Mimi
Internet-Draft
Intended status: Standards Track
Expires: 14 September 2023

J. Rosenberg
Five9
C. Jennings
S. Nandakumar
Cisco

13 March 2023

More Instant Messaging Interop (MIMI) Transport Protocol
draft-rosenberg-mimi-protocol-00

Abstract

This document specifies the More Instant Messaging Interop (mimi) Transport Protocol (MTP)- a protocol for inter-provider persistent group chat. MIMI utilizes Messaging Layer Security (MLS) for end-to-end encryption of content. The MIMI Transport Protocol plays the role of the Delivery Service (DS) defined by the MLS protocol. MTP is meant to represent the minimal protocol required to enable provider to provider federation for messaging exchange using MLS. It is not suitable for client to provider communications. MTP is based on a pull architecture, wherein message delivery from provider A to provider B is accomplished by having provider B pull messages from provider A. This provides better scalability and reliability and is amenable to implementation in modern cloud software architectures, while also reducing spam risk. MTP serves as a transfer protocol for opaque message content, the format of which is specified in a separate document. MTP is also designed to prevent spam, and does so by introducing a layer of authorization for the establishment of connections and addition to group chats.

Table of Contents

1. Introduction

The MIMI Transport Protocol (MTP) specifies inter-provider persistent group chat. It assumes the existence of chat providers, each of which provides both client applications (often mobile apps) along with a service backend. Many of these providers allow multiple simultaneous active clients per user. Examples of chat providers as of time of writing include WhatsApp, iMessage, Facebook Messenger, Signal and Telegram in the consumer space; and Slack, Microsoft Teams, Wire and Element in the enterprise space. MTP further assumes that the foundational unit of communications within each provider is a group chat. A group chat is a virtual place wherein a set of users can share content, including text messages, images, videos, and so on. A group chat has a membership, which represents a set of users who are permitted to participate in the group chat. These group chats are persistent, in that, a user that is a member of a group chat can see past history of the group chat. Furthermore, a user can send and receive messages into the group chat independently of whether other users in the group chat are currently logged into their client application. This persistent group chat model is different from the session-based real-time chat provided by protocols such as SIMPLE [RFC6914] and XMPP [RFC6120] [RFC6121].

For any given group chat, a single provider is said to be the owning provider for that group chat. The owning provider is the source of truth for the properties (including membership) and message content of that group chat. If the owning provider decides, for example, to close a group chat, it will no longer be possible for participants to post to it, no matter what provider those participants belong to. Typically, if a user is a customer of provider A, and creates a new group chat, provider A ends up being the owner of that group chat.

Mimi enables a group chat to have participants that exist in multiple providers. There will typically be participants in a group chat in the owning provider, but this is not a strict requirement of the mimi protocol. A participant in a group chat that doesnt reside in the same provider as the owning provider, is called a guest participant. Their provider, is called a guest provider. The roles of owning provider, guest participant and guest provider are scoped per group chat. Meaning, in one group chat, a particular provider might be the owning provider. But, for a different group chat, they would be a guest provider.

MTP provides several functions. First, it enables a user in one provider to establish a connection with a user in another provider. A connection is the baseline unit of authorization in MTP. It represents authorization from the inviting user, towards the invited user, for the inviting user to add the invited user to one or more group chats in the inviting user's provider. A connection is fundamentally asymmetric. The connection establishment processes allows the owning provider to obtain this authorization, and to also discovery the identity by which the invited user is known in its provider. MTP supports two distinct flavors of connection establishment - one where the connection is created separately from usage of that connection to add the user to a group chat, and a second in which the connection is concurrent with the addition of that user to a group chat.

Connection establishment makes use of an out of band communications from the inviting user, to the invited user, using a new URI - the mimi URI - that is conveyed out of band. This out of band communications can be via email or text, or even by written or verbal communications. This process is key to ensuring that mimi doesnt become a vehicle for spam and unwanted communications. Because the out of band communications is user to user, not provider to provider, it takes advantage of address books already in place and anti-spam tools already present in the global SMS and email networks. The out of band communications would also include personalized content crafted by the inviting user to help convey context, to help the invited user decide on whether to accept the connection. The MIMI URI, defined in this specification, contains a short lived connection ID. The MIMI URI is used by the guest provider to obtain the context of the connection request - the identity of the inviting user, and optionally the name of the group chat to which they have been invited. With this context, the invited user can decide to accept the connection or not. This acceptance is communicated back to the owning provider.

The second main function of MTP is to allow, for a given group chat, the set of guest providers to synchronize the state of the group chat with the owning provider, and to add messages into the group chat. In this regards, MTP is strictly a provider to provider protocol. We use the words provider to provider, and not server to server, because it is anticipated that MTP between a pair of providers would typically run across a multiplicity of servers utilized by each respective provider in order to achieve scale. Enabling this is an explicit design goal of MTP.

MTP is fundamentally an asymmetric protocol. For any given group chat, transmission of a message from one provider to another provider depends on their roles - owning or guest. Sending a message from a guest provider to an owning provider is accomplished using a RESTful style POST operation using HTTP. In this direction, the guest participant - and thus the provider acting on their behalf - is adding a message into the group chat. That operation must be authorized by the owning provider, and assuming it is allowed, the message is accepted. In the reverse direction, when a message is to be sent from the owning provider to a guest provider, the message is already part of the group chat, and the message delivery is performed using a synchronization technique that utilizes an HTTP long poll.

MTP protocol operations for the purposes of content delivery always take place from the guest provider, towards the owning provider. In other words, the guest provider always acts as an HTTP client. There is no way for the owning provider to send messages to any other provider - guest providers must explicitly retrieve them, and retrieve them for specific, named group chats. This part of the protocol design is also meant to ensure that mimi does not become a vector for spam. It is simply impossible for messages to be pushed into a guest provider using MTP. The guest provider has to ask for them, and to ask for them using a specific group chat. The only way a guest provider can even know to ask for messages in a group chat, is because one of its users learned a group chat ID via a mimi URI and asked their provider to retrieve messages for that group. This technique also means that, the default failsafe for mimi is non-delivery of messages. Meaning, if the guest user or their provider decide to no longer participate in a group chat, messages will not be delivered to the guest provider. It takes active effort on behalf of the guest provider to pull messages.

MTP makes use of Messaging Layer Security (MLS) for end-to-end (e2e) encryption of message content [I-D.ietf-mls-protocol] [I-D.ietf-mls-architecture]. In that regards, MTP plays the role of the Delivery Service (DS) as defined in that specification. It includes the needed primitives for retrieving key packages, transmitting proposal and commit messages, delivery of welcome

messages, and sequencing of commits. MLS is an end-to-end protocol which occurs between client applications. Consequently, the MIMI protocol assumes that providers merely collect those messages from clients and send them across via MIMI when needed, and transmit them to clients when delivered by MIMI. MTP provides additional security measures ontop of MLS. Specifically, it authorizes provider access to a group chat.

Note that MTP does not offer directory services of any kind, and does not allowing an owning provider to determine whether an invited user - as identified by an email address or phone number - is a user of any particular provider. This decision is an explicit one, in order to avoid the significant privacy considerations that would arise from a central directory. In a similar way, mimi does not provide any form of interprovider search. The mimi invitation process assumes that the inviting user has a unique identifer for the invited user, obtained out of bands of the protocol.

2. Definitions

Provider: A provider is an entity which is capable of providing persistent group chat capabilities to a set of users that have signed up to their service. A provider offers its users both a client application and a backend cloud service that powers its client application. A provider is also capable of authenticating its users.

Group Chat: A group chat is a named resource that has properties, most notably the membership of the group, and has a sequence of messages.

Owning Provider: For a given group chat, the owning provider is the provider which acts as the source of truth for the membership, messages, and policy. A request to post a message or change a property must be processed at the owning provider. [TODO: the owning provider is not the DS/AS. Both providers and the client play part of those abstract roles]. In MLS terminology, the owning provider acts as the DS and AS for a particular group chat.

Participant: A user of a group chat.

Guest Provider: For a given group chat, a participant can be a user of the owning provider, or they can be part of a different provider. A provider associated with a user that is not a user of the owning provider, is called a guest provider.

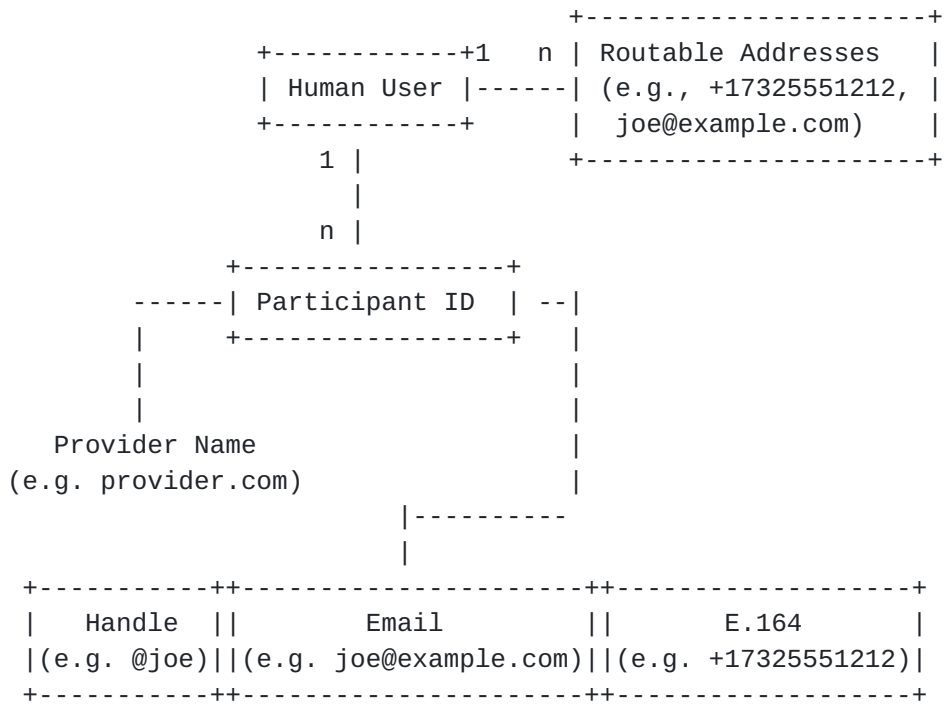
Connection: A unit of authorization between a user in a guest provider (the invited user) and one in an owning provider (calling the inviting user), which grants permission for the inviting user to add the invited user to one or more group chats.

Inviting User: A user that seeks to create a connection with a user in a different provider, or add a user in a different provider to a group chat.

Invited User: A user that has received a request for a connection, or has received a request to be added to a group chat.

3. Identity

Identity is central to the operation of mimi. The mimi identity model is shown in the figure below.



The center of this identity model is a human user (though nothing in mimi requires that it be a human per se). A given human user has a set of routable addresses which can be used to reach them through communications channels outside of group chat. Specifically, they include phone numbers and email addresses over which texts and emails can respectively be received. These routable addresses exist outside of the mimi protocol, but are used to deliver a mimi invitation, which is discussed below.

Within mimi, we have participants, each of which has a unique ID. A participant ID is composed of two parts. The first is the provider name. In mimi, this MUST be a valid DNS name that identifies the provider. Furthermore, the provider entity MUST be the owner of that DNS name, and MUST be capable of obtaining TLS certificates usable with HTTPS for this domain name. The second is the user ID. The user ID takes one of three forms. First, it can be an E.164 number. Secondly, it can be an email address. Thirdly, it can be a handle. When it is a handle, it is unique only within the context of the provider. A phone number or email address is globally unique.

The partipant ID can be written by taking the provider name, followed by a colon, and then the userID. The following are some examples of valid participant IDs:

provider.com:joe@example.com
foo.com:+1732551212
twitter.com:@joe

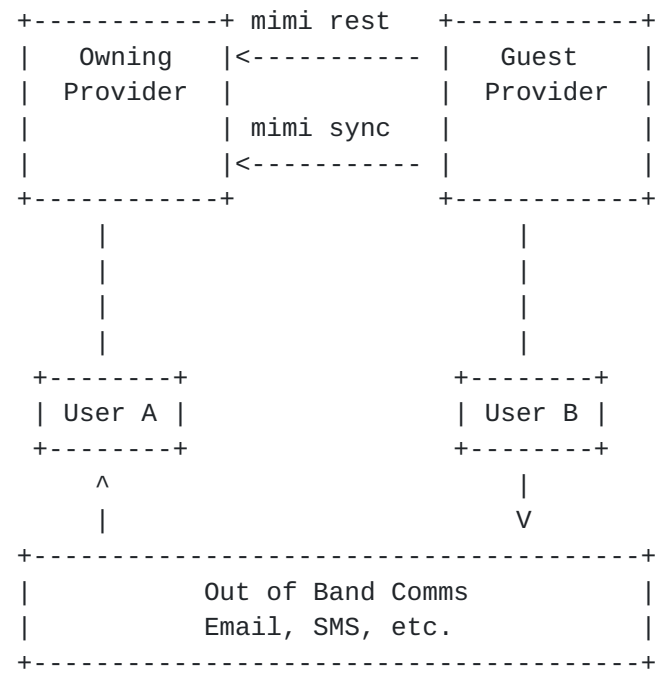
TODO: add formal ABNF

It is extremely important to note that the domain portion of the email address need not (and most often, will not) match the provider name. As a real world example, the provider could be Facebook Messenger, whose provider name is messenger.com. A user within this provider could be Joe Smith, and they have a user ID of joe.smith@gmail.com. In this case, their email is provided by Gmail (gmail.com), but Google/Gmail is not their messaging provider.

Of course, a given human user may have accounts in multiple providers, and perhaps even have multiple accounts within the same provider. Each of these would correspond to a unique participant ID, and thus be viewed as unique participants within the mimi protocol.

4. Reference Architecture

The figure below is the reference architecture for mimi operations.



MTP operations are defined within the context of a particular group chat, or a particular connection. For different group chats or connections, the roles may change. In other words, a particular provider might act as owning provider for some group chats and connections, and guest provider for others. The owning provider for a group chat is the one that acts as the source of truth for a group chat, and acts as the MLS DS for that group chat. The guest provider is the provider for a user that is a guest member of the group chat. In this picture, user A is a participant in the group chat, and is a user of the owning provider. User B is a participant of the group chat, but is a user of the guest provider. There is a connection between user A and user B, and for this connection, user A's provider is the owning provider, and user B's provider is the guest provider. For purposes of establishing connections, MTP assumes the usage of an out of band conveyance technique.

The mimi protocol has two distinct components, which are referred to as mimi rest and mimi sync. The mimi rest component are a sequence of REST APIs which are implemented by the Owning provider, and consumed by the guest provider. The mimi rest component of the protocol includes RESTful resources for retrieving membership, retrieving properties (such as group name), and adding messages into a group chat. It also provides REST endpoints to support MLS operations - specifically, publishing a KeyPackage, retrieving KeyPackages, and sending commit messages which increment the epoch. The second part of the protocol is used to enable real-time synchronization. The guest provider opens an HTTP long poll onnection to the owning provider, and subscribes to a set of group

chats and connections. This subscription specifies the start timestamp for the subscription. The owning provider will authorize and accept the subscription, and then deliver past and future messages. Through this synchronization protocol, the guest provider receives actual messages (text, images, emails, threads, reactions and so on) that have been posted into the group chat. When subscribing to a connection, it receives group addition requests, which must be authorized by the invited user. It will also receive MLS Proposal, Commit, and Welcome messages, and change notifications of group membership and group properties.

The out of band communications modality facilitates the operation of creating a connection. From a mimi perspective, the owning provider generates a mimi URI for the connection, which is then delivered to the inviting user. The inviting user delivers this URI through an out-of-band technique, not specified. The invited user passes this to the guest provider, which then acts on it. The guest provider can then utilize the URI to invoke mimi REST APIs on the owning provider.

5. 1-1 Chats

The atomic unit of communication in MIMI is a group chat, which is a conversation between 1 or more participants. Mimi imposes no restrictions on the number of group chats that can exist with identical membership. A group chat is identified by a UUID, not the identities of its participants.

This introduces some complications in the handling of 1-1 communications, for two reasons. First, many modern messaging systems only show the user a single chat space associated with the target, and do not permit any other users to be added to that 1-1 chat. A second consideration is a complication unique to mimi. Consider two users in different providers. Which provider would act as the owning provider for the 1-1 chat?

Mimi chooses to solve this problem outside of the protocol itself, through the following recommendations.

Firstly, a user has symmetrical group chats - where both have the same pair of users, but differ only in their owning provider, each provider SHOULD merge the contents of these chats when rendering them to the user. This hides the fact that the content may in fact be split across two different group chats under the hood.

Secondly, if the chat provider distinguishes, within its UI, 1-1 vs. group chats, and the user creates a 1-1 chat with a user in a different provider, the inviting provider should treat this group chat as a 1-1 chat, and not permit its user to add additional

parties. This does not prevent the other provider (or its user) from adding participants however, and thus there is some risk that this might happen. [OPEN ISSUE: we can solve this, by adding another property to the group chat to indicate that this meant to be a 1-1 chat. That could then be honored by the other side such that they are not able to add users to it].

6. MIMI URI Syntax

This specification defines a new URI, called the mimi URI. A mimi URI is a URL as defined by [RFC2396]. This specification officially registers the mimi URL using the registration procedures defined in [RFC2717]. The syntax of the mimi URI is as follows, using ABNF [RFC5234]:

```
MIMI-URI         =   "mimi" ":" "//" provider [ ":" port ]
                     "/" connectionId
port             =   1*5DIGIT
provider         =   host
connectionId =   UUID
```

An example of a valid MIMI URI is:

mimi://provider.com/2afae6d3-0b55-4c2d-9dae-3f199d51f83f

The first portion of the mimi URI is the scheme, which MUST be "mimi". What follows is a colon and double slash, present because the MIMI URI is a URL. What follows is a subset of valid authorities as defined by RFC2396. Specifically, it MUST be a valid host. This is followed by an optional port number. Though the MIMI URI doesnt refer to a distinct protocol - it makes use of HTTP - when the port is present, it means that the HTTP request generated for this connection MUST utilize the port number specificed. This is followed by a path separator "/" and then a connectionId. The connectionId MUST be formatted as a UUID. This specification does not require a specific technique for generating UUID, by they MUST be cryptorandom. The cryptograndomness is one dimension of the overall security offered by the mimi protocol. In particular, it prevents malicious guest users and malicious guest providers from trying to forge connection requests.

The MIMI URI is formally registered with IANA using the registration in section {#iana}.

7. Connections

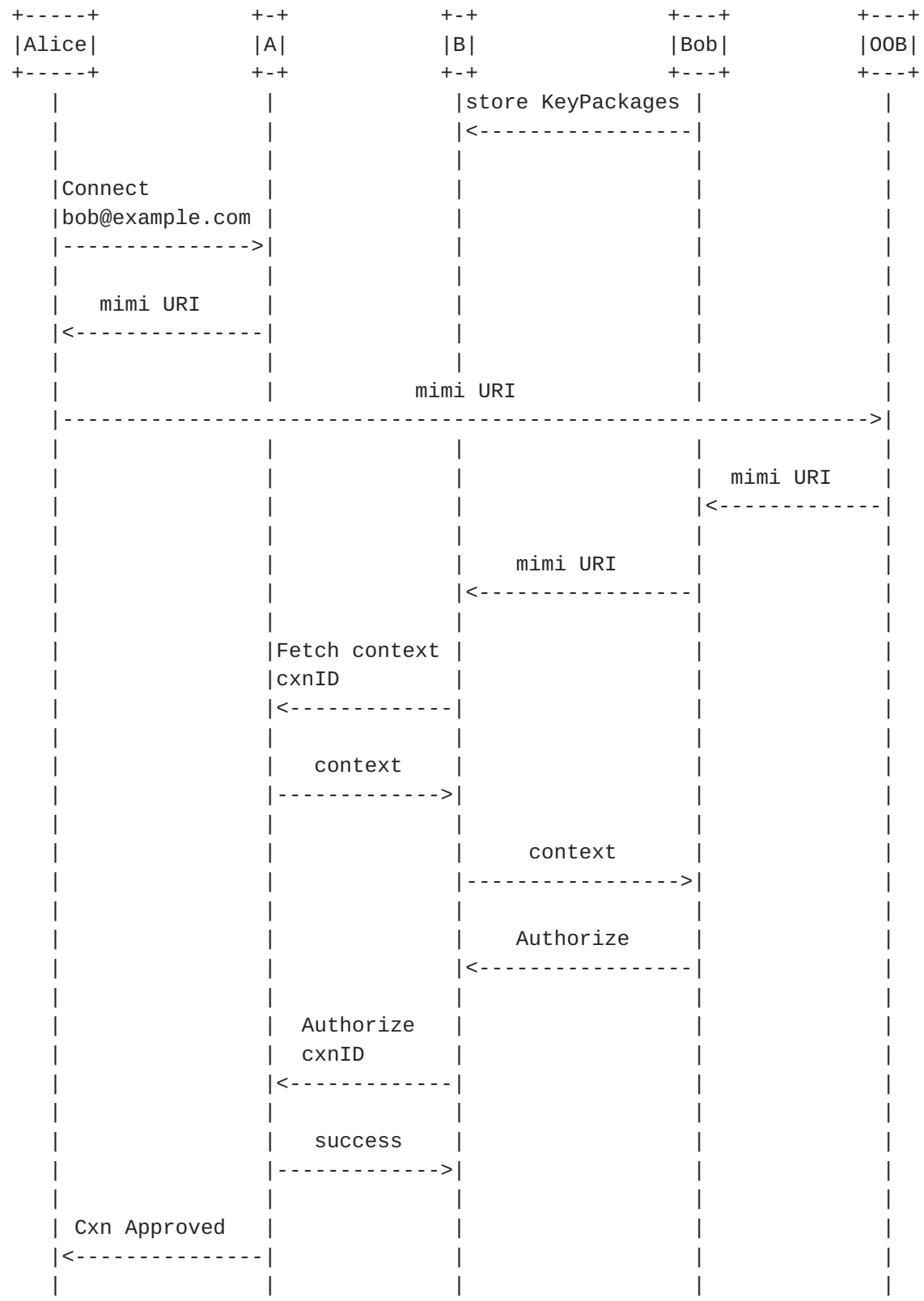
This section specifies the connection process defined for MTP. MTP supports two distinct variations on the connection process - group-chat dependent, or group-chat independent.

In the group-chat independent process, the inviting user first creates the connection to the invited user by specifying their userID. This triggers an out-of-band authorization process, and may take some time for the invited user to accept. Once they accept, the inviting user is notified. At that point, the inviting user can add the invited user to group chats. These subequent additions may be subject to authorized by the invited user as well. Those authorizations do not make use of any out-of-band communications.

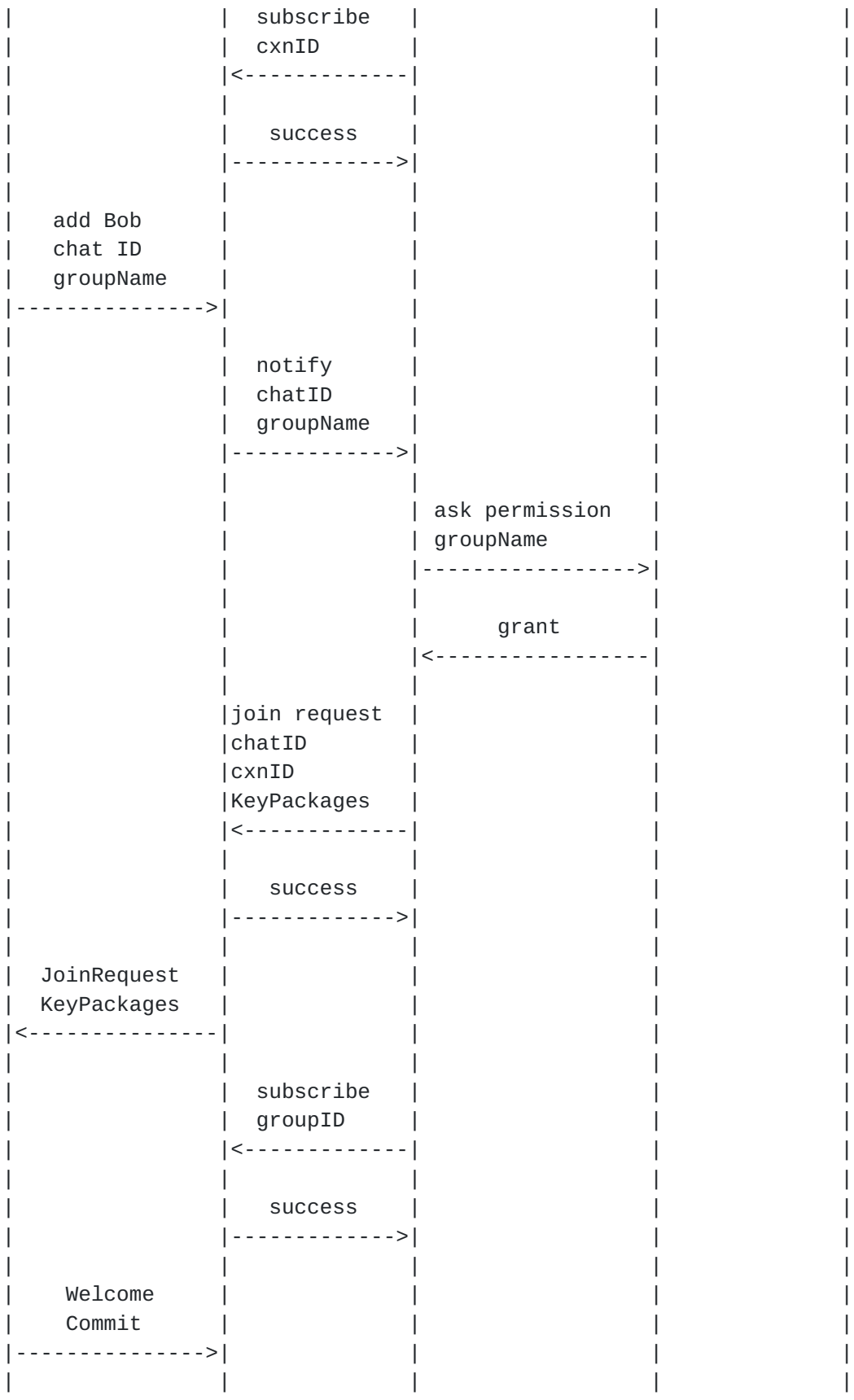
In the group-chat dependent process, the inviting user goes to a group chat, and adds the invited user by specifying their userID. This then triggers the out-of-band process, which may take some time for the invited user to accept. Once they accept, they are added to the group chat. Any any point in the future, the inviting user can add the invited user to other group chats. Those new additions may be subject to authorization by the invited user as well. Those authorizations do not make use of any out-of-band communications.

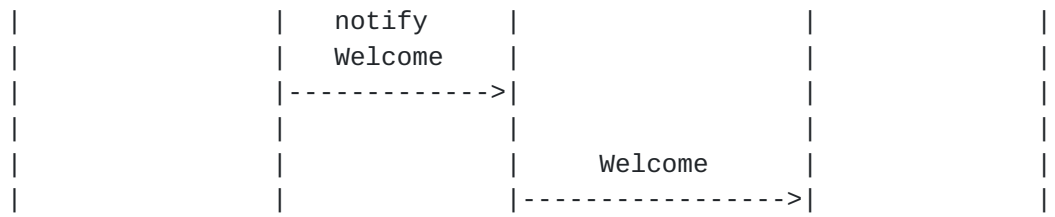
7.1. Group-Chat Independent Connection

The group-chat independent connection process flows as follows:


Alice
A
B
Bob
OOB
store KeyPackages
Connect
bob@example.com
mimi URI
mimi URI
mimi URI
mimi URI
Fetch context
cxnID
context
context
Authorize
Authorize
cxnID
success
Cxn Approved

Continuing:


subscribe
cxnID
success
add Bob
chat ID
groupName
notify
chatID
groupName
ask permission
groupName
grant
join request
chatID
cxnID
KeyPackages
success
JoinRequest
KeyPackages
subscribe
groupID
success
Welcome
Commit



When Bob first logs into a client, which causes him to upload KeyPackages to his provider B for usage in the connection process which follows. Each client used by Bob will trigger an upload of KeyPackages. It is a matter of local policy on how many KeyPackages Bob provides. KeyPackage exhaustion attacks are mitigated by MTP, this is discussed below.

The connection process starts when Alice decides to create a connection to Bob. She requests to her provider to establish the connection, and provides an identifier for Bob. This identifier can be an email address, phone number or handle at a provider. The provider generates a connectionId, which MUST be unique in space and time. The provider MUST store this ID for a mimimum of 24 hours. The provider MUST store, associated with that ID, the identity of the user requesting the connection - userID and display name (Alice Doe, alice@gmail.com in this example), the identity of the user being invited (bob@example.com in this example), and the time at which the connection was requested. It then returns a MIMI URI to Alice's client, and instructs her to communicate this MIMI URI to Bob through some out of band means.

Alice then sends the MIMI URI to Bob, through email, text or other means. This communication is directly from Alice to Bob, and does not utilize their respective providers chat services. In the case of texting for example, the SMS would be sent from Alice's mobile number to Bob's mobile number. In the case of email, it would be sent using one of Alice's email addresses from whatever email provider she uses, and be delivered to Bob's inbox. If Bob has Alice is his address book, or has received prior emails or texts from Alice, this will help him confirm that the connection request is truly from Alice. Typically, Alice would include additional information to help Bob decide whether to accept the request or not. For example, "Hi Bob - lets chat together, you can use whatever chat client you want - click here to accept. I want to discuss the new MIMI working group content with you.".

When Bob clicks on this link, it launches his preferred MIMI-enabled client, and the connection ID is passed to his client application. His client application would pass that to his provider. To gain more information about this connection request, Bob's provider utilizes MTP, and initiates a GET request to the owning provider, using the provider name extracted from the MIMI URI. Provider B specifically invokes the following REST API call:

```
GET https://{provider-name}/.well-known/mimi/
  connections/{connectionId}
```

This request will contain an HTTP Authorization header field, containing a bearer token obtained out of bands from MTP. This is described further below. Using this bearer token, provider B will know that this is provider A, and can authorize the request.

Once the GET request is authorized, Alice's provider A looks up the connection ID and retrieves Alice's name, Bob's userID and the time of the invitation. This information is passed back to provider B. Provider B MUST verify that the userID provided for Bob matches an identity by which Bob is known in the system. This is to prevent against copy-paste attacks, wherein the MIMI URI or connection ID is forwarded to a different user. Note that, there is no cryptographic verification of Bob's userID here. This component of the authorization process depends on mutual trust between the owning and guest providers.

This context for the connection ID can then be shown to Bob in his client UI. He might see something like, "Alice Doe (alice@gmail.com) wants to add you to one or more group chats she's hosting on her provider. Do you want to allow that?". If Bob approves, he informs his provider. The guest provider MUST obtain explicit permission from the end user before accepting the connection request. Note that, this requirement is not enforced through e2e cryptographic means, and depends on a well-behaved guest provider. Once Bob's provider has obtained this permission, it MUST inform the owning provider as follows:

```
POST https://{provider-name}/.well-known/mimi/
  connections/{connectionId}?accept
```

The connection request can also be rejected, as specified in the REST API details below. The response to this HTTP POST will be a JSON object, described below, which includes a connection ID resource that lives on the owning provider, along with a URI for it. This URI allows the guest provider to create a subscription to the connection. This subscription, accomplished via a long poll, allows the guest provider to request a stream of events for the connection. The

events include actions by the inviting user in the owning provider - Alice here - to add Bob to group chats. Should Alice decide to de-authorize the connection, this would also be delivered as an event over the subscription.

Alice's provider will also inform her that Bob has accepted the connection. This may only get delivered next time Alice logs in, as she may be offline when Bob approved the connection.

At some point in the future, Alice does log in, sees that the connection was approved, and now is capable of adding Bob to a group chat, which she does. To allow Bob to make a useful decision on whether to join this group - the name of the group, which will not be e2e encrypted with MLS - can be provided by Alice to her provider along with the request to add Bob (alternatively, the provider may already have it through some other means). Her provider, provider A, then requests permission from Bob to be added to the group chat. This is done by adding an event to the connection, which is a "group chat addition request". This event will get delivered via an event over the subscription to the connection to provider B. It always includes the groupID. Had Alice, or her provider, had access to the group name, this would be included as well. Details on the syntax of this event are described below. With this event in hand, provider B - based on its policies - can either request Bob's permission for that specific group, or take Bob's prior authorization as blanket approval for addition to all groups. This is a matter of local policy at provider B. In this example flow, provider B does request permission, and passes the groupName to Bob to make a decision. For example, he may see something like, "Alice wants to add you to the group 'MIMI Discussion'. Do you want to be added"?

Bob informs his provider that he approves the addition to the group. His provider, provider B, takes all the previously stored KeyPackages from all of Bob's clients, and uses them to send a join REST API call to provider A. That is done using the following REST API:

```
POST https://{provider-name}/.well-known/mimi/group-chats/
     {groupChatId}/participants?connect={connectionId}
```

This request contains a single URI parameter - the connectionID - which is used for authorization purposes. The owning provider MUST verify that this connectionID exists and is currently valid, and that the particular group chat ID has been authorized for that connection (which, in this flow, it is).

The body of this request will contain a multi-part MIME body, each of which contains an MLS KeyPackage. Provider B stores these key packages for delivery to Alice's client. It informs provider A of

the success. The response contains a JSON blob that includes a timestamp (used for synchronization). Now, provider B can create a subscription to receive events for this group chat. It does so using the following long-poll URI:

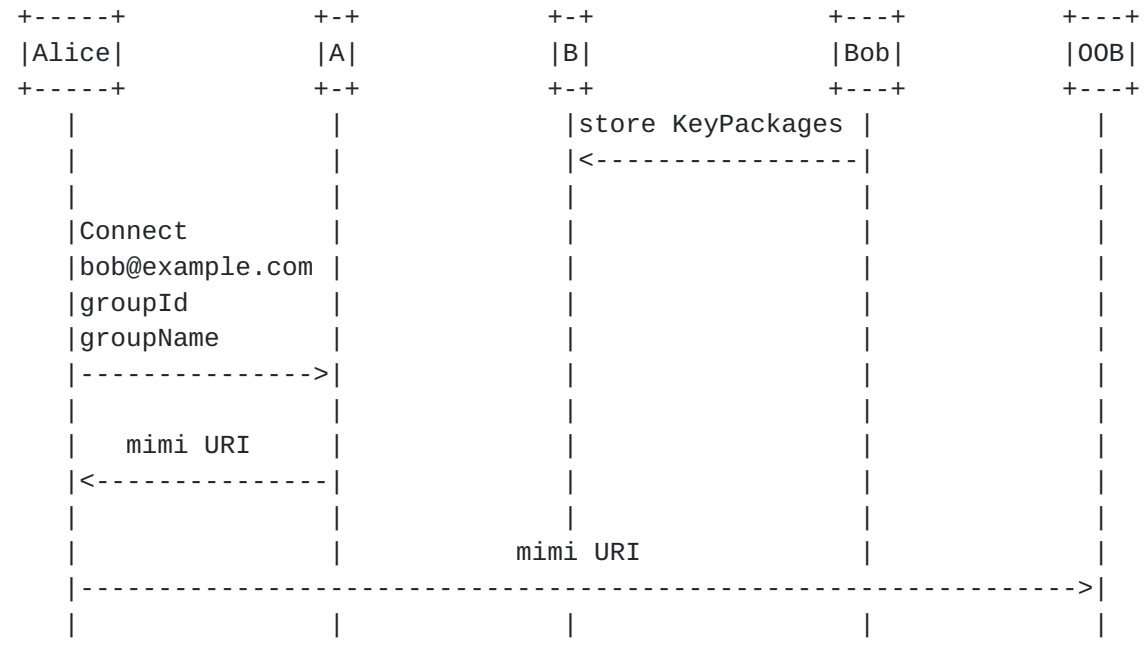
```
POST https://{provider-name}/.well-known/mimi/group-chats/
     {groupChatId}/subscriptions?connect={connectionId}&from={timestamp}
```

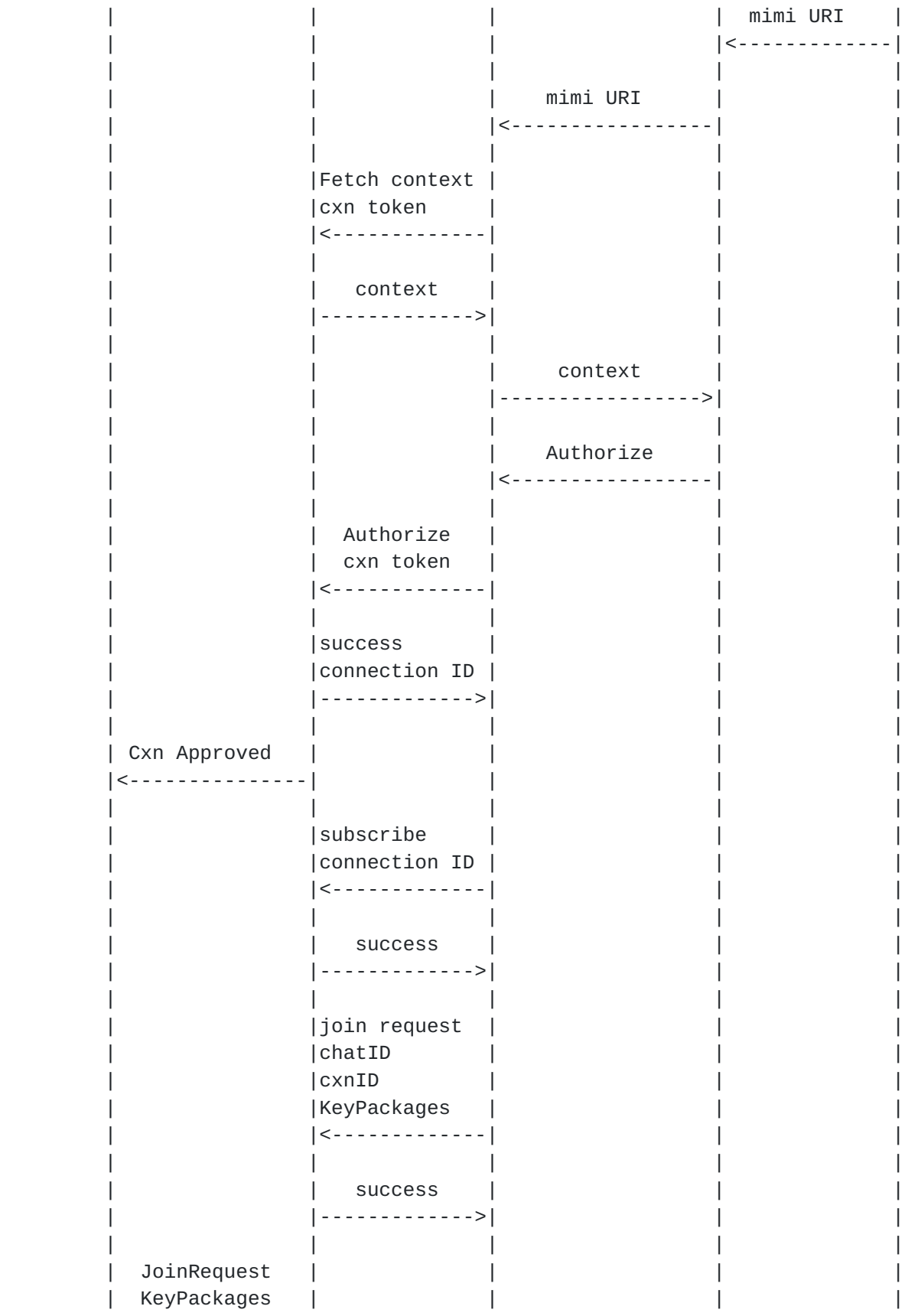
The connectionID is used by the owning provider once again to authorize the subcription, using the same logic as for the join request. The subscription includes a from URI parameter, which indicates the timestamp of the earliest set of events that the provider requires. This timestamp MUST be no larger than the value returned in the response to the join, to avoid race conditions where it misses the Welcome message.

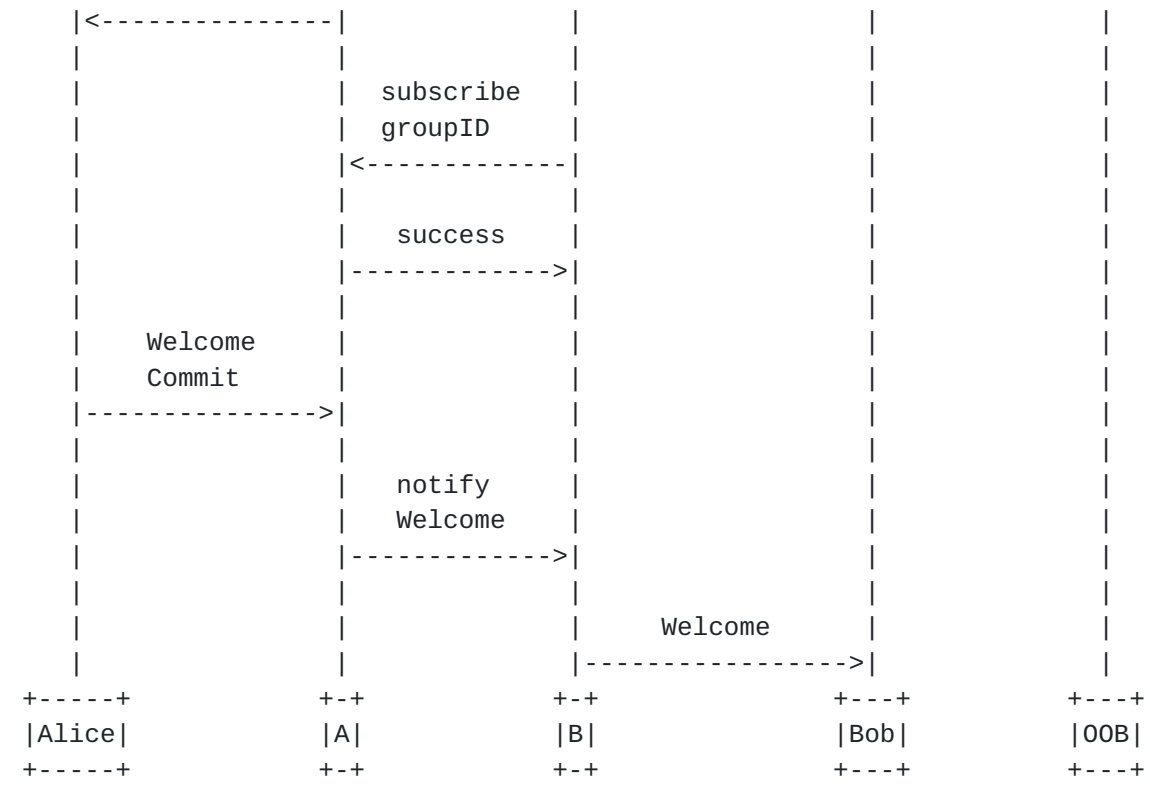
Provider B delivers these keypackages to Alice next time one of her clients comes online [OPEN ISSUE: the provider could also deliver them to any other user, not just ALice. Doing that however introduces an attack wherein a malicious owning provider can trick clients into adding users they didnt intend to. So, this seems like a bad idea.] Once Alice gets these KeyPackages, she generates the MLS Welcome and Commit messages and sends them to her provider. The provider then sends the Welcome message over the long poll connection, and this too is delivered to Bob.

7.2. Group Chat Dependent Connection

The group chat dependent process flows as follows:




mimi URI
mimi URI
Fetch context
cxn token
context
context
Authorize
Authorize
cxn token
success
connection ID
Cxn Approved
subscribe
connection ID
success
join request
chatID
cxnID
KeyPackages
success
JoinRequest
KeyPackages



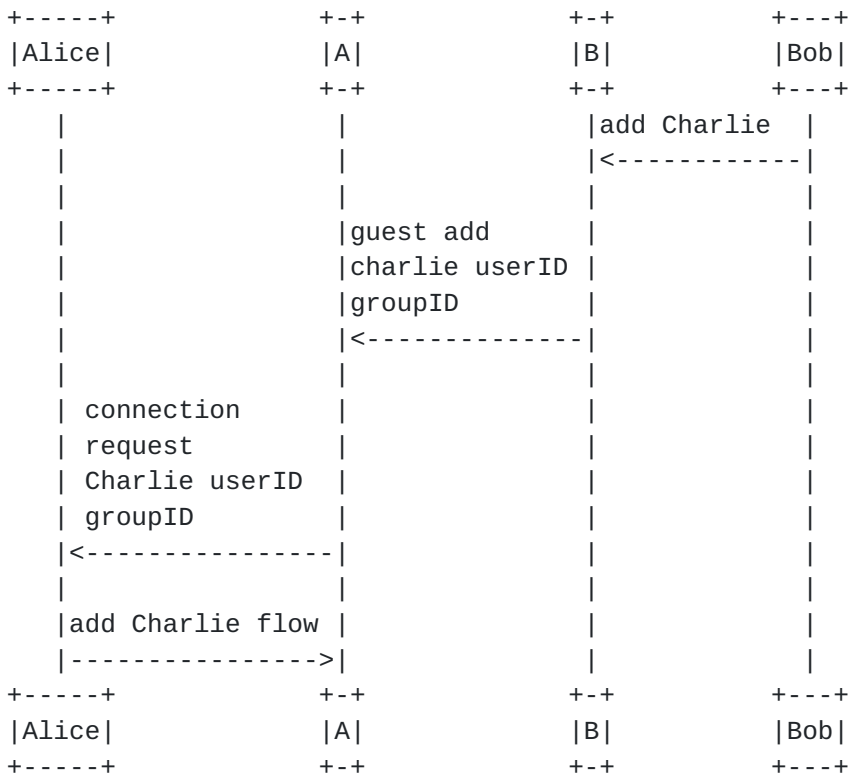
The significant difference from the connection independent flow is that this connection request is made coincident with adding the user Bob to the group chat. Alice's provider, in this case, offers her the ability to go to a group chat she is in, hit the 'add' button, and enter a userID for a user in another provider. She does so, entering a userID for Bob and passes this to her provider. Alice's provider generates a connectionID and MIMI URI as in the use case above. However, in this flow, it associates the groupName and groupId to the connectionID in addition to the timestamp, Alice's display name and userID and Bob's userID.

This MIMI URI is delivered out of band to Bob. Bob's provider fetches the context for the connection ID. As with the other flow, the context includes Alice's email address and display name. However, it also includes the groupName and groupID. To avoid risks of spamming users with content delivered via these connection requests, the provider B SHOULD first prompt Bob with just Alice's email address, and if Bob confirms, only then prompt Bob with the group name.

Once Bob authorizes - he is authorizing the connection and the addition to the group chat concurrently. His provider, provider B, will authorize the connection using the connection ID. It subscribes to the connection, as above. In this case, it can immediately turn around and request a join to the group. This join is identical to the case above, and will include the groupID, connectionID and KeyPackages. The remainder of the flow is identical to the use case above.

7.3. Guest Connections

Another use case supported by MTP, is that a guest user has already been added to a group chat, and that guest user wishes to add someone else to the group chat. We call this a guest addition. The flow for that works as follows:



In this case, we have a group chat owned by owning provider A. Alice is one of the members of the group chat, also in the owning provider. Bob is a guest participant, and his provider is provider B. Charlie is also a user of provider B. Bob uses his client application and adds Charlie to the group chat. His provider would have a directory of users in this use case (since Charlie is in the same provider). Once Charlie has been selected, provider B then sends a request to the owning provider, provider A, to add Charlie to the group. This is done using the guest addition REST API:

```
POST https://{provider-name}/.well-known/mimi/group-chats/
     {groupChatId}/participants/{userId}?connect={connectionId}
```

In this case, because the userID to be added is different than the one bound to the connection, the owning provider knows this is a guest addition. There will be no KeyPackages in this POST of course. This is treated like a request to the owning provider, for someone in the owning provider to create a connection to Bob in order for him to be added. The owning provider can send this to all participants in the group that are within the owning provider, or just some. One might imagine they get a message like "Bob wants to add Charlie to this group. To do that, you need to reach out to Charlie and add him in. OK?". If the user accepts, this would trigger a connection request flow - either group dependent or group independent, based on the policy of the provider.

[OPEN ISSUE: this is definitely clunky - what if Alice doesnt know Charlie at all? We could support the ability for Bob to get a MIMI URI directly as well. That would allow Bob to communicate to CHarlie directly, but now imposes a greater security risk because A must trust that Bob isnt going to generate a lot of spam invites. The solution here optimizes for spam reduction at the expense of user experience.]

7.4. Mobile Centric UI

This section describes a suggestion user interface implementation for mobile apps, and a mobile number centric invitation process. It also suggests an enhancement to mobile OS's for formal support for the mimi URI.

In one possible implementation of a UI, a provider may elect to allow its users to add other users by entering their mobile phone numbers, perhaps by allowing the user to select from an address book. The provider would likely check its internal databases to determine whether the selected phone number is already a user of the system, and if not, offer the user the ability to invite them to join the conversation on whatever is their preferred provider. The mobile app of the owning provider would then cross-launch the native texting application on the mobile phone, and prepopulate a text message along the lines of, "Hi! I'd like to chat with you, you can do so using your favorite chat app - click here to join me and others: [link]". The link in the text message would be the mimi URI created by the owning domain.

Assuming the inviting user already has some kind of SMS chat in progress with the invited user, this new message would appear within that context. The inviting user can press send, and this is

delivered via text to the invited user. This provides a form of forward routability, so that the invitation is only delivered to a user that owns that phone number. The invited user can, at their discretion, send additional text messages to provide context.

This text would be received on the mobile phone of the receiving user. Note that it is a new URI scheme - the mimi URI. As of writing, both iOS and Android allow applications to register custom URI handlers. When an application supports mimi, it would register itself as a handler for the mimi URI. Furthermore, it would provide a setting in its settings menu, to say whether that particular app is the default application for handling inter-app messaging. If a user were to have multiple mimi-enabled apps on their phone, it would be at the discretion of the user to select only one as their default. When an app is unselected as default, it would unregister itself as a handler for the mimi URI. When the user clicks on the mimi URI, it would cross-launch their selected mobile app and pass it the URI parameters. The mobile application would prompt the user with something like, "Accept invitation to join chat room?" and if they say yes, it would initiate the acceptance procedures defined above.

If mimi were to be adopted by the mobile OS vendors, an even better experience could be provided. The mobile OS vendors would explciitly recognize the mimi URI scheme, and allow multiple apps to register as handlers. The OS itself would have a setting for the default, which is perhaps the first one to register. When the second mobile app registers as a handler, the OS could ask the user if they wish to switch to that app as the default, or keep it. Similarly, there would be a setting in the mobile OS, which allows the user to pick their preferred app. This would show a dropdown list of all vendors whose apps are currently installed, but have registered themselves as handlers for the mimi URI. This is a (relatively) small change in functionality and thus not a big change. It is important to note that mimi is designed to work without any mobile OS support - it only requires implementation within the providers. Mobile OS support improves the experience by making management of the default messaging app much better.

8. REST APIs

This section defines the REST endpoints for mimi. All mimi REST calls are from the guest provider to the owning provider. As a result, mimi only specifies behaviors for these endpoints on the owning provider. [TODO: add rest from owning to guest for claiming keypackages.]

8.1. Authentication and Authorization

The owning provider MUST provide a mechanism, outside the scope of this specification, for a guest provider to obtain an OAuth access token that can be used as a bearer token in all HTTP requests. [@cullen: suggestions for additional wording here please]. This allows the owning provider to make decisions about which guest providers it will permit. Mimi does not require any specific authorization policy, and leaves this to the discretion of an owning provider, on which guest providers it will enable to connect. [OPEN ISSUE: an alternative is mutual TLS. Which do we want, or both?]

8.2. General Request Guidelines

All mimi REST API operations MUST occur over HTTPS. All requests MUST include an Authorization header field containing the access token obtained by the provider out of band. All URI endpoints exist within the .well-known path, in order to avoid any possible namespace collisions with existing HTTP URIs hosted on the provider [RFC5785]. This specification registers a new well-known URI, "mimi".

REST endpoints which return collections of resources support pagination. Pagination makes use of a page cursor technique. Paginated results do not guarantee that an item appears in only one page - the client must be prepared for duplication. A client can request an upper bound on elements per page with an optional pageLimit URI query parameter. For any paginated results, the server MUST include a pageLimit URI parameter in the response body, whose value is equal to or less than any pageLimit specified in the request URI. Each response will contain a "next" URI that includes a link to the next page, if present. That request will contain a "pageCursor" value, which is a string generated by the server and opaque to the client. This can be used by the server to support pagination and contain whatever content it likes. THe server MUST NOT send a pageCursor longer than 1023 characters.

The body of any response that provides pagination looks like this:

```
{
  "items": [
  ...
  ],
  "paging": {
    "next": "{uri}&pageCursor={pageCursor}",
    "limit": {pageLimit}
  }
}
```

Where the items property contains the array of items, and a paging property contains the information need for pagination.

Where {variable} is substituted with the value of that variable.

8.3. Connection Information

This REST endpoint allows a guest provider to obtain the meta-data associated with a connection. This is used to render the information to its user, to allow the user to make a decision on whether to accept the connection. The syntax for this endpoint is:

```
GET https://{provider-name}/.well-known/mimi/
  connections/{connectionId}
```

The owning provider MUST authorize that this is a valid guest provider. The owning provider MUST validate that the connection ID is valid. Owning providers MUST maintain a minimum validity period of 24 hours for connection IDs in the PENDING state. Once authorized, the server returns the following payload in the 200 OK response, which represents the state of the connection resource:

```
{
    "id" : "{connectionId}",
    "uri" : "https://{provider-name}/.well-known/mimi/
             connections/{connectionID}",

    "createdAt" : "{timestamp}"
    "state" : "{state}",

    "source" : {
      "userID" : "{userId}",
      "provider" : "{provider-name}"
    },

    "target" : {
      "userId" : "{userId}"
      "provider" : "{provider-name}"
    },

    "groupChat" : {
        "id" : "{groupChatId}",
        "uri" : "https://{provider-name}/.well-known/mimi/
                group-chats/{groupChatId}/"
    }
}
```

The id, uri, createdAt, source and target properties are mandatory. The source property refers to the user that is requesting the connection, and will be a user of the owning provider. The target represents the user whose connection is requested. Initially, the provider for this user is unknown. Consequently, only the userID property of the target will be present. The state property indicates the state of the connection, and is either PENDING or ACTIVE. When first created, its state is PENDING. Once confirmed, its state moves to ACTIVE. In the ACTIVE state, the provider-name for the target MUST be present.

The groupChat property is present only when the connection was requested in the context of a specific group chat. [OPEN ISSUE: should this be a list, so as more group chats are authorized this grows?]. In the case of a group chat independent connection request, it would be absent.

8.4. Connection Acceptance or Rejection

This REST endpoint allows a guest provider to accept a connection request on behalf of one of its users. The syntax for this endpoint is:

```
POST https://{provider-name}/.well-known/mimi/
  connections/{connectionID}
```

The request can take either take the "accept" URI parameter, which indicates acceptance, or the "reject" URI parameter, which indicates rejection. Only one of the two can be present; a request with both is rejected with a 400.

The owning provider MUST authorize that this is a valid guest provider. The owning provider MUST validate that the connection ID is valid. Owning providers MUST maintain a minimum validity period of 24 hours for connection IDs in the PENDING state.

If the prior state of the connection was either PENDING or ACTIVE, and the accept parameter was present, the new state is ACTIVE. If the reject parameter is present, this is equivalent to deletion of the connection and the resource would no longer exist [OPEN ISSUE: Should we just do DELETE instead of a reject URI param]

If the guest provider has accepted the connection, the response is a 200 OK containing a JSON payload in the same syntax as above. However, in this case, the STATE would be active and the target will include the provider-name. The owning provider knows the guest provider name, as a consequence of the authentication process used to authenticate the guest provider.

8.5. Join Request

This REST endpoint allows a guest provider to request that their user join a group chat. This is always in response to a prior request from a user in the owning provider to add them, followed by an authorization decision by the invited user to proceed with the addition. The syntax for this endpoint is:

```
POST https://{provider-name}/.well-known/mimi/group-chats/
  {groupChatId}/participants?connect={connectionId}
```

From an MLS perspective, the user is not yet joined to the group - this method is delivering the key packages to the inviting user, so the inviting user in the owning provider can complete the addition of this user to the group.

The only valid method against this endpoint is POST, which performs the join request operation by creating this participant in the group.

The request has a single mandatory URI parameter, "connect", which contains the connectionID associated with this join request.

The server MUST validate that the groupChatId is valid, and represents an active group chat. The server MUST validate that the connectionID in the connect parameter is valid, and represents a valid connection whose state is ACTIVE. If any one of these steps fails, the request MUST be rejected with a 403. Otherwise, the server proceeds to the next step.

The body of the request MUST be of type multipart MIME. Each part MUST be of type message/mls, and MUST correspond to a KeyPackage. There SHOULD be a KeyPackage for each client that the guest currently has, and for each, include a KeyPackage for each ciphersuite supported by each client.

The response to a successful request is a 201 created, which contains a URI for the participant resource that was just created. This URI will be of the form:

```
https://{provider-name}/.well-known/mimi/group-chats/
  {groupChatId}/participants/{participantUUID}
```

Note that this is a UUID for the participant, distinct from its more human readable participantID. This allows the URIs to contain a UUID without needing to worry about URI encoding of the participantID.

The body contains a JSON representation of the participant. This JSON object is structured as follows:

```
{
    "id" : "{participantUUID}",
    "participantID" : "{participantId},
    "uri" : "https://{provider-name}/.well-known/mimi/
             group-chats/ {groupChatId}/participants/
             {participantUUID}",

    "joinedAt" : "{timestamp}"
    "provider" : "{provider-name}"

    "groupChat" : {
        "id" : "{groupChatId}",
        "uri" : "https://{provider-name}/.well-known/mimi/
                 group-chats/{groupChatId}/"
    }
}
```

The timestamp MUST be formatted as a string in JSON, and be no longer than 16 characters. Its content MUST be valid timestamp, a positive integer measuring the number of milliseconds since Unix epoch time. That allows the timestamp to be monotonically increasing at the owning provider, and valid across different backend servers that may process the request.

8.6. Leave Request

If a guest participant wishes to leave, its provider can invoke the following API:

```
DELETE https://{provider-name}/.well-known/mimi/group-chats/
  {groupChatId}/participants/{participantUUID}
```

The server MUST authorize that the request comes from a valid guest provider, and that the guest provider name matches the one associated with the user that is being removed. The server MUST validate that the groupID exists and is valid, and that the participant in question is part of the group chat at this time. Assuming all checks pass, the user is removed from the group at the MTP layer. The abouve REST endpoint will not longer be valid.

This operation has no effect yet on the MLS of course. To do that, the owning domain SHOULD request a user in the owning domain to remove the participant. A user, upon accepting this request to be removed, will generate a new Commit message and sent it to the owning provider. This new Commit, with this user removed, will be distributed via the event stream to all other remaining participants.

[OPEN ISSUE: Do we want to allow non-owning members to process leave requests?]

The DELETE request will generate an immediate 200 OK response and not await the MLS Commit message which removes the member from the group.

8.7. Guest Addition

```
POST https://{provider-name}/.well-known/mimi/group-chats/
     {groupChatId}/participants/{userId}?connect={connectionId}
```

Details to be filled in.

8.8. Add Message

If a guest user wishes to add a message to a group chat, their provider invokes the following API:

```
POST https://{provider-name}/.well-known/mimi/group-chats/
      {groupChatId}/participants/{participantUUID}/messages
```

The body of the request has the media type message/mls. It contains the end-to-end encrypted application message.

The server MUST validate that the groupChatId exists and is valid, the user with userId is a member of the group, has permissions to post messages, and that their provider is the same as the provider making the request (based on the Authorization header field). If any of these checks fail, the server MUST return a 403.

If the message addition is successful, the server returns a 200 OK that includes the following JSON payload. This is not strictly needed for synchronization but reflects best practice in REST APIs to return a copy of the resource that was created. In this case, the meta-data of the message.

```
{
    "id" : "{messageTimestamp}",
    "uri" : "https://{provider-name}/.well-known/mimi/group-chats/
             {groupChatId}/participants/{userId}/messages/
             {messageTimestamp}",

    "groupChat" : {
        "id" : "{groupChatId}",
        "uri" : "https://{provider-name}/.well-known/mimi/
                 group-chats/{groupChatId}/"
    }
}
```

It is very important to note that the ID of this messages is not a UUID, but rather is a timestamp that represents the number of milliseconds since the Unix epoch time. This makes each message uniquely identified by the group chat ID and the timestamp. The server MUST enforce a rule that no two messages posted ever have the same timestamp, in order for the timestamps to serve as unique IDs. The timestamp is generated by the owning provider.

8.9. Retrieve Membership

When a user intially joins the group, their UI will need to show the list of group members, including display names. Display names are not part of the MLS messages, and thus this endpoint is provided to allow a guest to retrieve them in paginated form for rendering purposes [OPEN ISSUE: This is contentious, do we need this]. This list is maintained by the owning provider. The following REST endpoint allows the guest provider to retrieve the current group membership:

```
GET https://{provider-name}/.well-known/mimi/group-chats/
    {groupChatId}/participants/
```

The server MUST validate that the groupChatId exists and is valid, and that at least one participant in the group is in a guest provider, and the guest provider name matches the provider name associated with the access token in the Authorization header field. If these checks do not pass, a 403 Forbidden response is returned.

Otherwise, the server returns the group membership as a pagination. This follows the guidelines defined above for pagination. Each element of the item array is formatted as following:

```
{
    "id" : "{userId}",
    "uri" : "https://{provider-name}/.well-known/mimi/group-chats/
             {groupChatId}/participants/{participantUUID}",
    "name" : "{displayname},

    "properties" {
        "provider" : "provider-name"
    },

    "groupChat" : {
        "id" : "{groupChatId}",
        "uri" : "https://{provider-name}/.well-known/mimi/
                 group-chats/{groupChatId}/"
    }
}
```

The displayname is for display purposes, and can be set by the user at any time. It is however RECOMMENDED that mimi implementations on mobile devices which have access to the address book, instead utilize the display name from the address book instead of ones conveyed by mimi.

Each user resource also contains a set of user properties. This set is extensible, either through an extension through a standards body, or through private registration. Initially, this specification has only a single property defined - provider - which contains the provider-name for that user. This would allow user interfaces to indicate when users are "guests" and which app they are using to access the chat. Additional properties which could be considered for future extension include whether or not a user is a moderator of the group, whether they have permission to post or delete messages, and so on. The owning provider includes the list of all properties it supports; any properties not understood by the client are just ignored.

The group membership will include members that joined the group through the join REST API above, but for whom the MLS rekeying has not yet completed. Similarly, the membership will omit members that left the group using the REST API above, but for whom the MLS rekeying has not yet completed.

8.10. Retrieve Group Chat Information

To retrieve information about the group:

```
GET https://{provider-name}/.well-known/mimi/group-chats/{groupChatId}
```

The server MUST validate that the groupChatId exists and is valid, and that at least one participant in the group is in a guest provider, and the guest provider name matches the provider name associated with the access token in the Authorization header field. If these checks do not pass, a 403 Forbidden response is returned.

Mimi allows groups to have properties. [OPEN ISSUE: should we have default properties to handle things like moderation or make this extensible] Properties represent information about the group, distinct from its membership and messages. This specification defines a single property - groupName. Mimi allows for these to be extended to provide additional properties in the future. These extensions can be through IETF consensus or private registrations. Examples of functionality which can be introduced through new properties in the future include broadcast or read-only groups, moderated groups, and so on. If an owning domain is providing a property which is not registered with IANA, it MUST use a property

name that contains a reverse DNS version of its provider name, i.e., "com.messenger.newprop" would be used for a property called newprop that is proprietary to Facebook Messenger.

These properties, including the group name, are returned in the response:

```
{
    "id" : "{groupId}",
    "uri" : "https://{provider-name}/.well-known/mimi/
             group-chats/{groupChatId}",

    "properties" {
        "groupName" : "{groupName}"
    },
}
```

8.11. Set Group Property

To set a property of the group, including the group name, the following API is used:

```
POST https://{provider-name}/.well-known/mimi/
      group-chats/{groupChatId}
```

This requires has a mandatory URI parameter, userID, which contains the user that is performing this request. This request requires a second URI parameter, which is the property. This specification defines only one - groupname, whose value is a URI encoded version of the group name to use for the group. Any other properties known to the server, can be accepted in the request and saved.

The server MUST validate that the groupChatId exists and is valid, that user userID in the userID parameter is valid and a member of the group, and their guest provider name matches the provider name associated with the access token in the Authorization header field. If these checks do not pass, a 403 Forbidden response is returned.

The server then executes the property change. For the name, this name is set as a property of the group.

Note that, the group name and associated properties are not e2e encrypted with MLS, since they may affect server side processing. [OPEN ISSUE: Im sure we need to discuss this one, including display name].

8.12. Commit

This REST endpoint is used to support the MLS protoocl, and allows users in guest providers to push a Commit message into the system, to be delivered to all other participants. [OPEN ISSUE: renaming this "handshake" to allow bare proposals as well.]

```
POST https://{provider-name}/.well-known/mimi/
      group-chats/{groupChatId}/commits
```

This requires has a mandatory URI parameter, participantUUID, which contains the participant that is performing this request.

The server MUST validate that the groupChatId exists and is valid, that participantUUID is valid and a member of the group, and their guest provider name matches the provider name associated with the access token in the Authorization header field. If these checks do not pass, a 403 Forbidden response is returned.

When receiving this request, the server MUST increase the epoch. [TODO: The _server_ doesn't increase the epoch. The Commit caused the group to enter a new epoch. The owning provider agrees that the new Commit is valid and therefore allows it to be sent to other clients.] It MUST ensure that all Commits are accepted and forwarded strictly in-order within an MLS group. [TODO: add reference to the language in section 4.2.1 of mls-architecture saying that this protocol requires Strongly Consistent ordering.]

The response is a 200 OK with no body. [Question: does the epoch need to be returned?? No, the Commit contained the epoch.] [TODO: you need a response code which clearly indicates that the Commit had the wrong epoch.]

Beyond that, the commit message is then distributed to participants using the synchronization protocol to drive MLS operations.

9. Synchronization

The second major piece of functionality provided by mimi is a synchronization function. This function allows a guest domain to receive changes to the group chats and connections in real time, so it can handle them and process accordingly.

A guest domain MUST maintain a list of the group chats and connections for which its users have an active membership, which are in a different owning provider. This list is potentially very long - millions of group chats might exist. To subscribe to changes, the domain utilizes a set of subscriptions using an HTTP long poll. Each subscription is to one or more group chats or one or more connections.

To subscribe to events for a group chat:

```
POST https://{provider-name}/.well-known/mimi/
      group-chats/{groupChatId}/events
```

and for connections:

```
POST https://{provider-name}/.well-known/mimi/
      connections/{ConnectionId}/events
```

Both requests take two optional parameters - "from" and "to" which contain timestamps. If the "to" parameter is absent, it means that the subscription should continue to deliver new events as they happen. If the "from" is absent. it means taht the subscription is from the beginning of time, and thus generate all historical events.

The body of the response is a streaming JSON, representing an array of events.

This naming introduces an important concept - that of an event. An event often is associated with the posting of a message, and this will be the most common event. But, it is not the only event. There are also events for changes in group membership, changes in group properties, changes in user properties, MLS Welcome and Commit messages. Each event has a unique timestamp, different from all other events. This means that all events form an ordered stream, which can be subscribed to. When a guest provider has missed events, it can also subscribe to events within a range of time, to fill in missing parts of the event stream.

Each event in the stream is formatted as a JSON payload, with the following common format:

```
{
    "eventTimestamp" : "{timestamp}",
    "type" : "{eventType}",


}
```

The type conveys the type of event. Defined types are "message", "mls", "groupPropertyChange" and "userPropertyChange", "groupChatAddRequest", and so on. Depending on the type, there will be additional JSON properties, unique to that specific type. Details TBD.

10. Security Considerations

Security is an essential part of this specification. The following subsections consider different threat vectors and how they are countered by mimi.

10.1. Spam Prevention

The single most important consideration in the design of mimi, is ensuring that it doesnt become a vector for spam.

In traditional email, it is possible for a malicious organization to send unwanted messages to any user, at very low cost. MIMI layers in several layers of protections to help prevent this. Of course, no technology can completely block all spam.

10.1.1. Consent Based Protection

The first line of defense against this is the consent-based model used in mimi. Consider a malicious provider that acts as an owning provider for a particular group chat. In order for that provider to deliver messages to a target user with a given email address or mobile phone number, they cannot just send a message. This is because the guest provider associated with that user needs to actively pull messages for that group chat. To entice the target user to ask their guest provider to pull messages for this group chat, all of the following must happen:

1. A message, such as an email or text, must be sent to the target user with a mimi URI,

2. The target must click on the mimi URI,

3. The target must have a messaging app installed on their device, registered as a mimi handler

If all three conditions are met, it would be possible for a malicious provider to cause unwanted chat messages to appear to the target.

Clearly, it is possible for the malicious provider to send an email or text to the target with a mimi URI. In mimi, valid emails are sent by the email address and/or mobile number of the actual user,

not a system address. Emails from malicious providers are likely to come from a system address. As such, they are likely to be caught by normal email spam filters and by SMS filtering techniques used by mobile operators. In essence, MIMI "inherits" the existing network of anti-spam and SMS filtering techniques in place today. This helps reduce the likelihood of the malicious email getting through.

Assuming it does get through, the target must be enticed to click on the MIMI URL. This is important. In mimi, it takes an active action on behalf of the recipient, to receive messages as part of a group chat. This is unlike spam emails or texts, where no action is required for the recipient to receive the unwanted message. Mimi allows users to differentiate a legiminate email or tet with a mimi URL, from an unwanted one. The main differentiator is the source of the message. In mimi, the text or email messages will be delivered using the mobile number or email address of the user generating the invite. Often, this will be someone known to the recipient, and thus match an entry in the user's mobile or email address book. In those cases, it is easy for the user to confirm that the request is legitimate. In cases where the inviting user is not known to the recipient, the email address or mobile number may not match an address book entry. In those cases, the invited user can respond and request further information before accepting the invitation.

It is also the case that, over the past many years, many users have been trained to not click on links in unsolicited communications. As a result, one would expect the click rates on these links to be low, reducing the chance of the user accepting the invitation.

As a final protection (albeit a small one), the receiving user must have a messaging app installed on their device that is compliant to mimi. If they do not, the invitation to chat will not be accepted.

10.1.2. Leave the Group

In the case where a user is tricked into joining a group chat that delivers them unwanted actions - the user has a recourse - they can always leave the group. [NOTE: This is actually really hard in MLS.] This is in contrast to email, where a user has no way to opt-out of communications. With persistent group chat, they can always leave. Beacuse mimi is based on a pull architecture, once a user leaves a group by asking their guest provider to do so, the guest provider ceases to synchronize (and thus receive) messages for that group.

10.1.3. Malicious Sender Traceability

Owing to the usage of MLS, all senders of chat messages are authenticated. This means the sender's original email address or mobile phone number will be signed. If the signature is provided by the malicious provider themselves, this is perhaps small consolation. However, should mimi make use of identities verified with a different, trusted third party identity source, this will provide good identification of the originator of the unwanted communications.

10.1.4. Provider Authorization

In email, any domain is allowed to send email to any other domain. This has allowed email to be a global messaging fabric. But, its completely open nature have made it trivial for a malicious entity to spin up as many domains as it wants, and send email from any domain.

In mimi, we do not expect this to be the case. In mimi, each provider needs to explicitly opt-in to the set of other providers whose mimi connections they will accept, or connect to. We anticipate that the larger providers in particular will implement quite strict policies on which other providers they connect to. Such connections are likely to be accompanied by significant offline vetting of the providers, to ensure they are legitimate, and provide human contact information that can be used to trace back in case malicious messages start to be sent from that provider.

10.2. Malicious Providers Manipulating other Users

Consider a group chat with owning provider A, and guest providers B and C. The group chat consists of users A1, B1 and C1 from their respective providers.

In this configuration, malicious provider C might attempt to manipulate the state of the group chat, by removing users A1 or B1. This attack is prevented by mimi. As part of the authorization checks performed by owning provider A, a provider is only permitted to request that a user leave the chat, if the user in question is a member of that provider. This check is done explicitly to prevent this attack.

There are other similar attacks, wherein provider C tries to post a message as if it came from user B1 or A1. These attacks are prevented in mimi with this same mechanism, but are also prevented by MLS. This provides two layers of security for those kinds of attacks.

11. IANA Considerations

11.1. URI Scheme Registration

This specification registers the mimi URI syntax as required by [RFC7595].

Scheme Name: mimi

Status: Permanent

Applications/protocols that use this scheme name: (NOTE TO RFC-EDITOR: Replace with the RFC number for this specification)

Contact: Jonathan Rosenberg, jdrosen@jdrosen.net (mailto:jdrosen@jdrosen.net)

Change Controller: IETF

References: (NOTE TO RFC-EDITOR: Replace with the RFC number for this specification)

11.2. Well-Known URI Registration

This specification registers the mimi well-known URI as required by [RFC5785].

URI Suffix: "mimi"

Change Controller: "IETF"

Specification Document: (NOTE TO RFC-EDITOR: Replace with the RFC number for this specification)

11.3. MIMI Property Registry

[TODO: can you please provide an existence proof of a property that shows we needs this?]

This specification establishes a new registry with IANA for MIMI properties. This registry contains the following parameters:

Property Name: the name of the property being registered

Context: The context is "user" for properties that apply to users only, "groupchat" for properties that apply to the group chat only, or "both" for properties which are defined for both.

Type: The syntax for the property. This must be one of string, enumerated string, boolean, or integer. [OPEN ISSUE: should we just make these all string and require the spec to define the type]

Governing Authority: This can either be a recognized standards body, such as the IETF, or a private registration.

Governing Specification: A link to the document defining the property. This can either be an IETF RFC, a specification from another standards body, or a document published somewhere on the Internet that contains information on the property.

Contact: Name and email address of the contact for the registration

{backmatter} This document instructs IANA to accept all registrations coming through the IETF standards process, for which an RFC performs the registration. For all others, expert review is required.

12. Normative References

[I-D.ietf-mls-architecture]
Beurdouche, B., Rescorla, E., Omara, E., Inguva, S., and A. Duric, "The Messaging Layer Security (MLS) Architecture", Work in Progress, Internet-Draft, draft-ietf-mls-architecture-10, 16 December 2022, <https://datatracker.ietf.org/doc/html/draft-ietf-mls-architecture-10>.

[I-D.ietf-mls-protocol]
Barnes, R., Beurdouche, B., Robert, R., Millican, J., Omara, E., and K. Cohn-Gordon, "The Messaging Layer Security (MLS) Protocol", Work in Progress, Internet-Draft, draft-ietf-mls-protocol-17, 19 December 2022, <https://datatracker.ietf.org/doc/html/draft-ietf-mls-protocol-17>.

[RFC2119] Bradner, S., "Key words for use in RFCs to Indicate Requirement Levels", BCP 14, RFC 2119, DOI 10.17487/RFC2119, March 1997, <https://www.rfc-editor.org/info/rfc2119>.

[RFC2396] Berners-Lee, T., Fielding, R., and L. Masinter, "Uniform Resource Identifiers (URI): Generic Syntax", RFC 2396, DOI 10.17487/RFC2396, August 1998, <https://www.rfc-editor.org/info/rfc2396>.

[RFC2717] Petke, R. and I. King, "Registration Procedures for URL Scheme Names", RFC 2717, DOI 10.17487/RFC2717, November 1999, <https://www.rfc-editor.org/info/rfc2717>.

[RFC5234] Crocker, D., Ed. and P. Overell, "Augmented BNF for Syntax Specifications: ABNF", STD 68, RFC 5234, DOI 10.17487/RFC5234, January 2008, <https://www.rfc-editor.org/info/rfc5234>.

[RFC5785] Nottingham, M. and E. Hammer-Lahav, "Defining Well-Known Uniform Resource Identifiers (URIs)", RFC 5785, DOI 10.17487/RFC5785, April 2010, <https://www.rfc-editor.org/info/rfc5785>.

[RFC6120] Saint-Andre, P., "Extensible Messaging and Presence Protocol (XMPP): Core", RFC 6120, DOI 10.17487/RFC6120, March 2011, <https://www.rfc-editor.org/info/rfc6120>.

[RFC6121] Saint-Andre, P., "Extensible Messaging and Presence Protocol (XMPP): Instant Messaging and Presence", RFC 6121, DOI 10.17487/RFC6121, March 2011, <https://www.rfc-editor.org/info/rfc6121>.

[RFC6914] Rosenberg, J., "SIMPLE Made Simple: An Overview of the IETF Specifications for Instant Messaging and Presence Using the Session Initiation Protocol (SIP)", RFC 6914, DOI 10.17487/RFC6914, April 2013, <https://www.rfc-editor.org/info/rfc6914>.

[RFC7595] Thaler, D., Ed., Hansen, T., and T. Hardie, "Guidelines and Registration Procedures for URI Schemes", BCP 35, RFC 7595, DOI 10.17487/RFC7595, June 2015, <https://www.rfc-editor.org/info/rfc7595>.

Authors' Addresses

Jonathan Rosenberg
Five9
Email: jdrosen@jdrosen.net

Cullen Jennings
Cisco
Email: fluffy@iii.ca

Suhas Nandakumar
Cisco

Email: snandaku@cisco.com