MATRIX PROTOCOL

Interoperable, Decentralised, Real-Time communication over IP

by

Balan Marius-Alexandru
Craciun Ioan-Paul

Instructor: Lenuta Alboaie
Faculty: Faculty of Computer Science, Iasi

Cover: Matrix.org
Style: TU Delft Report Style, with modifications by Daan Zwaneveld

Summary

The aim of this report is to act as an overview and introduction on Matrix, a relatively new open standard for interoperable, decentralised, real-time communication over IP.
We hope that by reading this report to the end, the reader may be able to understand the ideeas behind the Matrix protocol, what the people behind it are trying to achieve, how it works on a surface level and be able to discern when and how they may use it.

Contents

Summary **i**

1 Introduction **1**

2 Architecture Overview **2**

2.1 Clients 2

2.2 Homeservers 2

2.3 Events 2

2.4 Rooms 3

2.5 Bridges 4

2.6 APIs and Matrix Specification 4

3 Advantages of Matrix **6**

3.1 Decentralisation 6

3.1.1 Client-Server API 6

3.1.2 Server-Server API 7

3.1.3 Queries 7

3.1.4 Ephemeral Data Units 7

3.1.5 Persistent Data Units 7

3.2 Interoperability 8

4 Conclusion **9**

1

Introduction

Matrix is an open standard and communication protocol for real-time communication, with the aim of providing a functionality similar to that offered by SMTP (Simple Mail Transfer Protocol). This means that the same way that you have a Gmail account and are able to receive mails from a Yahoo account, through Matrix, you should be able to receive messages from all the applications that you are using, such as WhatsApp, Telegram, Messenger and so on, all in one place.

What we just explained, in simpler terms, is what Matrix defines as interoperability. It is designed to interoperate with other communication systems, and it does so through the use of bridges. A bridge, as the name suggests, is the component that handles bridging messages from different applications into the Matrix environment.

To this extent, one of the advantages of the protocol is that it is Open Standard, meaning that developers are able to see and understand how to interoperate with it. While bridges to Slack, IRC, XMPP and Gitter were implemented and are being managed by the Matrix team themselves, bridges to other applications, such as the previously mentioned WhatsApp and Telegram are provided by the community.

As far as decentralization is concerned, there is no central point. To send or receive messages, you have to connect to a server, called homeserver. Whether you want to create your own or use an already existing one, they mainly work the same: the homeserver you connected to communicates with other homeservers and each one of them maintains a copy of the conversation. This means that even if one homeserver were to fall or be shut down, as long as there were multiple ones connected, the conversation will not be lost.

2

Architecture Overview

2.1. Clients

To connect to the Matrix federation, you will use a client, which in turn will connect to a server. There exist clients made available for everyone, the most popular being Element, but you are also able to make your own. Matrix offers HTTP APIs and SDKs for iOS, Android and Web, so that one is able to create his own software. Furthermore, you may also choose to expand on a pre-existing one. Element for example have their github repository public, allowing those interested to make their own local copy and work on it.

2.2. Homeservers

An user may run one or multiple clients, each connecting through to a Matrix homeserver. The homeserver stores their account information and conversation history.
When users connected to different servers take part in a conversation, the servers may communicate with one another and share this information, each having a copy of it, with no one server being the owner of it.

2.3. Events

All communication done in the Matrix environment, be it client-server or server-server, is done through the use of events. Each action, such as sending a message, is correlated to an event.
Events are JSON object and users are able to define their own events, however they must adhere to the Matrix specification. For example, each event must have a `type` key which is used to differentiate the types of events and consequently what data is being sent/received. To create their own events, users must follow the Java package naming conventions. The `m.` prefix is reserved, as it is used by those at matrix to define the events implemented by them.
Every event shared in a room is stored in a directed acyclic graph, called the event graph. This allows the servers, which hold the room information, to order these events chronologically. An event that was sent has no other event preceding it, meaning that it is the most recent one and it has one ore more

parent, in the case that multiple homeservers race when sending events.

These parents are the most recent messages or the most recent event that was sent to the room at the time of sending the current event. When the server receives the event, it should be able to figure out to which node in the event graph should the received one be attached. To ease this process, homeservers also maintain a `depth` metadata field for each event. It is a positive integer that is strictly greater then the depths of its parents.

2.4. Rooms

When we talk about conversation in the context of Matrix protocol, we are actually talking about rooms. Rooms have much importance as to how Matrix functions and things that are allowed inside them are strictly controlled by rules. They may also contain different algorithms to handle different tasks.

In simpler terms, conversing between users, whether it is a direct message or a group chat, it is done inside rooms. When a room receives an event, all the users that are part of the conversation, meaning that they are in the room, will receive that event.

```
{ @alice:matrix.org }                             { @bob:example.org }
        |                                                 ^
        |                                                 |
[HTTP POST]                                  [HTTP GET]
Room ID: !qporfwt:matrix.org                 Room ID: !qporfwt:matrix.org
Event type: m.room.message                   Event type: m.room.message
Content: { JSON object }                     Content: { JSON object }
        |                                                 |
        V                                                 |
+------------------+                          +------------------+
|   homeserver     |                          |   homeserver     |
|   matrix.org     |                          |   example.org    |
+------------------+                          +------------------+
        |                                                 ^
        |         [HTTP PUT]                              |
        |         Room ID: !qporfwt:matrix.org            |
        |         Event type: m.room.message              |
        |         Content: { JSON object }                |
        `-------> Pointer to the preceding message  ------`
                  PKI signature from matrix.org
                  Transaction-layer metadata
                  PKI Authorization header

              ....................................
             |           Shared Data              |
             | State:                             |
             |   Room ID: !qporfwt:matrix.org     |
             |   Servers: matrix.org, example.org |
             |   Members:                         |
             |    - @alice:matrix.org             |
             |    - @bob:example.org              |
             | Messages:                          |
             |   - @alice:matrix.org              |
             |     Content: { JSON object }       |
             |....................................|
```

The above figure shows how room message event is sent to the room `!qporfwt:matrix.org`. The Shared Data structure is maintained on both servers by the federation and the data is split into message events and state events.

- **State events:** When the state of the conversation has been modified, for example the number of participating users or number of participating servers. These also come with a `type` and a `state_key` one. If a state event with the same `type` and `state_key` already exists, then it will be updated to the new value.
- **Message events:** When a message is sent or when making a call. They describe communication activity in general,

The room state at some point in time is calculated by taking a given event and then all the events preceding it. In the case of a conflict between states, Matrix is able to handle their merge, so if one wants to make his own implementation, he does not have to worry about that aspect. This algorithm is called the ”state resolution algorithm”.

2.5. Bridges

Bridges are what covoer the concept of interoperability in Matrix, making it open to exchange data with other platforms. They basically allow you to bring the conversations from different applications all in one place. What needs to be noted is that the capabilities of what you can do when bridging to a remote network is often limited by regulations imposed on their side. There are multiple approaches to bridging: **bridgebot-based bridges**, **bot-API based bridges** and **puppeted bridges**. Bridgebot-based bridges are the simplest ones, as they use predefined bots to relay the traffic, however they are not able to transport metadata at all, so all the informaiton is lost.
Bot-API based bridges, also referred to ad Virtual user based bridges, are one step above bridgebot-based ones. As the name implies, they use bots as well, but this time they act like virtual users. On the matrix side of things, when you connect to a conversation on a remote network, on another app, a new room is created where the participants are you and one or more bots, depending on if it is a direct message or a group conversation. Matrix will display that bot with the same nickname as the user that messaged you on the remote network. On the remote network's side, a bot is also created to represent yourself, but its capabilities are limited by the fact that it is virtual. Let us take Slack for example, a Matrix virtual user may not have a profile or be direct-messaged, among other things.
The first two methods of bridging that were describe work without needing to properly log in to the remote network, but puppeted bridges on the other hand require that you have an existing account on the network. They log into it as if they were a 3rd party client for the network and Matrix users may control their remote user, as if it were a puppet. All of the features of the remote network are available through this method of bridging and other users on the network are not aware when they are speaking to an user that is actually connected through Matrix.

2.6. APIs and Matrix Specification

Matrix offers a number of APIs for decentralised communication, for different uses. They can be used to persist, publish and subscribe to data, in a decentralised environment, meaning that there is no single point of control and no single point of failure. The APIs that they offer to be able to do this are the following:

- **Client-Server API**
- **Server-Server API**
- **Application Service API**
- **Identity Service API**
- **Push Gateway API**
- **Room Versions**
- **Appendices**

More about some of this APIs will be discussed in the following chapter as their features and the logic behind them are necessary to be talked about when explaining more about some of the concepts that are touched upon.

3

Advantages of Matrix

In this chapter we will talk about the reasons why one would choose to use Matrix and how Matrix achieves these results. Some of the facts may have already been mentioned while talking about the Matrix architecture, but they will be reiterated here also, with additional details, the focus being on them this time.

3.1. Decentralisation

We have already talked about decentralisation before and how it is achieved. Each homeserver that participates in a room keeps a copy of the conversation so that no one has full control of the data. Additionally, in the event of a server failure, data can be recovered from one of the homeservers that were still running during this time.

To understand how decentralisation is achieved, while still maintaining a synchronised state between what each users sees in their client, we will have a look at the Client-Server API and the Server-Server API and how they work.

3.1.1. Client-Server API

We have already talked about how they way that Matrix syncrhonises conversations is by keeping track and updating the event graph that represents the conversation history. When the client sends a new message, a new event is created. The server receives this event and then adds it to the graph so that it is properly ordered. The process of sending such an event is quite simple: you make a PUT request to the proper endpoint and give as request body the message and the sender, the complicated part being ordering and synchronising.

When a client connects to its homeserver, in order to synchronise itself, it should call on the `/sync` endpoint, which returns the most recent events for each room, as well as their states. One of the fields it returns is the `next_batch` field, which can be given as a parameter to `/sync`, so that it would return only updates from that point onward. It also includes a `prev_batch` field, that can be used on another endpoint to return earlier messages.

To continue receiving events, the clients can do this by long-polling to the homeserver, using the `next_batch` that was just mentioned. The server will holde the connection to the client open for a

period of time, period which can be modified by using the passing a `timeout` parameter.
An example of how this process happens can be viewed as follows:

```
        [E0]->[E1]->[E2]->[E3]->[E4]->[E5]
                    ^                     ^
                    |                     |
              prev_batch: '1-2-3'    next_batch: 'a-b-c'
```

While the event graph actually starts with event [E0], the request only returns events [E2] to [E5]. By long-polling, we are notified when a new event is added and we receive another response that may look as follows:

```
        [E0]->[E1]->[E2]->[E3]->[E4]->[E5]->[E6]
                                          ^     ^
                                          |     |
                                          |  next_batch: 'x-y-z'
                                        prev_batch: 'a-b-c'
```

3.1.2. Server-Server API

While the Client-Server API focuses on making sure that a client connected to a homeserver is synchronised to the state of the room that is stored on the server, the Server-Server API makes sure that every homeserver taking part in this room has a consistent room state.
When a client sends a new event, it is first added to the event graph on the homeserver that it is connected to. This event is then propagated to the other homeservers, which then send the event to their active clients. It is interesting to see how Matrix handles cases when homeservers race with one another, meaning that two clients sent events and now the two homeservers that they are each connected to have different event graphs.
To better understand how Matrix handles this problem, we will first have a look at how communicatino between homeservers takes place. There are 3 categories for it: **Persistent Data Units (PDUs)**, **Ephemeral Data Units (EDUs)** and **Queries**.

3.1.3. Queries

Queries are single request/response interactions between a given pair of servers. It is initiated by one of the servers sending an HTTPS GET request to retrieve information from the other about an user or a room.
The the requests are often made in conjunction with a request on the Client-Server API, sent from the client, so that the Client-Server API is able to complete the call.

3.1.4. Ephemeral Data Units

These events are pushed between pairs of homeservers. As the name implies, they are not persistent and do not remain the the history of a room. Also, there is no need to reply to these events.
EDUs, as opposed to PDUs, do no have an ID, a room ID or fields such as the `prev_events`. They are mainly used for things like typing notifiactions, user presence and so on. They simply request a snapshot state at the instant the query is made.

3.1.5. Persistent Data Units

PDUs are the events that are sent to each of the homeservers participating in a conversation. They are persistent and record the message history and state of the room.

Each PDU contains a single event that needs to be sent to the destination. The JSON object sent has a `prev_event` field that identifies the parent or parents of an event. Through this field the partial ordering of the event graph is established. It should be populated with all the events that have no child, meaning that they are the latest known events.

Homeservers also maintain a metadata field named `depth` that helps in chronological ordering and comparison of the events within the graph. The field must be a positive integer, strictly greater than the depths of its parents and the root should have `depth` 1.

```
            { Matrix client A }                            { Matrix client B }
                ^          |                                   ^          |
                |  events  |  Client-Server API                |  events  |
                |          V                                   |          V
            +------------------+                           +------------------+
            |                  |---------( HTTPS )--------->|                  |
            |   homeserver     |                           |   homeserver     |
            |                  |<--------( HTTPS )---------|                  |
            +------------------+     Server-Server API     +------------------+
                                 History Synchronisation
                                     (Federation)
```

3.2. Interoperability

As we have already touched on before, Bridges are what offer interoperability in Matrix and also about the different types of bridges. They basically allow you to bring the conversations from different applications all in one place, however what needs to be noted is that the capabilities of what you can do when bridging to a remote network is often limited by regulations imposed on their side.

We have talked about different the different kinds of bridging and now we are going to talk about how the messages are actually passed around.

Bridges are constructed using the Application Service API, also called the Bridge API, which offers a way to implement consistent self-contained federated messaging fabrics.

Each time a homeserver receives an event, this event will be forwarded to the Application Service API, which then facilitates forwarding it further to the remote network.

When considering the round trip, the Application Service API uses an API similar to the Client-Server one. It works in a similar fashion the that API, however it is also able to create virtual users and mask them correspondingly to the person that sent the message from the remote network.

Looking at the implementation side of things, the part that is inside the Matrix environment, sending and receiving messages to and from the Application Service API, is not so difficult since Matrix already has implemented this kinds of interactions. What is more difficult comes to the remote network side of things: figuring out how to inject the messages received from Matrix into it and how to get new messages from other users and forward it to Matrix. This however may differ from one remote network to another, as it mostly depends on their architecture.

Let us look at the Gitter bridge for example. To be able to send a message to Matrix as soon as it would normally be received in the main app, a webhook was implemented on the database. So whenever a change is made in the database, the notification will also be taken by the webhook and forwarded. To display received messages, the `virtualUser` field was added to the chat message schema and whenever there is a virtual user, the information from it will be used instead of the normal author.

4

Conclusion

This concludes our overview of the Matrix protocol. We tried to highlight the core functionalities of Matrix on a surface level, by offering an overview of the Matrix architecture by looking at its core concepts. We also tried to offer an understanding of how these concepts that make Matrix so attractive, decentralisation and interoperability work.

The need for a decentralised web seems to only grow bigger by the day, as more and more privacy concerns come to the surface when all the date is held into one place. In the same manner, having all this data in one place gives too much power to the one who holds it, as we see cases of misuse of personal data, as well as imposed censorship without reason.

Matrix may prove to be the solution to this problem. As it reached stable versions, we see Matrix being adopted by more and more companies and entities, such as Mozilla and the Government of France.

One of the most attractive aspects of Matrix remain the bridges and the ability they give to interoperate with other applications. While actual implementations do satisfy most of the needs that one would have while trying to bridge together two applications, they are still limited by the remote network side and the restrictions they impose.

The next level of puppeted bridges, Double-puppeted Bridges, would take care of these issues. The ideea is that puppeteering can be done from both sides of the conversation, from Matrix and from the remote network alike. This would imply that both the Matrix account and the third party account are accurately represented, with their metadata intact. There are projects that work towards a solution to this, such as the matrix-puppet-bridge community project, but they still present impediments, such as requiring the user to run his own server.