Workgroup:
More Instant Messaging Interoperability
Internet-Draft:
draft-ralston-mimi-linearized-matrix-00
Published: 24 March 2023
Intended Status: Standards Track
Expires: 25 September 2023
Authors: T. Ralston
The Matrix.org Foundation C.I.C.
M. Hodgson
The Matrix.org Foundation C.I.C.

# Linearized Matrix API

## Abstract

Matrix is an existing openly specified decentralized secure communications protocol able to provide a framework for instant messaging interoperability. Matrix rooms (aka conversations) currently use a Directed Acyclic Graph (DAG) for persisting events/messages. Servers broadcast their changes to the DAG to every other server in order to create new events.

This model provides excellent decentralization characteristics and conversation history replication, but proves complex when aiming to use Matrix strictly for interoperability between today's existing messaging service providers, which often do not persist chat history serverside, and do not seek to replicate it between servers.

This document explores an API surface for Matrix which optimizes for ease of interoperability at the expense of decentralized conversation history at a per-room level. We call this API surface "Linearized Matrix".

## 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/. Subscribe at https://www.ietf.org/mailman/listinfo/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 25 September 2023.

## 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. Identifiers
4. Room Architecture
5. Event Signing
6. Membership
7. State Events API
8. Room Events API
9. Room Transfers
10. Other APIs
11. Matrix Room Version
12. Anti-Abuse and Anti-Spam
13. DAG-Compatible Event Structure

14. Security Considerations
15. IANA Considerations
16. References
  16.1. Normative References
  16.2. Informative References
Acknowledgments
Authors' Addresses

## 1. Introduction

At a high level, rooms using Linearized Matrix have a single server which owns that room. The owner can change, but will typically be the server which created the room. All other servers are known as participating servers and call the owner server to send events. The owner server is then responsible for informing all the other servers of any changes/messages in the room.

Many aspects for how Matrix works as an interoperable messaging framework is described by [I-D.ralston-mimi-matrix-framework]. This document replaces the eventual consistency model, federation API, and DAG-related features of the framework document by presenting rooms as a single, flat, array of events, without being incompatible with those same replaced components.

This document does not currently define a transport layer for the Linearized Matrix API, instead focusing its efforts on the operational aspects of a room.

## 2. Conventions and Definitions

This document additionally uses the following definitions:

* Owner Server: The server responsible for holding the room history, accepting new events, etc.

* Participant Server: Every other server. Note that a server may inherit this role even if not (currently) participating in the room.

**TODO**: Merge/add definitions from framework to here, such as "homeserver", "user", etc.

## 3. Identifiers

**TODO**: Expand upon this section.

A room ID has the format !localpart:domain, where the localpart is an opaque string and the domain provides global uniqueness. The domain does not indicate that the room exists on that server, just that it was originally created there.

A user ID has the format @localpart:domain, where the localpart is again an opaque string and the domain is where the user was created (the server owns the user account).

## 4. Room Architecture

As mentioned, rooms over Linearized Matrix have a concept of an "owning server". Typically the room owner will be the server which created the room, however ownership can shift as needed. The room owner is responsible for applying the room version semantics/access controls and distributing the changes to other applicable servers (called participant servers).

At an implementation level, it should be possible for an owning server to use a DAG if it so wishes, however for the protocol considerations a room has a single flat array to store state changes and room events.

Room state is the same as non-Linearized Matrix: represented by an event type and state key tuple which maps to a state event. "Current state" is simply the most recent instance of each event type and state key pair in the array.

To send an event into the room, each "participant server" (non-owner) asks the owner to send it to the room. The owner applies access controls to the event, following Matrix's existing access controls (power levels, bans, server_acls etc.) and then adds the event to the room's array, and sends it out to all participating servers (including the original sender, for simplicity of implementation). If the owner would like to send an event, it simply adds the event to the array (assuming such an action is valid) and broadcasts it. The owner server MUST follow the access control semantics defined by the room's current state - it MUST NOT make up its own rules. For instance, the owning server must only let Alice invite Bob to a room if Alice has permission to invite, and if Alice's server sent the invite event.

Each room additionally records which Matrix room version it is using for access control behaviours, such as Authorization Rules [MxV10AuthRules]. This is required for when rooms gain a DAG-compatible server in them. Note that this document introduces new semantics requiring a new room version.

## 5. Event Signing

Events are signed by the participant/original server to ensure the owning server is not spoofing events on behalf of another server. The exact details for how a server's signing keys are shared to other servers is left as a transport consideration, however signing keys are currently expected to be Ed25519 keys.

In the existing Matrix Federation APIs, a PDU [MxV10PDUFormat] contains an event and has several DAG-specific fields to it. When using the Linearized Matrix API, we introduce a concept of a *Linear PDU* which looks similar to a regular room event, but has all non-essential fields removed.

```
{
  // the room ID the event is sent within
  "room_id": "!room:example.org",
  // the implied (or explicit) event type
  "type": "org.example.event_type",
  // for state events, even if an empty string
  "state_key": "",
  // the user ID of the sender
  "sender": "@user:example.org",
  // milliseconds since epoch
  "origin_server_ts": 123456789,
  // the domain of the room owner
  "authorized_sending_server": "owner.example.org",
  "content": {
    // the normal event content
  },
  "hashes": {
    "sha256": "<content hash, just like in Matrix today>"
  }
}
```

[MxContentHashCalculation]

The Linear PDU is then redacted [MxRedaction], canonicalized [MxCanonicalJSON], and signed [MxSigning]. The signature is supplied to the owner server alongside the event itself for sending to the room.

## 6. Membership

After a room is created (by an imagined /createRoom API, for example), it will exist on a single server: the owner's. This is not particularly helpful if the goal is to talk to other people, so a way to involve others in the conversation is needed.

Matrix currently has membership states for join, leave, invite, kick, ban, and knock (request invite). These states have their own set of rules governed by the room version to prevent cases of, for example, ban evasion.

**TODO**: Describe those membership transitions. Currently specified in the Client-Server API https://spec.matrix.org/v1.6/client-server-api/#room-membership (we should move that).

The owner server broadcasts successful membership changes as m.room.member events to all participant servers in the room, including the sending server.

A server is considered to be "in the room" if it has at least one user with join membership state.

## 7. State Events API

Matrix, and therefore Linearized Matrix, tracks changes to the room as *state events*. State events have both an event type and state key to differentiate them from room (or non-state) events. While history for state changes is stored in the room, only the most recent change for an event type and state key pair is considered "current state". For example, the current room name is the most recent m.room.name state event.

As mentioned above, a transport layer would be responsible for the request/response structure for this API, however a need would be present to send (arbitrary) state events, read those state events back, and read the whole of current state (including membership).

## 8. Room Events API

Room events include messages and redactions, as well as messaging features like reactions, edits, etc. These may be encrypted in supported rooms (ones which specify an encryption algorithm in their room state), and in their unencrypted form will use the Matrix concept of Extensible Events.

**TODO**: Update the message format I-D for MSC1767 extensible events and link it here. https://github.com/matrix-org/matrix-spec-proposals/blob/main/proposals/1767-extensible-events.md

A transport layer would specify request/response structures for sending, receiving, reading, and discovering nearby events (for scrollback purposes).

## 9. Room Transfers

The current room owner would be stored as a state event within the room, defaulting to the room creator. To transfer ownership, the current owner chooses a participant server and requests that it accept the ownership role. If the participant server agrees to take ownership, it would create and sign a new room ownership state event. The current owner then signs the ownership state event itself and sends it to all participating servers (including the new owner), just as it would for any other event. All requests from that point forward now go to the new owner, and the old owner becomes a regular participating server.

**TODO**: What do you do if the owner server dies or partitions before transferring to a successor?

## 10. Other APIs

**TODO**: Expand upon this section.

A transport layer would specify request/response structures for:

*Media/attachment distribution

*APIs to support end-to-end encryption

*Ephemeral data such as receipts, typing notifications, and presence

*Resolving an identifer to a room ID

*User profiles (display names and avatars)

*Other APIs as required to support interoperable messaging

## 11. Matrix Room Version

The first room version which supports Linearized Matrix will base its requirements on Matrix's existing Room Version 10 definition [MxRoomVersion10]. Changes will be made to support the following features:

*A description of the authorization rules when not using a DAG.

*The DAG-compatible signing structure for events.

*The use of Matrix's Extensible Events content format.

**TODO**: Expand upon this section with formal details of what the above looks like for a room version.

## 12. Anti-Abuse and Anti-Spam

**TODO**: Expand upon this section.

In a Matrix room, state events get appended to the DAG/array to show intent. If a server wishes to decline the request, such as in the case where the recipient server believes an invite is spammy, it can do so by sending another event to the room. For example, an antispam system might issue redactions for messages which look spammy on behalf of a room admin.

## 13. DAG-Compatible Event Structure

Linearized Matrix is essentially an alternative API for accessing normal Matrix rooms over federation, which means servers which support a full-blown DAG can still join and participate in the room. This is critical in order to avoid breaking compatibility with today's fully-decentralized Matrix, and provides a way to decentralize ownership of rooms even if large messaging providers are themselves not able to implement full decentralization yet. [I-D.nottingham-avoiding-internet-centralization]

With DAG-compatible servers in the room, the DAG-compatible servers talk to each other directly as they do with the current Matrix APIs. Any DAG-compatible server which can also speak Linearized Matrix can connect to the owner server - effectively trunking Linearized Matrix into normal Matrix and tracking its events into the DAG. As long as servers speaking Linearized Matrix uphold the room's access controls, then they appear as a single logical DAG-compatible server to normal Matrix, and will maintain consistency with the rest of normal Matrix.

Normally, events are checked for signatures from the "origin" server implied by the sender on an event. Events sent with the Linearized Matrix API are already signed by the participant server to ensure the owner server isn't spoofing them, however an owner server might not always be DAG-compatible itself. To remedy this, owner servers can delegate their DAG involvement to a DAG-compatible server in the room.

Delegating to a DAG-compatible server means creating a *Delegated Linear PDU* from a *Linear PDU*. The owner moves the authorized_sending_server value (which should be itself) to original_authorized_sending_server then populates authorized_sending_server with the domain name for the DAG-compatible server it is using. The owner server then signs the Delegated Linear PDU and sends it to the DAG-compatible server, which then appends all the DAG-specific fields and signs the resulting PDU itself before sending it to all the other DAG-compatible servers in the room.

Note that while a Delegated Linear PDU modifies the structure that was signed as a Linear PDU, it is easily possible to reconstruct a Linear PDU from a Delegated Linear PDU. Similarly, a DAG-ready PDU can be redacted down to a Delegated Linear PDU with ease.

A complete DAG-ready PDU would look like:

```
{
  // the room ID the event is sent within
  "room_id": "!room:example.org",
  // the implied (or explicit) event type
  "type": "org.example.event_type",
  // for state events, even if an empty string
  "state_key": "",
  // the user ID of the sender
  "sender": "@user:example.org",
  // milliseconds since epoch
  "origin_server_ts": 123456789,
  // the domain of the room owner
  "original_authorized_sending_server": "owner.example.org",
  // DAG-capable server
  "authorized_sending_server": "dag.example.org",
  "content": {
    // the normal event content
  },
  "hashes": {
    "sha256": "<content hash, just like in Matrix today>"
  },
  // other event format stuff:
  "auth_events": ["$event1", "$event2", "$etc"],
  "depth": 42,
  "prev_events": ["$event3", "$event4", "$etc2"],
  "signatures": {
    "dag.example.org": {
      "ed25519:abc": "<signature for PDU>"
    },
    "owner.example.org": {
      "ed25519:def": "<signature for Delegated Linear PDU>"
    },
    "example.org": {
      "ed25519:ghi": "<signature for non-delegated Linear PDU>"
    }
  }
}
```

When validating the signatures [MxSignatureValidation] on this PDU, DAG-capable servers would apply the following algorithm. If at any point the check fails, the algorithm bails.

1. If an original_authorized_sending_server is present, construct the implied Linear PDU from the PDU and validate the signature for the server implied by the sender. Additionally, redact the PDU down to a Delegated Linear PDU and validate the signature for the authorized_sending_server value.

2. If an original_authorized_sending_server is NOT present, redact the PDU down to a Linear PDU and validate the signature for the authorized_sending_server value.

3. Without considering the signatures from the domains in previous steps, verify the signatures per normal. Note that the step where the signature for the "origin server" (defined as the one implied by the PDU's sender) is implicitly checked as part of step 1 and 2, and not actually possible to verify in the traditional sense. That particular step in event validation is therefore skipped when step 1 or 2 is performed.

**TODO**: How does an owner server pick a DAG server to communicate with, and how does the owner receive events from the DAG to send to participant servers? One option might be to have DAG-capable servers identify themselves during joins with the owner server, then the current owner can transfer ownership to the DAG-capable server. The DAG-capable owner would simply shuffle events around internally to feed both API surfaces, though this means all DAG-capable servers need to implement both API surfaces.

## 14. Security Considerations

**TODO**: Expand upon this section.

As discussed in the Event Signing section, we ensure servers are not able to spoof events.

## 15. IANA Considerations

This document has no IANA actions.

## 16. References

### 16.1. Normative References

**[I-D.ralston-mimi-matrix-framework]**
Ralston, T., "Matrix as a Messaging Framework", Work in Progress, Internet-Draft, draft-ralston-mimi-matrix-framework-01, 13 March 2023, <https://datatracker.ietf.org/doc/html/draft-ralston-mimi-matrix-framework-01>.

### 16.2. Informative References

**[I-D.nottingham-avoiding-internet-centralization]**
Nottingham, M., "Internet Centralization: What Can Standards Do?", Work in Progress, Internet-Draft, draft-nottingham-avoiding-internet-centralization-09, 17

February 2023, <https://datatracker.ietf.org/doc/html/draft-nottingham-avoiding-internet-centralization-09>.

**[MxCanonicalJSON]** The Matrix.org Foundation C.I.C., "Matrix Specification | v1.6 | Appendices | Canonical JSON", 2023, <https://spec.matrix.org/v1.6/appendices/#canonical-json>.

**[MxContentHashCalculation]** The Matrix.org Foundation C.I.C., "Matrix Specification | v1.6 | Federation API | Calculating the Content Hash", 2023, <https://spec.matrix.org/v1.6/server-server-api/#calculating-the-content-hash-for-an-event>.

**[MxRedaction]** The Matrix.org Foundation C.I.C., "Matrix Specification | v1.6 | Client-Server API | Redaction Algorithm", 2023, <https://spec.matrix.org/v1.6/client-server-api/#redactions>.

**[MxRoomVersion10]** The Matrix.org Foundation C.I.C., "Matrix Specification | v1.6 | Room Version 10", 2023, <https://spec.matrix.org/v1.6/rooms/v10>.

**[MxSignatureValidation]** The Matrix.org Foundation C.I.C., "Matrix Specification | v1.6 | Federation API | Validating Hashes and Signatures", 2023, <https://spec.matrix.org/v1.6/server-server-api/#validating-hashes-and-signatures-on-received-events>.

**[MxSigning]** The Matrix.org Foundation C.I.C., "Matrix Specification | v1.6 | Appendices | Signing", 2023, <https://spec.matrix.org/v1.6/appendices/#signing-details>.

**[MxV10AuthRules]** The Matrix.org Foundation C.I.C., "Matrix Specification | v1.6 | Room Version 10 | Authorization Rules", 2023, <https://spec.matrix.org/v1.6/rooms/v10/#authorization-rules>.

**[MxV10PDUFormat]** The Matrix.org Foundation C.I.C., "Matrix Specification | v1.6 | Room Version 10 | Event Format", 2023, <https://spec.matrix.org/v1.6/rooms/v10/#event-format-1>.

## Acknowledgments

Thank you to the Matrix Spec Core Team (SCT), and in particular Richard van der Hoff, for exploring how Matrix rooms could be represented as a linear structure, leading to 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