Experimenting with Matrix federation over Yggdrasil

Timothée Floure

School of Computer and Communication Sciences

Decentralized and Distributed Systems lab

Bachelor Semester Project

Fall 2019

Responsible
Prof. Bryan Ford
EPFL / DEDIS

Supervisor
Cristina Basescu
EPFL / DEDIS

Contents

1 Introduction

Matrix is a federated event-based communication protocol largely used for instant messaging, which appeared in 2014 and reached its first stable release during the first half of 2019[1]. It has been steadily gaining adoption - especially among free software groups - over the past few years, as its ecosystem supports many modern features such as media content, message history and synchronization, end-to-end encryption and easy integration with external services (audio and video call via Jitsi, bridging with other IM networks). While decentralized to some extend given its federated structure, Matrix heavily depends on global and centralized - both from technical and governance point of views - infrastructure for name addressing (DNS) and routing. [2]. The aim of this project was to develop a proof-of-concept of Matrix federation over a peer-to-peer system, which has been done with Yggdrasil, an experimental end-to-end encrypted peer-to-peer overlay network featuring routing logic akin to compact routing.

I chose to work on those topics out of personal interest in the Matrix ecosystem - which I use daily - and peer-to-peer systems, although I had limited if no prior experience on the later. Yggdrasil was chosen due to direct interest from one of their core developers to support this project.

2 Matrix

2.1 Introduction & origins

Matrix is a federated event-based communication protocol, atop of which is built a decentralized instant messaging (IM) system designated by the *Matrix* name itself. The project addresses some of the shortcomings of previous protocols such as IRC, and is gaining adoption in many free software communities including big names like KDE and Mozilla. It was also featured on FOSDEM's[3] main track in 2019.

Matrix also get some 'corporate' usage with actors such as the French state and Matrix-as-a-Service providers; academic interest with the A Glimpse of the Matrix : Scalability Issues of a New Message-Oriented Data Synchronization Middleware preprint and even here at EPFL/DEDIS with Matrix being used as a test-subject for decentralized authentication against the DEDIS blockchain[4].

[1] Introducing Matrix 1.0 and the Matrix.org Foundation, Matthew Hodgson

[2] .ORG TLD rights being sold to a for-profit, traffic being unencrypted by default and rather 'easily' hijacked

[3] Largest European FOSS conference.

[4] Project I discovered by discussing with Linux Gasser, ex-DEDIS engineer now working

The Matrix project is governed by the Matrix Foundation and largely ported by the New Vector company.

2.2 Matrix internals

The Matrix network, as other federated chat protocols like XMPP, is organized around homeservers: each user is linked to a server, which is used as a gateway to send and receive messages. A Matrix user ID (*MXID*), is constructed as *@username:domain.tld* where *domain.tld* is the domain name linked to the homeserver. The federation address of homeservers are discovered via DNS resolution, either from a direct A / AAAA record or from specific SRV record.

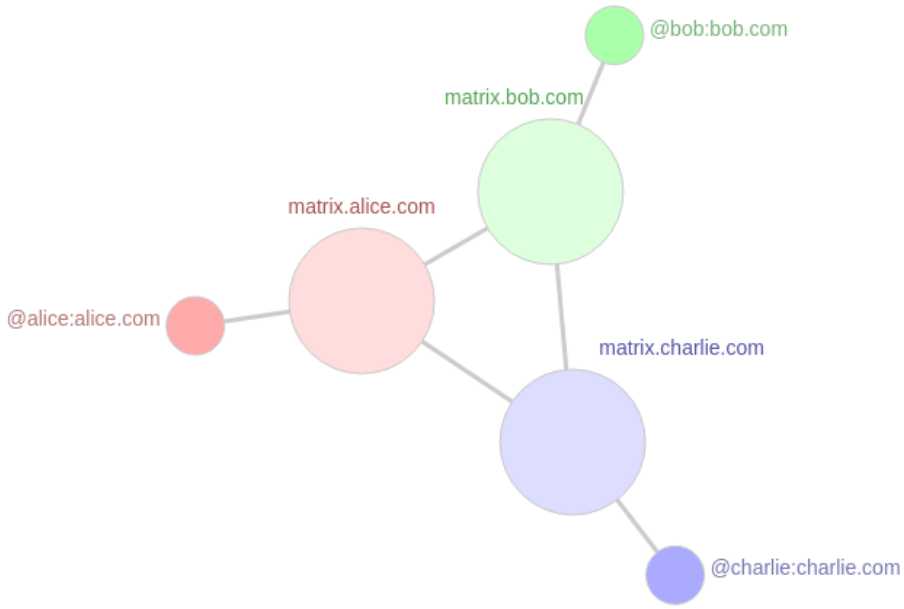

Figure 1: Three matrix homeservers, each with one connect client. Extracted from an interactive demonstration on matrix.org.

Matrix events are replicated on every server participating in a given conversation: even if one the server goes down, the others are still able to exchange messages. The basis of this scheme, which is critical to matrix, is the room state resolution algorithm defined in the room specification.

for EPFL's Center for Digital Trust, a few weeks ago.

2.3 Implementations and real-world-usage

The Matrix network is defined by a set of APIs (Client-Server, Server-Server, Application Service, Identity Service, Push Gateway) formalized in the Matrix Specification which is in turn maintained by the Matrix Foundation. There are many clients and a few server implementations, many of them still incomplete: my semester project makes use of the reference client (Riot) and server (Synapse). The wider Matrix ecosystem include many external components interfaced through the Application Service API, mostly 'bridges' to other networks such as IRC, Email, SMS, Mastodon, Telegram, Slack and others.

3 Yggdrasil

3.1 Introduction & origins

Yggdrasil is an experimental, decentralized, lightweigth, end-to-end encrypted and self-arranging overlay network providing cryptographically-bound addresses. Work on this project started in 2015 when the original author, Arceliar, moved away from CJDNS - a similar project - following the introduction of some level centralization in their network. More details on the history of Yggdrasil can be found in the related History of the World Tree, Part I article on the project's blog. A detailed description of Yggdrasil itself can be found in the Yggdrasil draft whitepaper.

Initial thoughts on the routing scheme were based on Thorup and Zwick's universal compact routing scheme but were eventually replaced by a logic inspired from Robert Kleinberg's Geographic Routing Using Hyperbolic Space, given that the author did not find a way to implement Thorup and Zwick's ideas securely for both a dynamic network and in a distributed way. Yggdrasil differentiates itself from Kleinberg's scheme by not embedding the spanning tree in the hyperbolic plane to prove there is no dead end, but by having nodes remember the path to the root instead. This difference is motivated by a reduction of complexity, which is a claim I did not challenge due to my lack of background on the topic.

3.2 Yggdrasil routing explained

An Yggdrasil node owns two key pairs: one for encryption and one for signing. The node's identity is derived from those keys as follow:

- A **NodeID** is the full address of node in the network. It is defined as sha512sum of the node public encryption key.

- The IPv6 address of a node is derived from its NodeID. This address is only used for compatibility with the IP world and is no more than an alias to its original NodeID.

- **TreeID**s are used to determine the root node of the network. They are defined as sha512sum of the node's public signing key.

Every Yggdrasil node is connected to the network by a set of peer manually defined by the node operator (there is no automatic peering system). A peering consist of a single TCP connection, used for all Yggdrasil traffic between the concerned nodes. While the direct peering connection between two nodes is reliable and ordered, the path between two arbitrary nodes - composed of multiple peering links - is not. When an arbitratry node A want to exchange with a node B, it will first have to initialize an end-to-end encrypted session with B. Such a session is unique, meaning that all (user/application) traffic must be multiplexed into it, and expire after a few minutes if unused.

Yggdrasil routing is based on coordinates derived from a minimal spanning tree (MSP). The root node (lowest TreeID) periodically sends signed advertisements which are forwarded through the whole network, each node selecting its 'parent' as shortest path (latency-wise) from the root node. The coordinates (or 'locator') of a node is its position in the MSP, which change over time. A NodeID to coordinates mapping is published in a Chord-like DHT for nodes to be able to contact each other.

In order to join a node given a NodeID (or IPv6 address, which is a subset of the NodeID), the node first resolve the current coordinates of the target from the DHT and attach it to the message, then 'greedily' forward it to a node strictly closer to its destination in the metric space generated by the coordinate system. Intermediate nodes only have to read the destination of the message and forward it to a peer which is closer to the destination. If no closer hop is available (anymore), the message is dropped.

The above routing schemes yield a strech of roughly 1.04, determined by observation (note: this is a claim from the Yggdrasil project, which I did not confirm myself). A strong point of this scheme is that each node only has to maintain its location, the location of its direct peers as well as a few DHT entries: this is rather low given than a typical node do not have many peers. Another interesting point is that nodes are not aware of the real destination of the traffic they forward but only of a temporary locator (i.e. current location in the MSP). Yggdrasil is also able to quickly adapt itself to changes in the network's topology.

3.3 Implementations and real-world usage

The initial Yggdrasil proof-of-concept has been written in Python but the current reference implementation is written in Go. I am not aware of any other usable implementation right now but a draft specification for Yggdrasil has been published in mid-fall 2019, which open the way to a broader ecosystem[5]. There is a single Yggdrasil network as of today.

The Yggdrasil Go implementation provides an API for use as a Go library (since Auguest 2019) as well as a TUN/TAP support for use as virtual network interface in Linux. There also is support for *BSD (including MacOS), Android, iOS and Microsoft Windows, although I am less or not familiar with these platforms. There currently is a few hundred nodes in the network, which can be explored with the yggdrasil-map tool:

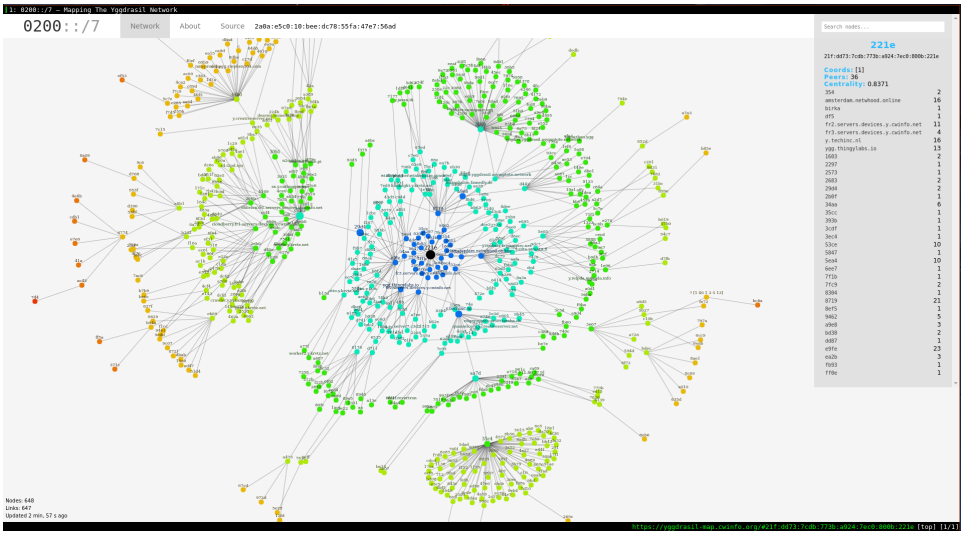

Figure 2: Representation of Yggdrasil's spanning tree, 2020-01-05, `https://yggdrasil-map.cwinfo.org/`

4 Matrix Federation over Yggdrasil

The goal of the project was originally discussed with Mathew Hodgson from the Matrix Foundation, which introduced Neil Alexander from Yggdrasil upstream. Although other overlay networks such as CJDNS or GNUNet could have been used, support from Yggdrasil upstream was the decisive factor in its choice. We also deemed that the project - getting a working proof-of-concept of Matrix federation over Yggdrasil - was good enough to fit the academic requirements of a semester project, which was accepted by

[5]Mastodon post of work-in-progress modern Python implementation of Yggdrasil.

Cristina Basescu from DEDIS.

It would have be easy to use the TUN/TAP support provided by Yggdrasil and have any matrix homeserver listen over the derived IPv6 address, yet this means tampering with the host system with high privileges and opening Yggdrasil connectivity to the whole system. This was obviously no good in term of security, did not allow for self-contained applications and did not make use of the recent Yggdrasil Go library which was hoping for some real-world usage.

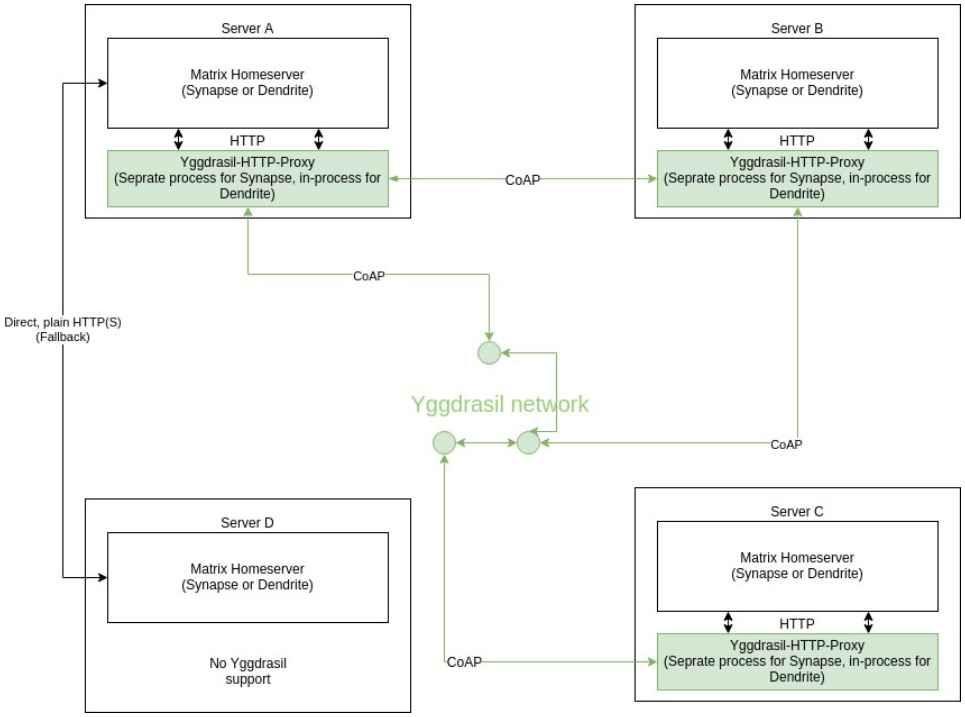

Figure 3: The 'big picture' of what I achieved. Green represents Yggdrasil-aware software or traffic over Yggdrasil.

4.1 Getting started

While I was somewhat familiar with Matrix and its internals, I had very little if no notions in decentralized systems such as Yggdrasil. I also never used the Go programming language before, although it was not an issue given that I was familiar with the concepts it uses. I would like to say that I fully documented myself beforehand, yet a lot of things became clear(er) to me along the way thanks to my supervisor's (Cristina Basescu) weekly feedback.

The first tools I built were dummy Yggdrasil nodes leveraging the recently published yggdrasil Go API. I encountered a few bugs along the way - such as a type incompatibility between ‘yggdrasil.Core.Address‘ and Go’s built-in ‘net.Go‘ - which were quickly fixed by upstream. This work was the starting point I hoped with both Yggdrasil and Go, and worked fine to exchange a few bytes over Yggdrasil.

4.2 HTTP over Yggdrasil

I soon wanted to transfer meaningful data on the link I created with the dummy node pair, and initially planned to write a custom go net/http transport to use my Yggdrasil session as a transport for HTTP exchanges. I soon figured that the ‘Conn‘ exposed by the Yggdrasil Go library was more akin to UDP than TCP since messages were unordered and not necessary delivered. It implied that using plain HTTP was out of question short of reimplementing TCP’s control layer myself: it made more sense to re-use an existing UDP-frienly implementation from the Go ecosystem. The candidates were: quic-go, kcp-go and go-coap: the later - implementing the Constrained Application Procotol - was what I believe to be a ’perfect’ fit:

- REST-like model akin to HTTP’s, making translation easier than with the others.
- Designed with low-bandwidth environment and unreliable network links in mind.
- Already been used in a Matrix.org low-bandwidth experiment one year ago (see below).
- Easy reuse of opened connections/multiplexing (remember that Yggdrasil only allows a single session between two nodes).

I have to admit that I did not investigate on the other two protocols in depth, given that CoAP was exactly what I wanted. They both were lower-level - and likely harder to use - than the CoAP and the QuiC implementation was not production-grade yet.

The low-bandwith experiment mentioned above was presented during the FOSDEM’19 conference in early 2019, during the Breaking the 100 bits per second barrier with Matrix talk, but did not see any work since. The experimental transport mechanism built was based on CoAP + CBOR (Concise Binary Object Representation) + Flate (compression) + Noise (encryption) + UDP, and lives in the matrix.org/coap-proxy github repository. Some of my work has been inspired from this code, although most of what I extracted has been rewritten from scratch since given that the original source was neither documented, modular or very well written. The ’noise’ encryption layer

was also irrelevant given that Yggdrasil natively provides end-to-end encryption.

4.2.1 CoAP as a translation layer

The next step was to add Yggdrasil support to the go-coap library, which has been successfully done on this fork. This was quite tedious given then even if go-coap already supported many underlying transports and was rather modular, it was not documented in any way. It took me some time, grepping, network debugging, RFC reading and a few pulled hairs to actually understand what was going on and plug Yggdrasil support into it. I also revamped my Yggdrasil toy nodes into an yggdrasil-go-coap example, extending documentation on Yggdrasil support for the go-coap fork.

CoAP over UDP is defined in RFC 7252 and guidelines for HTTP-CoAP mapping are available under RFC 8075. CoAP, being 'REST-like', has the concepts of method, body, and URL similar to HTTP's, which makes the translation easy. With some tweaking it is even possible to fit HTTP headers into CoAP options, which is what I did to transmit the X-Matrix-Authorization HTTP header (using an option ID from the experimental range).

4.3 Integration with Matrix homeservers

The original idea was to directly integrate Yggdrasil support in the Dendrite next-generation Matrix homeserver, which is also written in Go. After more discussion with dendrite's development team, it happen that its federation support is still incomplete and partially broken. I thus switched gears and went to use the reference python implementation - synapse - for my work. I ended up writing an HTTP-to-CoAP-and-back proxy since directly integrating the support in the homeserver codebase was not possible anymore (Yggdrasil only provides Go bindings). The produced tool, matrix-yggdrasil-http-proxy takes HTTP requests as input, convert to CoAP, transport over Ygdrasil, convert from CoAP to HTTP, send to local node and back for the answer.

This work required a few patches to the synapse homeserver in order to resolve Yggdrasil addresses from DNS entries and proxy federation requests. The changes took a lot of time to implement even if they were rather trivial, due to the size of the synapse project - more than 70K lines of python (number obtained with SLOCCount) and some home-made and undocumented python dependency injection scheme. The patches can be found in this

synapse github fork.

The proxy and CoAP library themselve required subsequent tuning to complete the integration, as I discovered some bug in yggdrasil-go/go-coap connection reuse and handling of large body sizes (a CoAP packet is usually no more than 1280 bytes, meaning that large HTTP requests must be split into multiple, smaller CoAP requests). Getting everything right for the Matrix event signatures to be valid also took some adjustments and code reading up to synapse's python-signedjson depency to actually understand the exact data structure representation that was being signed and verified.

4.3.1 Resolving Yggdrasil addresses

The federation address of a matrix homeserver is resolved from DNS entries on its linked domain name, which can be done in three different ways:

- From an SRV record matching '_matrix._tcp.domain.tld'. Example: 'matrix.tcp.gnugen.ch SRV IN 3300 matrix.gnugen.ch'.
- From a /.well-known/matrix/server (HTTPS) URI on a node resolved from an A or AAAA record.
- By directly using the node resolved by the A or AAAA record.

As mentioned earlier, Yggdrasil nodes addresses are either large NodeIDs or IPv6 addresses. I chose to directly resolve the NodeIDs given that the IPv6 address was not required, allowing to remove a translation layer. The homeserver itself could (and should!) already have a public IPv6 address, which would have opened the door for confusions. There is, by design, no naming system in the Yggdrasil network, as it is considered not to the job of the network itself but that of an higher-level abstraction layer. Although it is not a long-term solution (remember that we want to remove DNS from matrix, eventually), I fell back to standard DNS resolution and added support for a '_matrix._yggdrasil.domain.tld' SRV record in synapse, which is the simplest and likely best way to deal with address resolution at the moment.

4.4 Yggdrasil peer selection

While it is possible to add and remove peers at runtime, Yggdrasil does not provide automatic peering update. This means that peer selection and initial bootstrapping has to be handled by the application making use of Yggdrasil. A node requires no more than a single peer to particpe in the network, adding a more improving resilience and allowing to forward traffic

for others. Some guideline are provided in the Practical peering blog post but it is quite hard to prove a scheme efficient given that Yggdrasil does not provide much guarantees.

Another factor to take into account is that operator of Matrix homeserver might not want to forward traffic unrelated to Matrix, which is not possible with Yggdrasil right now. The best we can do at the moment is to preferably peer with other matrix homeserver, which can be easily discovered by watching the traffic going in and out of the proxy. The scheme I started to implement - and did not complete since I was out of time - tries to maintain an arbitrary number of peers (e.g. 5) by running a peer update algorithm periodically (e.g. every hour). This algorithm takes into account the latency of a link, how much bandwidth it exchanged (i.e. how used it is) as well has its age (i.e. how stable it is). The 'worst' peering will be forcefully replaced at each run in order to 'try' new peers and eventually converge with a low-latency and stable peer set. The pondering is not defined yet, given that this peer selection work is no more than a work-in-progress.

I plan bootstrapping - joining the network for the first time - to be done by selecting a few random peers out of a list of well-known nodes provided by some upstream (e.g. Matrix.org or Yggdrasil's public peer listing).

4.5 Results

I was able to build a working proof-of-concept, fulfilling the main goal of the project. All this work was as I hoped a great introduction to Matrix, Yggdrasil and distributed networking internals in general, of which I am very glad. It is, from my point of view, **a success**!

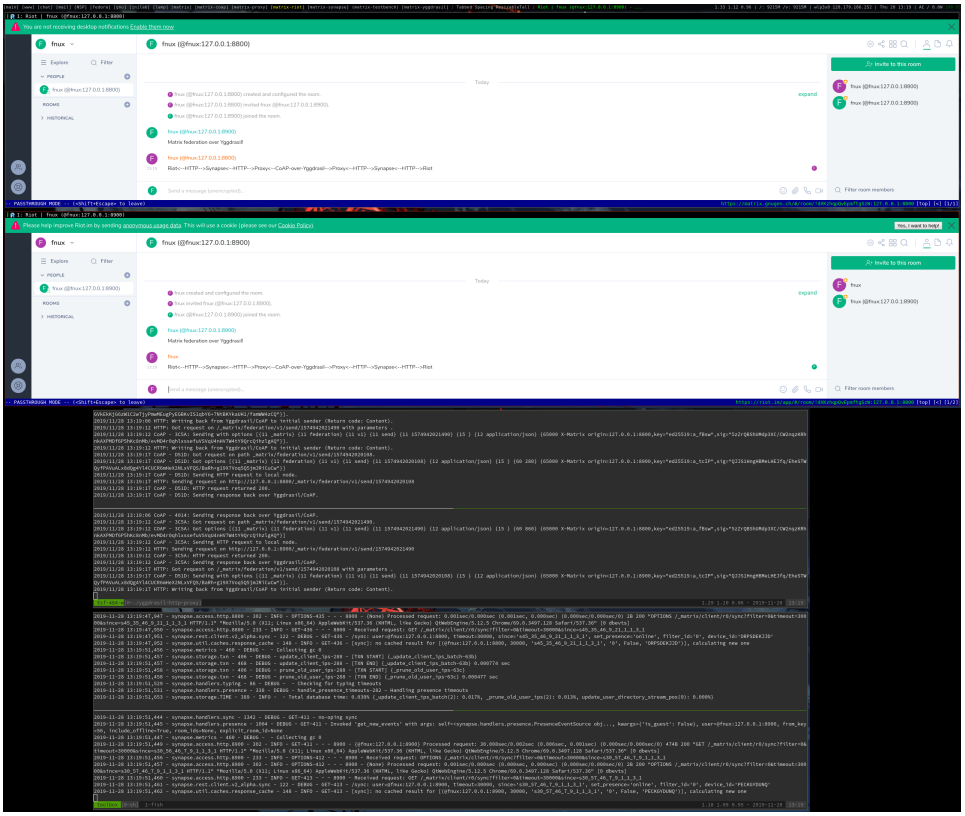

Figure 4: A working proof-of-concept! Screenshot of two matrix client, on different homeservers, discussing over Yggdrasil.

The tools I wrote still have some rough edges and did not undergo thoughtful testing or real-world usage, but are usable. Some of the shortcomings are detailed in the next section and I hope to address them after further discussion with Matrix upstream, which is likely to happen at FOSDEM in early February 2020.

There is not much data in term of performance or overhead compared to 'standard' federation given the early state of the tooling and lack of optimizations in order to keep the codebase as simple of possible and due to lack of time. A quick local (two Yggdrasil nodes peered together on my laptop) test measured a latency overhead of 25% on 1000 HTTP GET requests. No compression was implemented but there is a lot of potential in that domain given what have been done previously in the low-bandwith Matrix CoAP experiment in 2018.

4.6 Discussions & future work

4.6.1 Compression

The matrix-yggdrasil-http-proxy does not support compression but reusing previous work done by matrix.org on the low-bandwith experiment (which was also using a HTTP-to-CoAP proxy) should allow large improvement on the currently deployed plain HTTP transport. The mentioned experiment was able to use an average of 35x-70x less bandwidth than standard transport in a controlled environment, which would be especially be welcome if the CoAP proxy is extended to the Client-Server API which is often used from mobile devices.

4.6.2 Extended testing & real-world usage

This proof-of-concept would gain from large-scale testing and - if possible - real-world usage, which might happen depending on the direction taken by the Matrix project in the near future. Various P2P tests and experiments are taking place in the Matrix ecosystem and some or a mix of them will eventually be chosen for proper integration with the production environment. I expect to clarify this topic at FOSDEM'20 following the The Path to Peer-to-Peer Matrix talk (in which I expect this work to be mentioned) and live discussions with Matrix upstream.

It would also make sense to properly integrate this proxy with sytest, a black-box integration test tool for matrix homeserver.

4.6.3 Completing peer selection

As mentioned above, automatic peer selection is still a work-in-progress. Getting it done and be able to simulate/observe the generated peering graph would be very useful.

Regarding the ownership of forwarded Yggdrasil traffic, and not adding non-matrix load to Matrix operators, a solution could be to create a second Yggdrasil network exclusive to Matrix federation. It would be, however, hard if not impossible to stop malicious actor to piggy-bag on this new network given that all traffic is encrypted and thus cannot be filtered.

4.6.4 Homeserver discovery & name resolution

There is no addressing scheme shipped with Yggdrasil, which is a design choice in order to keep the tooling and logic as simple as possible (other

networks, such as GNUNet, provide their own). This also means that we fall back to the global DNS infrastructure, stopping us from killing two birds with one stone. I deployed a convenience helper generating 'easy to remember' names such as *academy-podium-medical.yggdrasil.ungleich.cloud* for any Yggdrasil node in the meantime, as remembering IPv6 addresses can be painful. This was quite convenient for testing purposes, but is more akin to a toy when one think about real-world usage.

Note that the Matrix project is experimenting with alternative to DNS, starting with the suppression of MXIDs from federated events[6]. More details will, again, be given during the The Path to Peer-to-Peer Matrix FOSDEM'20 talk.

4.6.5 In-browser homeserver & Dendrite integration

Dendrite - the next-generation Matrix homeserver - has been successfully compiled to WebAssembly, which would eventually allow to run in-browser Matrix homeservers. Such a setup would eventually allow on-the-fly temporary yet fully-fledged matrix sessions, similar to what is often done with web-based IRC clients. This is, however, blocked by that fact that Matrix federation requires homeservers to be publicly accessible: it is not doable in-browser without an additional layer of network abstraction, which could be provided by Yggdrasil. I already discussed with some Dendrite developers, which are interested in the proxy I wrote for this project.

5 Source code

All the sources and patches written for this project can be found online:

- git.sr.ht/ fnux/yggdrasil-go-coap and git.sr.ht/ fnux/yggdrasil-coap-nodes for the CoAP over Yggdrasil Go library.
- git.sr.ht/ fnux/matrix-yggdrasil-http-proxy for the Matrix-over-Yggdrasil proxy.
- github.com/Fnux/synapse for the synapse fork supporting Yggdrasil address resolution and federation proxying with matrix-yggdrasil-http-proxy. The patches have been applied on the develop branch.
- git.sr.ht/ fnux/matrix-yggdrasil-epfl-report for the LaTeX sources of this report.

[6] MSC1228: Removing MXIDs from events

6 Thanks

I kindly thanks Cristina Basescu from the DEDIS EPFL lab, who supervised and guided me through this project, Neil Alexander and Arceliar from the Yggdrasil project who were always present and quick to help me with technical details, and finally, Matthew Hodgson from the Matrix foundation who helped defining the project and linked me to the wider P2P Matrix initiative.