.7.3. Joins

Joins for users SHOULD be sent directly whenever possible, and otherwise use the "make and send" handshake (Section 12.7.1) approach described here.

12.7.3.1. GET /_matrix/federation/v1/make_join/:roomId/:userId

Requests an event template from the hub server for a room. This is done to ensure the requesting server supports the room's version (Section 3.2.1), as well as hint at the event format needed to participate.

Note that this endpoint is extremely similar to /make_leave (Section 12.7.2.2.1).

Rate-limited: Yes.

Authentication required: Yes.

Path parameters:

*:roomId - the room ID (Section 3.2) to get a template for.

*:userId - the user ID (Section 3.3) attempting to join.

Query parameters:

*ver (string; required; repeated) - The room versions (Section 3.2.1) the sending server supports.

Request body: None applicable.

200 OK response:
The response body **MUST** be a partial LPDU ([Section 3.5.1](#)) with at least the following fields:

* type of m.room.member.
* state_key of :userId from the path parameters.
* sender of :userId from the path parameters.
* content of {"membership": "join"}.

The sending server **SHOULD** remove all other fields before using the event in a send_join ([Section 12.7.3.2](#)).

If the receiving server is not the hub server for the room ID, an HTTP status code of 400 Bad Request and error code M_WRONG_SERVER ([Section 12.2.2](#)) is returned. If the room ID is not known, 404 Not Found is used as an HTTP status code and M_NOT_FOUND as an error code ([Section 12.2.2](#)).

If the user does not have permission to join under the auth rules ([Section 5.2](#)), a 403 Forbidden HTTP status code is returned alongside an error code of M_FORBIDDEN ([Section 12.2.2](#)).

If the room version is not one of the ver strings the sender supplied, a 400 Bad Request HTTP status code is returned alongside M_INCOMPATIBLE_ROOM_VERSION error code ([Section 12.2.2](#)).

### 12.7.3.2. POST /_matrix/federation/v3/send_join/:txnId

Sends a join membership event to the room through a hub server.

Note that this endpoint is extremely similar to /send_leave ([Section 12.7.2.2.2](#)).

**Implementation note:** Currently this endpoint doesn't actually exist. Use POST /_matrix/federation/unstable/org.matrix.i-d.ralston-mimi-linearized-matrix.02/send_join/:txnId when testing against other Linearized Matrix implementations. This string may be updated later to account for breaking changes.

**TODO:** Remove implementation notes.

**Rate-limited:** Yes.

**Authentication required:** Yes.
Path parameters:

*:txnId - the transaction ID ([Section 12.2.5](#section12.2.5)) for the request. The event ID ([Section 3.5](#section3.5)) of the contained event may be a good option as a readily-available transaction ID.

Query parameters: None applicable.

**TODO:** Incorporate faster joins work.

Request body:

```json

/* LPDU created from make_join template */
```  

200 OK response:

```json

{
  "state": /* events */,
  "auth_chain": /* events */,
  "event": /* fully-formed event */
}
```

state is the current room state, consisting of the events which represent "current state" ([Section 3.5.2](#section3.5.2)) prior to considering the membership state change. auth_chain consists of the events which make up the auth_events ([Section 5.2.1](#section5.2.1)) for the state events, and the auth_events of those events, recursively. event will be the fully-formed PDU ([Section 3.5](#section3.5)) that is sent by the hub to all other participants in the room.

The errors responses from /make_join ([Section 12.7.3.1](#section12.7.3.1)) are copied here (with the exception of M_INCOMPATIBLE_ROOM_VERSION, as the server already checked for support). Servers should note that room state MAY change between a /make_join and /send_join, potentially in a way which prevents the user from joining the room suddenly.

### 12.7.4. Knocks

To knock on a room is to request an invite to that room. It is not a join, nor is it an invite itself. "Approving" the knock is done by inviting the user, which is typically only allowed by moderators in these rooms. "Denying" the knock is done through kicking (sending a leave membership) or banning the user. If the user is kicked, they may re-send their knock.

Senders should note the reason field on m.room.member events ([Section 3.5.3.3](#section3.5.3.3)) to provide context for their knock.
To retract a knock, the sending server uses the same APIs as rejecting an invite (Section 12.7.2.2).

Where possible, knocks from users SHOULD be sent directly, otherwise using the "make and send" handshake (Section 12.7.1) approach described here.

12.7.4.1. GET /matrix/federation/v1/make_knock/:roomId/:userId

Requests an event template from the hub server for a room. This is done to ensure the requesting server supports the room's version (Section 3.2.1), as well as hint at the event format needed to participate.

Note that this endpoint is almost exactly the same as /make_join (Section 12.7.3.1).

TODO: It's so similar to make_join that we should probably just combine the two endpoints.

Rate-limited: Yes.

Authentication required: Yes.

Path parameters:

*:roomId - the room ID (Section 3.2) to get a template for.

*:userId - the user ID (Section 3.3) attempting to knock.

Query parameters:

*ver (string; required; repeated) - The room versions (Section 3.2.1) the sending server supports.

Request body: None applicable.

200 OK response:

```json
{

/* partial LPDU */
}
```

The response body MUST be a partial LPDU (Section 3.5.1) with at least the following fields:

*type of m.room.member.

*state_key of :userId from the path parameters.
*sender of :userId from the path parameters.

*content of {"membership": "knock"}.

The sending server SHOULD remove all other fields before using the event in a send_knock (Section 12.7.4.2).

If the receiving server is not the hub server for the room ID, an HTTP status code of 400 Bad Request and error code M_WRONG_SERVER (Section 12.2.2) is returned. If the room ID is not known, 404 Not Found is used as an HTTP status code and M_NOT_FOUND as an error code (Section 12.2.2).

If the user does not have permission to knock under the auth rules (Section 5.2), a 403 Forbidden HTTP status code is returned alongside an error code of M_FORBIDDEN (Section 12.2.2).

If the room version is not one of the ver strings the sender supplied, a 400 Bad Request HTTP status code is returned alongside M_INCOMPATIBLE_ROOM_VERSION error code (Section 12.2.2).

12.7.4.2. POST /_matrix/federation/v3/send_knock/:txnId

Sends a knock membership event to the room through a hub server.

Implementation note: Currently this endpoint doesn't actually exist. Use POST /_matrix/federation/unstable/org.matrix.i-d.ralston-mimi-linearized-matrix.02/send_knock/:txnId when testing against other Linearized Matrix implementations. This string may be updated later to account for breaking changes.

TODO: Remove implementation notes.

Rate-limited: Yes.

Authentication required: Yes.

Path parameters:

*:txnId - the transaction ID (Section 12.2.5) for the request. The event ID (Section 3.5) of the contained event may be a good option as a readily-available transaction ID.

Query parameters: None applicable.

Request body:

```json
{
  /* LPDU created from make_knock template */
}
```
200 OK response:

```
{
  "stripped_state": [
    /* stripped state events */
  ]
}
```

stripped_state are the stripped state events (**Section 3.5.2.1**) for the room.

The errors responses from /make_knock (**Section 12.7.4.1**) are copied here (with the exception of M_INCOMPATIBLE_ROOM_VERSION, as the server already checked for support). Servers should note that room state **MAY** change between a /make_knock and /send_knock, potentially in a way which prevents the user from knocking upon the room suddenly.

### 12.8. Content Repository

The content repository, sometimes called the "media repo", is where user-generated content is stored for referencing within an encrypted message.

**TODO**: Complete this section. We want auth/event linking from MSC3911 and MSC3916.

**TODO**: Spell out that content is images, videos, files, etc.

### 12.9. Ephemeral Data Units (EDUs)

EDUs are sent out of band from rooms and are only persisted for exactly as long as they are needed. For example, once a to-device (**Section 12.10.2**) message is delivered to a client, the server may easily be able to delete its copy of the message.

EDUs contain the following mandatory fields:

```
{
  "type": "m.room.encrypted",
  "sender": "@alice:example.org",
  "content": {
    /* type-specific content */
  }
}
```

The type is similar to an event type (**Section 3.5**) and ultimately describes the schema for the content.
sender_id is the user ID (Section 3.3) which is sending the EDU. Typically, clients will not generate EDUs directly. Instead, the server will convert a client's request into an EDU for sending to a remote server, where that server then unpacks the EDU before delivering it to local devices.

Because EDUs are not sent in the context of a room, even if an MLS Welcome message is being sent for a room, servers MUST send the EDUs directly to the target server with the send API (Section 12.5.1).

In this document, EDUs are only used for to-device messages (Section 12.10.2) and device list changes (Section 12.10.1), but could be used for read/delivery receipts, typing notifications, and more in future. This may necessitate routing EDUs through the hub rather than using full-mesh fanout.

TODO: Address EDU fanout; Document the implied missing features (receipts, typing notifs).

12.10. MLS

TODO: This section. Talk about to-device messaging, device management/querying/key claiming, etc.

There are several endpoints required by this document's MLS implementation (Section 4), largely around device management for each device's signing key, claiming key packages for those devices, and sending messages (Welcome in particular) after using a key package.

12.10.1. Device Info Publishing

When a user creates a new encryption-capable device, or removes one, a "device list update" is sent to all servers the user shares a room with. The receiving servers then determine which local clients need to be made aware of the device list change and sends the information to them. This is primarily used by this document's MLS implementation (Section 4) to indicate to other devices that either a new possible device has come online or that another needs to be removed from some MLS groups due to being deleted.

Typically, a device is created by a user when they log in to a new session. Similarly, a device is deleted/removed when they log out of that client.

The device list update takes the shape of an EDU (Section 12.9), as such:
The device objects are the same as in the response for /user/:userId/device/:deviceId (Section 12.10.1.1), indicating that either a new device was created or that information about a previous device has changed.

TODO: Matrix's m.device_list_update EDU is very different from this, and relatively complicated. Do we actually need a stream_id, like in Matrix? Do we then need the /devices endpoint?

12.10.1.1. GET /_matrix/federation/v1/user/:userId/device/:deviceId

Retrieves information about a specific device for a user. This request does not go via a hub, instead going directly to the server which owns the :userId.

**Implementation note**: Currently this endpoint doesn't actually exist. Use PUT /_matrix/federation/unstable/org.matrix.i-d.ralston-mimi-linearized-matrix.02/user/:userId/device/:deviceId when testing against other Linearized Matrix implementations. This string may be updated later to account for breaking changes.

TODO: Remove implementation notes.

**Rate-limited**: Yes.

**Authentication required**: Yes.

Path parameters:

*:userId - the user ID (Section 3.3) who owns the device.

*:deviceId - the device ID (Section 3.4) to get information about.

Query parameters: None applicable.

Request body: None applicable.

200 OK response:

```
{
  "type": "m.device_list_update",
  "sender_id": "@alice:example.org",
  "content": {
    "changed": [/* Device Objects */],
    "removed": [/* Device IDs */]
  }
}
```
Note that the response is the same object the device itself signed/created in Section 4.1.

If the user ID does not belong the receiving server, a 404 Not Found HTTP status code is returned with error code M_NOT_FOUND (Section 12.2.2). The same applies if the user ID does not exist, or the user does not have the device ID requested.

12.10.2. To-Device Messaging

To-device messaging is an ability to send information directly to another device, typically to carry MLS Welcome messages and similar. They are sent as EDUs (Section 12.9), one per recipient device and payload:

```
{
  "type": "m.direct_to_device",
  "sender": "@alice:example.org",
  "content": {
    "target": "@bob:example.org",
    "target_device": "ABCD",
    "message_type": "m.room.encrypted",
    "message": {
      /* message_type-specific schema */
    }
  }
}
```

target and target_device denote the destination user ID (Section 3.3) and device ID (Section 3.4) for that user. This EDU MUST be sent to the server denoted by the target user ID. If the target user doesn't
exist or doesn't have a device with the ID described, the receiving server drops/ignores the EDU.

See Section 4.3 for an example of a to-device message being used.

12.10.3. One Time Key Claiming

To enable two devices to communicate, they need to claim a key package (Section 4.4) for the other device. These key packages are also called "one time keys". This is done through the following endpoint.

12.10.3.1. POST /_matrix/federation/v1/user/keys/claim

Claims one time keys for devices. This request does not go via a hub, instead going directly to the server which owns the given user IDs.

Rate-limited: Yes.

Authentication required: Yes.

Path parameters: None applicable.

Query parameters: None applicable.

Request body:

```
{
  "one_time_keys": {
    "@alice:example.org": {
      "ABCD":
      "m.mls.v1.key_package.dhkemx25519-aes128gcm-sha256-ed25519"
    }
  }
}
```

one_time_keys MUST be specified and is a map of user ID (Section 3.3) to device ID (Section 3.4) to algorithm for the key package to claim. Currently the only expected algorithm is defined by Section 4.4.

Any user IDs which don't belong to the receiving server, or which don't exist, are ignored. The same applies for device IDs for which the user doesn't have.

200 OK response:
Like the request body, one_time_keys MUST be specified (but MAY be empty) and is a map of requested user ID to requested device ID to algorithm name. The value for the algorithm name is dependent on the algorithm itself. For m.mls.v1.key_package.dhkemx25519-aes128gcm-sha256-ed25519, this is an unpadded base64 (Section 10) string representing the key package itself.

Servers MUST NOT reuse a device's one time key, unless that key permits it. For example, MLS's "last resort" key MAY be used multiple times, but SHOULD only be used if no other one time keys remain for the device. Servers MUST NOT use an expired key.

Typically, the server will inform the device that a key was used so the device can upload additional keys. See Section 4.4 for further implementation-related concerns.

12.11. TODO: Remainder of Transport

TODO: This section.

Topics:

* More EDUs (typing notifications, receipts, presence)
* Query APIs (alias resolution, profiles)
* Other Encryption APIs??
* Server ACLs? (this probably should become part of the auth rules)

Notably/deliberately missing APIs are:

* get_missing_events - this is used by DAG servers only
* Public room directory
* Timestamp-to-event API
* All of 3rd party invites
12.11.1. Open Questions

*Should we include /_matrix/federation/v1/version in here? It's used by federation testers, but not really anything else.

13. User Privacy

TODO: Fully complete this section.

Messaging providers may have user-level settings to prevent unexpected or unwarranted invites, such as automatically blocking invites from non-contacts. This setting can be upheld by returning an error on POST /_matrix/federation/v3/invite/:txnId (Section 12.7.2.1), and by having the server (optionally) auto-decline any invites received directly through PUT /_matrix/federation/v2/send/:txnId (Section 12.5.1). See Section 12.7.2.2 for more information on rejecting invites.

14. Spam Prevention

TODO: Fully complete this section.

TODO: Talk about how to deal with spammy/unwanted invites.

Servers MAY temporarily or permanently block a room entirely by using the room ID. Typically, when a room becomes blocked, all local users will be removed from the room using m.room.member events with membership of leave (Section 3.5.3.3). Then, any time the server receives a request for that room ID it can reject it with an error response (Section 12.2.2).

Blocking a room does not block it from all servers, but does prevent users on a server from accessing the content within. This is primarily useful to remove a server from rooms where abusive/illegal content is shared.

15. Security Considerations

TODO: Expand upon this section.

With the combined use of MLS and server-side enforcement, the server theoretically has an ability to add a malicious device to the MLS group and receive decryptable messages. Authenticity of devices needs to be established to ensure a user's devices are actually a user's devices.
TODO: Should we bring Matrix's cross-signing here?

Servers retain the ability to control/puppet their own users due to no strong cryptographic link between the sending device and the event which gets emitted.

16. IANA Considerations

The m.* namespace likely needs formal registration in some capacity.

The I.* namespace likely needs formal registration in some capacity.

Port 8448 may need formal registration.

The SRV service name matrix may need re-registering, or a new service name assigned.

The .well-known/matrix namespace is already registered for use by The Matrix.org Foundation C.I.C.

17. References

17.1. Normative References


[I-D.ralston-mimi-terminology] Ralston, T., "MIMI Terminology", Work in Progress, Internet-Draft, draft-ralston-mimi-terminology-02, 10 July


Saint-Andre, P. and J. Hodges, "Representation and Verification of Domain-Based Application Service Identity


17.2. Informative References


Acknowledgments

Thanks to the Matrix Spec Core Team (SCT) for exploring methods of linearizing Matrix's DAG room model, especially Richard van der Hoff and Erik Johnston. Additional thanks to the Matrix core team as a whole for general review and input on this document, especially Patrick Cloke and Jim Mackenzie.

Matrix's crypto team graciously allowed their work-in-progress MLS considerations to be incorporated in this document. Thanks to Hubert Chathi for reviewing the MLS work for plausibility.

Thanks are also extended to the wider MIMI working group for continuous review and input on this document.

Authors' Addresses

Travis Ralston
The Matrix.org Foundation C.I.C.

Email: travisr@matrix.org

Matthew Hodgson
The Matrix.org Foundation C.I.C.

Email: matthew@matrix.org