MATRIX PROTOCOL

Interoperable, Decentralised, Real-Time communication over IP

Concurrent and Distributed Programming

Balan Marius-Alexandru
Craciun Ioan-Paul

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

Preface

A preface...

Might delete this page later

Balan Marius-Alexandru
Craciun Ioan-Paul
Delft, April 2022

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

Nomenclature

If a nomenclature is required, a simple template can be found below for convenience. Feel free to use, adapt or completely remove.

Abbreviations

Abbreviation	Definition
ISA	International Standard Atmosphere
...	

Symbols

Symbol	Definition	Unit
V	Velocity	[m/s]
...		
ρ	Density	[kg/m^3]
...		

1 Introduction

An introduction... [1]

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

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 objects and each must respect the Matrix specification. For example, each event has a `type` key which is used to differentiate the types of events and consequently what data is being sent/received. To this extent, users are able to define their own events, but they 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. Conversing between users 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.

- **Message events:** When a message is sent or when making a call. They describe communication activity in general,

2.5. Bridges

Bridges are what covoer the concept of interoperability in Matrix, making it open to exchange date with other platforms. There are multiple approaches to bridging: **bridgebot-based bridges**, **bot-API based bridges**, **simple puppeted bridges** and **double-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.

2.6. APIs

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**

The two main components of Matrix are the clients and the servers. There are more than just these two, however they are the ones that handle the main function, that is: messaging, storing and synchronizing. A client communicates with other clients by synchronising the conversation with their homeserver, this is done using the Client-Server API. The homeserver in turn synchronizes this conversation to that on the servers that the other clients are connected to.
The other APIs offer functionalities related to custom behaviour, notifications and more, and we shall discuss about them in turn.

Client-Server API

Supports both lightweight clients, which store no data about the server state or lazy-load when requires and heavyweight clients, which maintain a local persistent copy of it. Communication between a client and its server is done through JSON objects. HTTPS is recommended, but HTTP is supported as a fallback for basic clients.
As we mentioned before, the way that Matrix synchronises conversation is by keeping track and updating the conversation history. The way that the client-server API handles it may be looked at as a series of events. The server receives these events and must be able to order them in order of precedence, creating an ordered event graph.

The events created by the Matrix team all have the `m.` prefix in the type key and users are free to add and define their own events. To do this however, they must follow the REST API presented in the Matrix specification. The following two tables represent the format typically used for events returned from a homeserver to a client. Most events will abide by this format and what differs from one to another is actually the content.

Name	**Type**	**Description**
content	object	Required: The body of this event, as created by the client which sent it.
`event_id`	string	Required: The globally unique identifier for this event.
`origin_server_ts`	integer	Required: Timestamp (in milliseconds since the unix epoch) on originating homeserver when this event was sent.
`room_id`	string	Required: The ID of the room associated with this event.
sender	string	Required: Contains the fully-qualified ID of the user who sent this event.
`state_key`	string	Present if, and only if, this event is a state event. The key making this piece of state unique in the room. Note that it is often an empty string.
type	string	Required: The type of the event.
unsigned	UnsignedData	Contains optional extra information about the event.

Name	**Type**	**Description**
age	integer	The time in milliseconds that has elapsed since the event was sent. This field is generated by the local homeserver, and may be incorrect if the local time on at least one of the two servers is out of sync, which can cause the age to either be negative or greater than it actually is.
`prev_content`	EventContent	The previous content for this event. This field is generated by the local homeserver, and is only returned if the event is a state event, and the client has permission to see the previous content.
`redacted_because`	ClientEvent	The event that redacted this event, if any.
`transaction_id`	string	The client-supplied transaction ID

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. It also returns a `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.

2.7. Server-Server API

Also known as the Federation API, it is used by homeservers to communicate with one another. It is used to push messages in real-time, to retrieve conversation histories and to get information about users and their profiles, when they are connected to different servers.
Similar to how it is for Client-Server API, the Server-Server API also uses HTTPS to make request between servers. The requests are authenticated using public key signatures at the TLS transport

layer and using public key signatures in HTTP Authorization headers at the HTTP layer.
The communication between homeservers can be divided into 3 categories: **Persistent Data Units (PDUs)**, **Ephemeral Data Units (EDUs)** and **Queries**.

2.7.1. Persistent Data Units

Users from connected to multiple servers may take part in the same conversation. At that point, PDUs are the events that are sent to each of the participating homeserver. They are persistent and record the message history and state of the room.
While it is the responsability of the sender to deliver the PDU, the PDUs are also signed using the sender's private key, making it possible to leave the process of delivering the message to a third-party server.
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, in the case that multiple homeservers race when sending messages, which are used to achieve the ordering that we talked previously in the case of the Client-Server API. To this extent, the field should be populated with all the events that have no child, meaning that they are the latest known events.
A server that is on the receiving end of a PDU must perform some checks, for each one that it receives:

- Validity check
- Signature check
- Hash check
- Authorization rules check

2.7.2. 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 are not persisted and contain no long-term significant history. They simply request a snapshot state at the instant the query is made.

2.7.3. 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

Conclusion

A conclusion...

References

[1] I. Surname, I. Surname, and I. Surname. “The Title of the Article”. In: *The Title of the Journal* 1.2 (2000), pp. 123–456.

A
Source Code Example

Adding source code to your report/thesis is supported with the package `listings`*. An example can be found below. Files can be added using* `\lstinputlisting[language=<language>]{<filename>}`*.*

```python
"""
ISA Calculator: import the function, specify the height and it will return a
list in the following format: [Temperature,Density,Pressure,Speed of Sound].
Note that there is no check to see if the maximum altitude is reached.
"""

import math
g0 = 9.80665
R = 287.0
layer1 = [0, 288.15, 101325.0]
alt = [0,11000,20000,32000,47000,51000,71000,86000]
a = [-.0065,0,.0010,.0028,0,-.0028,-.0020]

def atmosphere(h):
    for i in range(0,len(alt)-1):
        if h >= alt[i]:
            layer0 = layer1[:]
            layer1[0] = min(h,alt[i+1])
            if a[i] != 0:
                layer1[1] = layer0[1] + a[i]*(layer1[0]-layer0[0])
                layer1[2] = layer0[2] * (layer1[1]/layer0[1])**(-g0/(a[i]*R))
            else:
                layer1[2] = layer0[2]*math.exp((-g0/(R*layer1[1]))*(layer1[0]-layer0[0]))
    return [layer1[1],layer1[2]/(R*layer1[1]),layer1[2],math.sqrt(1.4*R*layer1[1])]
```

B Task Division Example

If a task division is required, a simple template can be found below for convenience. Feel free to use, adapt or completely remove.

Table B.1: Distribution of the workload

	Task	Student Name(s)
	Summary	
Chapter 1	Introduction	
Chapter 2		
Chapter 3		
Chapter *		
Chapter *	Conclusion	
	Editors	
	CAD and Figures	
	Document Design and Layout	