Cactus Comments

A Decentralized Solution for Web Comments Using the Matrix Protocol

Asbjørn Kjær Olling - S163615

Abstract

The current state of web comment systems is characterized by centralized solutions hosted by third party service providers. The centralized and closed nature of these solutions causes a number of unnecessary problems, which impact both website owners and end-users. Six such problems are identified: identity fragmentation, inconvenient follow-ups, fragility, data disappearance, forced censorship, and limited interaction options. The claim is made that a web comments system could be designed in a decentralized manner, in order to resolve these six problems. Cactus Comments is introduced: a web comments system that leverages the Matrix network to provide web comment services using a federated architecture for decentralization. The system is implemented, documented and tested. Seven formal experiments were designed to verify the distributed properties of the system. Cactus Comments is then evaluated against the six stated problems, and it is concluded that the system solves or lessens the severity of all six problems.

Contents

Abstract **2**

Introduction **6**

Background **7**
- Matrix . . . 7
- Matrix APIs . . . 7
- Homeserver . . . 7
- Directed Acyclic Graph (DAG) . . . 8
- Brewer's Theorem (CAP Theorem) . . . 8
- Federated Model of Decentralization . . . 8
- Federation in Matrix . . . 9
- Room . . . 11
- Events . . . 11
  - Message Events . . . 11
  - State Events . . . 11
  - Power Levels . . . 12
- Identifiers . . . 12
- Client . . . 12
- Application Service (Appservice) . . . 12

Problem Analysis **14**
- Disqus . . . 14
- Facebook Comments . . . 15
- Commento . . . 16
- Summary and Comparison . . . 17

Design **18**
- The Federated Solution . . . 18
- Designing for the Web Platform . . . 18
- Designing for the Matrix Network . . . 19
- System Overview . . . 21
- Web Client . . . 22
  - Determining Alias of Comment Room . . . 22
  - Authenticating as a Guest . . . 22
  - Authenticating as a User . . . 23
  - Accessing the Comments Room . . . 23
- Application Service . . . 24
  - Managing State . . . 24
  - Moderation Rooms . . . 25
  - Comment Section Rooms . . . 25

Implementation **26**
- Technology Choices . . . 26
- Choosing Elm . . . 26

Interaction Model . . . 27
Room State Manipulations . . . 27
Comment Composition . . . 28
Authentication . . . 29
Error Handling . . . 30
Automatic Testing . . . 30
Styling . . . 31
Distribution . . . 32
User Guide . . . 33
Usage Example . . . 33
Handling Formatted Messages . . . 33
Security . . . 34
Implementation of the HTML Sanitizer . . . 34
Fuzz Testing . . . 35
Performance Optimizations . . . 36
Limiting the Problem . . . 37
Selecting an Implementation . . . 37
Accessibility . . . 38

Experiments **39**
Setup . . . 39
Experiment 1: Authenticate Using an Existing Matrix Account . . . 40
Purpose . . . 40
Hypothesis . . . 40
Method . . . 40
Result . . . 40
Experiment 2: Convenient Cross-platform Follow-ups . . . 41
Purpose . . . 41
Hypothesis . . . 41
Method . . . 41
Result . . . 41
Experiment 3: Continued Conversation while the Default Homeserver is Offline . . . 42
Purpose . . . 42
Hypothesis . . . 42
Method . . . 42
Result . . . 42
Experiment 4: First-time Access by Invite while the Default Homeserver is Unavailable . . . 43
Purpose . . . 43
Hypothesis . . . 43
Method . . . 43
Results . . . 43
Experiment 5: First-time Access via Alternate Alias while the Default Homeserver is Unavailable . . . 44
Purpose . . . 44

Hypothesis . . . 44
Method . . . 44
Results . . . 44
Experiment 6: Resilience to Application Service Disruption . . . 45
Purpose . . . 45
Hypothesis . . . 45
Method . . . 45
Result . . . 45
Experiment 7: Application Service Moderation State Persistence . . . 46
Purpose . . . 46
Hypothesis . . . 46
Method . . . 46
Results . . . 46

Discussion **47**
Disruption Scenarios . . . 47
Unavailable Application Service . . . 47
Unavailable Default Homeserver . . . 48
Unavailable Website . . . 48
Federating Comments Through Malicious Homeservers . . . 49
Tackling the Identified Problems . . . 49
Identity Fragmentation . . . 49
Inconvenient Follow-ups . . . 49
Fragility . . . 50
Data Disappearance . . . 50
Censorship . . . 50
Limited Interaction Options . . . 51
Comparison to Existing Solutions . . . 51

Conclusion **52**

Future work **52**
Other Applications of Matrix for Web-Embedded Communications . . . 52
Message-level Cryptographic Secrecy and Integrity . . . 52
MSC1772: Matrix Spaces . . . 53
MSC2753: Peeking Via Sync . . . 53

References **54**

Appendix A: Application Service Implementation **59**
Responsibility and purpose . . . 59
Application service registration . . . 60
User experience: initial setup . . . 60
Technology Choice . . . 61
State . . . 61

Introduction

Most people read comments attached to online content: web sites, personal blogs, news sites and on proprietary networks such as Facebook or YouTube. A few people engage by posting comments themselves, for others to view.

Most independent websites with comment sections will not implement this functionality themselves. This is especially true for fully static websites, which do not have an existing back-end component. For these reasons, there is a sizeable market for third-party comment system providers. A number of providers exist, with different approaches to the problem.

However, most existing solutions have avoidable problems that unnecessarily harm website owners and end-users. This report identifies a number of problems with existing comment systems, and demonstrates that the root causes of these problems are centralization and closed ecosystems.

We introduce Cactus Comments, a comment system that will resolve these issues. Cactus Comments is built on top of Matrix - an open specification for federated communication. With Cactus Comments, comment sections do not rely on any specific host to operate. Because Cactus Comments is built on an open specification, it can be extended, has increased compatibility with existing systems and allows users to adapt the comment system to many circumstances.

First, we will investigate major existing comment section providers, to analyse how existing problems are dealt with. In the background section we briefly explain relevant theory and parts of the Matrix standard. Then, we will describe our overall system architecture as well as the design of our client and backend system. Next, we look at their implementations and how the various components in the Matrix ecosystem fit together. In the chapter on experiments, we will demonstrate the capabilities of Cactus Comments. Then in the discussion chapter, we will evaluate the degree to which the finished system meets the stated goals and its' shortcomings. Finally, we conclude our work and describe potential future work in this space.

This project was designed and implemented partly in collaboration with Carl Bordum Hansen. The system design for Cactus Comments was arrived at in collaboration. The web client was implemented entirely by Asbjørn Olling, and the application service was implemented by Carl Bordum Hansen. Experiments were conducted in collaboration, while the analysis and discussion is by Asbjørn Olling.

Background

This chapter gives a brief introduction to concepts relevant to Cactus Comments. It covers some concepts from distributed systems theory, as well as a number of Matrix-specific concepts.

Matrix

Matrix is an open standard which specifies an protocol for secure real-time communications over a *federation* (defined below) of servers[1].

Most applications built on Matrix focus on group instant messaging, but it is intended for all sorts of real-time communication applications.

Matrix APIs

The Matrix specification defines five APIs. All interaction between components in the Matrix ecosystem is done through these APIs. Each API consists of requests and responses sent and received using JSON over HTTP.

- Client-Server API: Interaction between clients and servers.
- Server-Server API: Federation between servers.
- Application Service API: Privileged server add-ons.
- Identity Service API: Associating Matrix accounts with third party identities.
- Push Gateway API: Integrating push notifications for different platforms.

The Client-Server API is also used by bots and application services for operations that are normally initiated by users.

Cactus Comments uses only three of them: the Client-Server, Server-Server, and Application Service APIs.[1]

Homeserver

A Matrix homeserver is a server program. Users register accounts on specific homeservers. For instance, the Matrix homeserver running at `a.com` might have a user with the user id `@alice:a.com`. This is analogous to how email works. In this example, `a.com` is the *server name* of the homeserver.

Homeservers have a number of responsibilities, such as storing message events, managing user accounts, federating messages to other homeservers, resolving state conflicts between servers, and serving and encoding media. Homeservers implement all five of the Matrix APIs, and are the most central components in the Matrix network.[1]

Directed Acyclic Graph (DAG)

A directed acyclic graph is a graph data structure, in which all edges are directed, and there can not be any cyclic routes between the nodes of the graph. Synchronizing DAGs in a distributed system allows us to always maintain a partial ordering of events, even when merging two inconsistent states of the same DAG.

A well-known example of using DAGs to maintain partial ordering in a distributed system is the Git version control system.[32] Some distributed ledger systems also opt to use DAGs for the same synchronization properties.[33] In Matrix, rooms are a DAG of events. Each new event is added to the DAG with an edge that goes to the preceding event.[1]

Brewer's Theorem (CAP Theorem)

The CAP Theorem is describes necessary compromises for distributed data stores. It states that any distributed data store can fulfill only two of the following three properties:

- Consistency: nodes in the system will always agree on the state.
- Availability: nodes in the system will always respond to requests without error.
- Partition tolerance: the system continues to operate normally, regardless of connection interruptions between nodes.

This theorem applies to Matrix, and by extension to Cactus Comments.

Federated Model of Decentralization

Decentralization broadly describes a system architecture in which nodes don't depend on a single central node for any task. Decentralized architectures include full-mesh peer-to-peer systems, partial-mesh peer-to-peer systems, and federated systems, among others.

A federated architecture is a decentralized architecture in which clients and users' identities are associated with one specific server, and the servers are responsible for exchanging events in a decentralized manner. From the perspective a client in a federated system, the server is not very different from a centralized server.

This brings a number of advantages like simplifying client implementations, and allowing clients to be run in networks that only allow outgoing connections, so only the servers need to be open to incoming connections. It also has the potential to move the data persistence and maintenance responsibilities from the end user to a trusted server administrator.[39]

Federation in Matrix

Homeservers can be run by anyone, and will by default[1] be able to communicate with any other homeserver. This means that a Matrix user on a given homeserver may communicate with Matrix users on any other homeserver.

Federation is the mechanism by which homeservers share chat history. Chat histories (or more broadly: *rooms*, defined below) are replicated in their entirety across all homeservers with participating users. Rooms are synchronized using the Server-Server API[8].

Since message histories are replicated in their entirety across *all* homeservers which have participating users, there is no single point of failure as long as at least two homeservers replicate the room. As long as at least one available homeserver replicates the room, the conversation in that room can continue without interruptions.

This process of synchronization is only eventually consistent. Partitioned homeservers may disagree on which events are present in a room. Thus, the Matrix federation procedure sacrifices the Consistency property in favor of the Availability and Partition tolerance properties of CAP theorem [1].

Homeservers synchronize room histories by sending two types of data units over the Server-Server API. Persistent Data Units (PDUs) contains things such as room events, which are merged into the room state and stored in the homeserver's persistence layer. Ephemeral Data Units (EDUs) represent events that are not persisted, or merged into the room history. Read receipts and typing notifications are examples of EDUs.

PDUs are distributed by the originating homeserver to other homeservers which federate a given room. PDUs are relayed to other homeservers until an receipt acknowledgement is received by the originating homeserver. EDUs are not acknowledged, so they may not be received by all participating homeservers.

Matrix supports relaying PDUs through other homeservers. This means that a homeserver can receive PDUs from a server different from the originating server. This enables new homeservers to join a room, despite the room having participating homeservers which are unavailable.

When relaying PDUs, a question of integrity arises. How can a receiving homeserver guarantee that a given PDU was not altered by the relaying homeserver? Matrix solves this by the originating homeserver cryptographically signing all outgoing PDUs with its' private key. Receiving homeservers can verify the integrity of the event by comparing the signature to the public key of the originating homeserver.

This assumes that the receiving homeserver knows the public key of the originating homeserver. Although homeservers can query each other directly for

[1] Most homeservers support disabling or restricting federation, but the typical configuration enables global federation.

Figure 1: Illustration of Matrix federation. 4 users federated across 3 servers can all communicate.

a list of their public keys via the Server-Server API, it is still possible for a homeserver to receive a PDU from a homeserver it has never communicated with, and is inaccessible at the moment. In this case, the receiving homeserver will query multiple other participating homeservers for the public keys of the PDU-originating homeserver. This approach lets the individual homeserver decide on which homeservers to trust to share the correct keys for a third homeserver. Additionally, this makes it harder for malicious homeservers to provide false keys for a PDU-originating homeserver, since it only requires one faithful homeserver to reveal that malicious activity is happening. This technique of using notary servers to verify public keys is adapted from the Perspectives project[29].

Room

A Matrix room can be thought of as a regular chat-room.

Rooms can be configured in many ways. For instance, they may be open-to-all, invite-only or disallow anonymous access. Rooms do not exist on one homeserver, but are federated between all homeservers, which have users that participate in the room.[8] All messages and room state are represented with events, meaning a room is a directed acyclic graph (DAG) of events. Once a user has joined a room, they can send events in the room. All participating users will receive all events sent in the room.

Events

Under the hood, a room is a DAG of events. Events are described in the Matrix Client-Server API specification[2].

Every event has an event type. This is a string that indicates the kind of data in the event. There are official event types described by the specification, but application developers are free to implement custom types.

There are two categories of room events: message events and state events.

Message Events

Message events are events that generally represent communication activity. These can be many things like VOIP call setups, file transfers or simply text messages. In Cactus Comments, all comments are `m.room.message`-type message events. These can contain either text or multimedia content.

State Events

State events represent state that is persistent over time. State events are modelled as a lookup table. For instance, a client might look up the state key `@alice:example.com` to find Alices display name or avatar image URL. When a new state event is submitted, it replaces the previous value at the same key.

Power Levels

It is not always desirable that everyone in a room have the same permissions. For instance, you may only want designated moderators to be able to ban users.

Power levels are numbers between $-2^{53}+1$ and $2^{53}-1$. They define the minimum power level required for certain actions as well as the power level of each user in the room. The power level configuration for a room is defined by the latest state event in the room DAG with the type `m.room.power_levels`.

Identifiers

Matrix uses identifier strings to represent resources like rooms, users, events, et cetera.

All Matrix identifiers begin with a symbol indicating the kind, followed by a local name for the resource and the server name of the homeserver that allocated the identifier, separated by a colon.

Here are examples of some common identifiers:

- User ID: `@alice:example.com`
- Room Alias: `#myroom:example.com`
- Room ID: `!YeaXUCNFAusqeLQrjB:example.com`

Room IDs uniquely identify a Matrix room. The server part of the room ID is the server which originally created the room. A Room IDs is the primary way of specifying a room used in client-server interactions.

Room Aliases are human-readable names that map to a some Room ID. The server part of the room alias is the server which allocated this name. There may be multiple aliases for a given room, allocated by different servers. Homeservers can resolve room aliases to room IDs.[4][2]

Client

A Matrix client is a program used by end users to interact with the Matrix network. Clients often resemble instant messaging applications, although their features and use cases vary. In the email analogy, Matrix clients are analogous to email clients.[2]

Application Service (Appservice)

An application service is a program that extends the functionality of a homeserver using the Application Service API (defined below)[3].

An application service has higher privileges than a typical client (enforced by a shared secret with the homeserver), which lets it subscribe to certain events. Furthermore, an application service may register namespaces for users and rooms, which it will have exclusive and complete control over. [3]

Cactus Comments uses an application service to manage moderators in the comment room and to create new comment section rooms on-demand.

Problem Analysis

We have identified the following six problems:

- **Identity fragmentation**: To participate in a discussion on a typical article on the web, one often has to create a new identity for a unique comment system.
- **Inconvenient follow-ups**: After commenting, one has to decisively return to the same web page, to follow the conversation.
- **Fragility**: Many of the existing solutions are tightly bound to a central server. It becomes a single point of failure for the whole system.
- **Data will disappear**: The comment section exists only as long as the third party provider. Companies bankrupt, projects get abandoned.
- **Censorship**: A third party service provider often has the power to censor comments. The website's owners and users are not necessarily in agreement with the moderation principles of the third party. This is especially problematic for oppressed minority groups.
- **Limited interaction options**: One is forced to interact with comments through a single client, which can give problems for people with disabilities or unsupported devices.

In this section we will compare three major comments systems, with respect to each of the six identified problems. The aim is to establish that these problems are prevalent, and provide a basis to compare Cactus Comments with.

These comment systems were selected to demonstrate three different common approaches to web comments. The three comments systems we will compare are:

- Disqus[23] (commercial web comments service provider)
- Facebook Comments[22] (embeddable comments plug-in by a social network)
- Commento[24] (open-source web comments)

Disqus

Disqus represents the proprietary third-party service provider. A website owner may use the system by registering an account on Disqus' servers, and then including code for web component in their website along with an API key, which will receive and send comments using Disqus' proprietary API, on their servers only.

Following up on a conversation requires navigating to the site with a Disqus comments section to manually check for replies. This is inconvenient for the user, and can easily leave unfinished conversations.

Disqus is entirely dependant on Disqus' server infrastructure. If Disqus has a site-wide critical outage, the comments will be unavailable to all users.[2]

[2] Disqus have had six outages of varying severity in 2020.[25]

If Disqus were to bankrupt or otherwise discontinue the Disqus comments product, the service and data managed by it would become permanently unavailable. Disqus state that: "without prior notice to you, [...] we may suspend or stop the Service altogether".[26]

The Disqus Terms of Service explicitly restrict the freedom of speech of its users. Disqus reserves the right to remove content and revoke user access of the infringers, based on these restrictions. When using Disqus, the website owner is restricted from a number of conversation topics.[26]

As of December 2020, Disqus only supports interacting via the web component. Although there was at one point an iOS app, this has since been discontinued, and no Android app was ever made. Users have no choice over how to interact with Disqus, which can be a problem on specific hardware, or for disabled persons.[23]

Comment systems with similar properties to Disqus include IntenseDebate, GraphComment, Muut, CommentBox, and others.

Facebook Comments

Facebook Comments is a comment system that integrates with the Facebook social media platform. A website owner can set up Facebook Comments by authenticating as a Facebook user, then providing the Facebook servers with a content identifier of a Facebook post to replicate the comments to. Facebook then generates code for a web component, that the website owner can put in their website.

Facebook Comments have convenient follow-ups for users who already have Facebook accounts. When a comment is replied to, the user receives a notification in the Facebook interface. This means that the user is notified about a continuing conversation, without having to actively navigate to the comment section.

Similarly to Disqus, Facebook can at any time choose to discontinue the product, without any obligations to the user. This makes the website dependant on Facebook's continued interest in maintaining the Facebook Comments infrastructure. If Facebook discontinues the product, the website owner has no recourse.

Facebook also maintains guidelines for limiting free speech, and reserves rights to remove content or revoke the access of users that don't comply. This means that websites using Facebook Comments will be subject to the same restrictions. This can be problematic to oppressed or otherwise persecuted communities.[3]

Facebook does have a number of clients for different devices, many of which are capable of commenting and receiving notifications regarding comments. However, the APIs are not public, and Facebook does not allow third parties to create additional clients. Because of this, there is a hard limit to how many clients that

[3] Recent years have seen politically controversial groups migrating to federated social media due to problematic censorship terms.[27][28]

can exist (decided by Facebook). Users with unsupported devices or acessibility needs are left without recourse.[22]

There are other comment systems with similar properties to Facebook, which come with similar problems. These include YouTube, Reddit, Instagram and Twitter.

Commento

Commento is an open source comment system which anyone may use and/or modify to better suit their needs. It consists of a backend service to manage users and store comments, and a frontend web-component to interact with the comment section. Website owners can either pay for access to Commento's backend services, or download the backend software and host it themselves. In either case, the website owner embeds the Javascript code for the front-end element in their website.

Commento manages identities in the backend. Identities can not be shared between Commento instances, so a user has to create a new identity for each instance of the server-side Commento service.

When someone directly replies to a given comment, that user can be notified over email of the new comment. The email will contain a link to the relevant comment section. The user will have to use an email client to receive the notification, and from the email client switch to using a web-browser to follow up on the conversation. The user has to use two different programs to complete the interaction. [4]

Commento depends on a specific instance of the backend service to be running, for the comment sections to be available. If the backend service fails, or if the service if offline for any reason, the comment section becomes unavailable.

The comments are stored in a PostgreSQL database managed by the backend service. In a self-hosted Commento installation, the website owner has full ownership and control over the comment and user data. The system does not depend on third party services for the continued existence of the data. Additionally, the code itself is released under the MIT license, which grants users the rights to use, modify and distribute the software. This means that an entire Commento system can run without dependence on entities outside the control of the website owner.

This independence from external services also implicates independence from external control. When self-hosting Commento, the website owner defines the terms of usage, without any third-party influence.

[4] Interestingly, the Commento web client and a typical email client are both programs designed for facilitating natural-language conversations between humans. However, email canont be used to send messages in the Commento conversation, and is only used as a notifications service.

Since users are granted the rights to modify and extend the Commento system, there is not an enforced limit to the number of clients that can exist. However, as of December 2020, there is only the one web client. This is very different from the Facebook solution: where Facebook has several clients and forbids the creation of new clients, Commento has only one client and welcomes the creation of new clients.

Comment systems with similar properties to Commento include: StaticMan, Talkyard, Isso, Remark, and Hyvor Talk.

Summary and Comparison

The following table shows an overview of the issues that afflict each of the three solutions. An `X` indicates that the problem burdens the comment system.

.	Disqus	Facebook	Commento
Identity Fragmentation	X	X	X
Inconvenient Follow-ups	X		Email-based
Fragility	X	X	X
Data will disappear	X	X	
3rd party censorship	X	X	
Limited interaction options	X	X	X

Design

This section describes the design of Cactus Comments. It covers the considerations made, and the resulting design. The final design is the product of a tight feedback loop between the implementation process and executing design revisions. However, the vast majority of the design work was done before any code was written.

First, we discuss how the federated model of decentralization can be used to tackle the problems identified in the previous section.

We move on to investigate the possibilities afforded to us by the web platform and the Matrix ecosystem. These are evaluated specifically with regards to the web comments use case. A number of design goals are set for how the eventual implementation should interact with these.

Finally, we detail the components of the Cactus Comments system. Attention is given to the responsibilities of each component, how they interact with each other, and how they interact with the Matrix system at large.

The Federated Solution

We suggest that the Matrix protocol for web comments can tackle the six identified problems. The federated model of decentralization can give the comment data a resiliency and longevity that exceeds centralized solutions.

Fragility, data disappearance, and identity fragmentation can be solved letting users participate through the Matrix server of their choice. Fragility and data disappearance are solved by servers in the federation replicating all of the comment data, making each node independent of all other nodes for the continued existence of the comments. Identity fragmentation is solved by letting users choose which homeserver to trust with their identity.

Matrix being an open standard tackles the problems of censorship, limited interaction options, and inconvenient follow-ups. Since Matrix is open and not controlled by a single entity, a site owner does not have to depend on an external service provider to set the terms of usage for the system. Anyone can run a Matrix server, and choose to run Cactus Comments alongside it, setting the terms themselves. Matrix has a large number of tools for moderation, chat, or general purpose clients, giving users freedom in selecting how to interact with the system. Since the primary use case for Matrix today is instant messaging, being notified and following up on conversations is a built-in capability in most Matrix applications.

Designing for the Web Platform

Since the stated use-case is specifically to provide embeddable comment sections for web comments, Cactus Comments is fundamentally tied to the web platform, the web browser.

Web browsers are restricted to a select number of protocols for building networked applications. The most prominent among these being HTTPS. All five of the Matrix APIs are intended for use with HTTPS, which makes Matrix a remarkably good fit for building applications for the web platform. The embeddable web client should be able to communicate directly with any Matrix homeserver, without depending on any intermediary services.

The embeddable aspect means that Cactus Comments should be able to integrate well with existing web applications. There are two aspects to this:

The first is limiting the logical interface of the comment system, so as to minimize its effects on the application in which it is embedded. Having fewer interactions with the surrounding website we means having fewer requirements for the behavior of the website.

The second aspect is about allowing the embeddable web client to integrate visually with the aesthetic of the embed environment. The style of the client should be highly portable, so it can be embedded in a range of environments without damaging the visual design intents of the implementor. This should be achieved by making as few visual choices on behalf of the implementor as possible, while maintaining a layout which will not be adversely affected by choices made by the implementor.

Designing for the Matrix Network

As was covered in the background chapter, a Matrix room is a graph of events exchanged by any number of Matrix users. This is fit to model a to a web comments section.[5] For instant messaging applications, which is the most common application of Matrix, clients use rooms to represent group conversations.[12][13] Cactus Comments will represent comment sections using rooms.

Although custom event types are supported, the Matrix specification provides a number of standard event types with defined behavior. Specification-compliant clients and homeservers will react to these events in well-defined ways. Using standard event types will allow the comment system to take advantage of existing software built for the Matrix network. It is a design goal for Cactus Comments to exclusively rely on standard event types present in the r0.6.1 Client-Server API specification.

A notable standard event type is `m.room.message`, which is used to represent human-readable chat messages. This event type has eight variants, representing eight different media types. These include `m.text` for text messages, `m.image` for pictures, and `m.audio` for sound, among others. Cactus Comments should leverage these standard types to enable multimedia communications with a number of existing clients.[2]

[5] Existing comment sections often resemble a tree-structure, sometimes a flat sequence. [22][24][23]

Moderation capabilities are a highlighted feature of existing comment systems. [22][24][23] In order to be considered a viable alternative, Cactus Comment should have moderation capabilities comparable to existing solutions. Matrix provides the standard `m.room.power_levels` state event type, which can be used to granularly manage moderation roles of individual users, as well as the permissions afforded to individual roles. Using this power levels mechanism will allow Cactus Comments to make use of the existing tools for moderation in the Matrix network.

Following the observation that power levels are set on a per-room basis, a problem appears: if the comment system administrator would like to add a new moderator or ban a user across all comment sections on a website, they would need to do this for each comment section room individually. Since a website might realistically have hundreds or thousands of comment sections, this is not practical.

The same impracticality appears, when we look at room creation. Matrix rooms need to be explicitly created, in order for users to be able to interact with them. It was considered to simply let the first interacting user create a given room, if it does not yet exist. However, this is not a viable option, since the creator of a room is assigned a high power level. One of the stated goals of the comment system, is to let the website owner have full moderation power over the comment sections on their website, and letting the first user be a moderator is in direct conflict with that goal.

The Application Service API is a can be used to solve this problem. The application service is a networked program which is registered with a specific homeserver. An appservice will be notified of any transactions that involve a room with a room alias that fits a provided regular expression. The server waits for a response from the appservice, before responding to the user-provided request. The appservice has full control over all rooms in its namespace.

If the web client only ever tries to join room aliases that match the regular expression registered to an appservice, that appservice could initialize any not-yet-created rooms automatically, and set the correct moderation settings for it. Similarly, whenever changes are made to an existing room (such as banning a user), the appservice will be notified, and it could replicate the events across all of the other comment section rooms of a website.

Cactus Comments should demand as little as possible of the website owner/administrator to function. If we do not want to require every website owners to host their own instance of such an appservice, the appservice should be able to maintain separate moderator lists and ban lists, for separate websites, and be able to identify which moderation policies to initialize rooms with.

System Overview

The final design of the Cactus Comment involves two components, which should be used in combination with a web server and a Matrix homeserver.

One component is an embeddable web client, which can be used by website developers. The web client is a minimal Matrix client, designed to be embedded in a webpage, to add comment section functionality. It uses the Matrix Client-Server API[2] to read and send comments. The web client does not require users to authenticate using existing Matrix accounts, and accesses the comments using a guest account by default. However, the web client does allow users to authenticate using existing Matrix accounts, at which point the web client will only communicate with the homeserver on which the account is hosted. This gives the web client independence from any specific homeserver.

The other component is an application service. This appservice manages moderation roles and ban lists across all rooms contained in a namespace. Website owners can register such a namespace with the application service by interacting with a setup bot using a Matrix messaging client. The appservice's only persistence layer is Matrix, and it only ever uses standard Matrix events.

Figure 2: Components of the Cactus Comments system architecture

Web Client

The web client is a minimal, single-room Matrix client. An instance of the web client will only connect to one room, as a single instance of the web client only displays one comment section.

Determining Alias of Comment Room

In order to connect to a specific room, the client needs to know a room alias[6] which resolves to that room. It does so by generating the room alias from three fields provided by the site owner. These must provided when initializing the web client.

- `siteName`: Name identifying the site namespace, registered with the application service.
- `commentSectionId`: A unique identifier for this comment room.
- `serverName`: Name identifying the Matrix server.

The `siteName` and `serverName` fields may be set statically on a website-wide basis, but the `commentSectionId` field must be unique for each comment section.

The web client will generate a room alias following this scheme (text enclosed in angle brackets to be replaced with configuration variables):

```
#comments_<siteName>_<commentSectionId>:<serverName>
```

There are several reasons for choosing this format:

- The alias has a constant prefix, simplifying the namespace registration for the appservice.
- The `siteName` can be used by the appservice to infer which moderation settings should apply to the room.
- There is a clear distinction between which values should remain static between comment sections, and the parameter which should change on each comment section.

The client could allow the website owner to construct the room alias string manually. This would allow for attaching the web client to arbitrary rooms, and not necessarily those with a specific naming scheme. However, the decision was made to focus on clarity and ease-of-use for the stated use-case.

Authenticating as a Guest

While initializing, the web client will register a temporary guest user with the default homeserver. The guest account allows the web client to display comments, while the user is not logged in.[7]

[6] The *room alias* is a name by which a Matrix client can look up a specific room id.

[7] The guest account is registered using a single HTTP POST request to the Client-Server API `/register` endpoint, with parameter `?kind=guest`. The JSON response contains an access token which can be used for the following requests.[2]

The default homeserver URL is provided by the website owner when initializing the client. This URL must point towards a homeserver with guest registration enabled. The default homeserver is only used for guest access; Once a user logs in to their Matrix account, the default homeserver is no longer needed.

The guest access token can be used for both sending and receiving comments, provided that the room is public, and guest posting is permitted[8] [2]

Authenticating as a User

If the user wishes to submit comments using an existing Matrix account, they may authenticate with whichever homeserver they choose. As is typical for Matrix clients, this can be done through the web client UI.

The user interactively provides a username, password, and homeserver URL. The client uses this information to craft a login request for the provided URL. [9]

Once the commenting user has authenticated with some homeserver, all further Matrix communications from the web client will happen with chosen homeserver.

If the user logs out (or if an authentication error is returned to any request), the client will fall back to the default homeserver, and issue a guest registration task.

Accessing the Comments Room

First access (once authenticated as a guest or a user) to the comments room is done using a chain of three requests to the Client-Server API.

1. Look up room ID of generated room alias.
2. Get sync tokens and messages using the room ID.
3. Get member users of room, using the room ID.

The information collected can be used to initialize the web client's internal state for modelling the Matrix room.

This room-initialization flow was originally a chain of five requests, but has been optimized through several iterations to reduce the bandwidth and time to complete.

See Figure 3 for an overview of this interaction. [10]

[8] The application service will configure these properties.

[9] This is done by issuing a HTTP POST request to the Client-Server API's `/login` endpoint, with a JSON body consisting of the username and password. The response contains an access token for this user.[2]

[10] Although the sequence diagram seems to indicate no message latency, the latency of requests vary in practice.

This chain of three requests is carried out when the client is in an authenticated state, but does not yet have the room state.

Application Service

The application service (appservice) is a server-side program that interacts with any specification-compliant Matrix homeserver. The appservice is responsible for managing moderation policies across a range of comment sections, and initializing new Matrix rooms for new comment sections. It does this by carefully replicating events between Matrix rooms.

Managing State

Being able to persistent moderation policies for the comment sections of a website, the appservice needs to maintain some state. The original designs for the appservice included mainting lists of sitenames, owners and policies in a traditional relational database. This idea was abandoned in favor of using Matrix events for all persistent state. This gives our moderation metadata all of the resilience properties that Matrix provides, just like the raw comment data.

The appservice will register the prefix `#comments_` with the associated homeserver. This lets the homeserver know, that it should notify the appservice of new events in any room with a room alias that starts with `#comments_`.

Moderation Rooms

In order to register a new website to manage moderation policies for, the website owner should use a Matrix client to send a message to a specific user id, registered by the appservice. Through a chat interface, the user can register a new website namespace, or `siteName`.

This creates a moderation room with the room alias `#comments_<siteName>`, in which the website owner has a high power level. Select events in this room (power level changes, user bans, unbans) will be replicated from this room into all comment section rooms in that `siteName`'s namespace.\footnote{ The appservice finds all the rooms that are match `#comments_<siteName>_.*:<serverName>`. }

Comment Section Rooms

As covered in the web client design section, comment section rooms will have a room alias like `#comments_<siteName>_<commentSectionId>:<serverName>`. Whenever the appservice is notified of an event for a room which does not yet exist, but has a room alias that matches that grammar, it will create that room. The appservice will look up the moderation room of that `siteName`, and replicate the relevant moderation events into new the comment section room.

The comment section rooms should also replicate moderation events back to the moderation room. If a user is banned in a comment section room, that `m.room.member` event should be replicated to the moderation room, from where it will be replicated into all comment section rooms, as per the moderation room's behavior.

Implementation

This chapter will go over the implementation of the Cactus Comments web client. The Cactus Comments web client was implemented by Asbjørn Olling. The code is released as free software under the AGPL license.[11]

The Cactus Comments application service was implemented by Carl Bordum Hansen. A description of the Cactus Comments appservice written by Hansen can be found in Appedix A of this report.

Technology Choices

The embeddable client is implemented in Elm, CSS and JavaScript.

Elm is a programming language that compiles to JavaScript, which can be distributed with the web page. CSS and JavaScript are used for improving integration with the web platform.

As of Cactus Comments client v0.3.0, the linecount distribution is as follows:

Language	Files	Blank lines	Comment lines	Lines of code
Elm	9	403	181	1491
CSS	1	35	36	139
JavaScript	1	4	3	24
Sum	11	442	220	1654

Choosing Elm

Elm is a purely functional programming language; it is strongly typed and only supports immutable values. Elm claims that these robustness measures result in "No Runtime Exceptions"[18]. While there are technically still some scenarios that can crash an Elm program[12], it is true that Elm programs are much more explicit with regards to error handling. This results in more predictable behavior, and a more resilient program.

Elm does not support side-effects to happen inside functions in any way. The Elm runtime passes input events and the previous state object into a call to our update function, which we will guarantee returns a state object, and an optional effect. In Cactus comment, the most commonly used effect is an HTTP request to a Matrix homeserver. Effects are expected to generate new input events. These can contain data such as the success of the effect, and possible resulting data.

[11] The web client project page and git tree is publicly accessible at https://gitlab.com/cactus-comments/cactus-client

[12] For instance: exceeding the memory limits of the host system is still a fatal error.

In the web client, `Msg` is the input data type and `Cmd Msg` is the type of an outgoing effect, which result in data of type `Msg`. `Model` is the program state type. This means the type signature for all logic handling in our program is:

```
update : Msg -> Model -> (Model, Cmd Msg)
```

This architecture allows for precise and predictable composition of effects on the outside world. Additionally, the statically typed nature of the language makes the possible error cases semantically explicit, making the failure scenarios of a program clearer and simpler to manage. These two aspects combined, forces the web client to consist of a concise list of interactions, with well-defined possible results.

The web client is a very effectful application. A significant portion of the program logic is concerned with sending and receiving messages from Matrix homeservers. Getting these interactions right is critical to the correct functioning of the client. Using Elm as the primary language helps us model how these interactions compose in a precise and safe manner.

Interaction Model

The web client defines 11 different possible input messages, that may manipulate the client's internal state. Each input message is generated as the result of an effect on the outside world. Care was taken to reduce the possible interactions with the outside world to the smallest possible set of messages. This subsection will describe each message, how it affects the web client state, and which effects it may generate.

The 11 messages are as listed:

- `GotRoom`
- `ViewMore`
- `GotMessages`
- `EditComment`
- `SendComment`
- `SentComment`
- `ShowLogin`
- `HideLogin`
- `EditLogin`
- `Login`
- `LogOut`

Room State Manipulations

`GotRoom` This input message is generated whenever the web client has connected to a Matrix room for the first time with a given access token - or failed while attempting that action.

The data contained in a `GotRoom` message has the type:

```
Result Session.Error (Session, Room)
```

This type signature indicates that the data is either a Matrix API error (see the *Error Handling* section below), or a tuple consisting of a new `Room` object along with the `Session` object which was used to create it.

The current state object's `room` and `session` properties are replaced with the new data. This input message always results in an effect that stores this latest `Session` object in the web browser's HTML5 localstorage persistence layer. If the room contains fewer comments than the current pagination size, this input message will also result in an effect that fetches more messages using the Client-Server API's `/messages` endpoint[2], which will eventually resolve to a `GotMessages` input message.

`GotMessages` This input message indicates that an attempt to fetch more messages has completed (regardless of success or failure). The data associated with this input message has the type signature:

```
Session Room (Result Session.Error GetMessagesResponse)
```

The message data has a type signature including an existing `Session` and `Room` object, in order to maintain the guarantee that a `GotMessages` event can *only* be created when the web client is in a state where it has a valid access token from the homeserver, and an initialized room which it can merge new messages into. This implementation practice takes advantage of the thorough type system of Elm, to make guarantees about the possible states of the web client. This practice helps us avoid errors related to invalid program states.

The remaining data contained in this input message is either a Matrix API error, or a `GetMessagesResponse` object, which, if successful, contains a list of Matrix message events and pagination tokens to use for subsequent message fetches.

`ViewMore` This input message type indicates that the user has pressed the "View More"-button in the user interface. This will generate an effect to fetch more messages via the Client-Server API's `/messages` endpoint, which eventually resolves to a `GotMessages` input message. Similarly to the `GotMessages` input message, the `ViewMore` input message requires a `Session` and `Room` object in order to be constructed. This lets us leverage the Elm type system to enforce the guarantee that a `ViewMore` message cannot be generated, if the client does not have a valid room and token object.

Comment Composition

`EditComment` This input message indicates that the user has edited the text in the comment editing text field. This input message contains a simple `String`, with the new contents of the entire text field contents. The internal state object is updated with the contents of the text field. No effects are generated by this message.

`SendComment` This input message indicates that the user has pressed the `Send` button. Similar to other input messages, this message's constructor requires a valid `Session` and `Room` object. This guarantees that the web client cannot issue send effects while in an unauthenticated state.

The internal state is updated to let the comment editing text field contain an empty string, and an effect is output to sends the current contents of the text field as an `m.room.message` event with message type `m.text`. This effect eventually resolves to a `SentComment` input message.

`SentComment` This input message is generated upon the completion (successful or not) of an attempt to send a comment. The associated data has the type signature:

```
Result Session.Error ()
```

The type signature tells us that the message will either contain a Matrix API error, or the unit type `()`, which contains no data at all - indicating a success.

Authentication

`ShowLogin` This input message indicates that a user has pressed the "Log in" button. This displays a login form in the browser's Document Object Model (DOM).

`HideLogin` This input message indicates that a user has pressed the "Back" button of the login form. This removes the form from the DOM.

`EditLogin` This input message indicates that the user has changed the content in one of the three text fields in the login form. The client's internal state is updated to reflect the new content of the form.

`Login` This input message indicates that the user has pressed the "Log in"-button in the login form. The data associated with this input message is the three strings present in the login form: a homeserver URL, a username and a password. The client will generate an effect which first issues an authentication request to the given homeserver URL, then carries out the initial room fetch using the new access token, as described in the Design section (see Figure 3. for an overview). The effect will eventually result in a `GotRoom` message.

`LogOut` This input message indicates that the user has pressed the "Log out"-button. This initiates a guest registration request to the default homeserver, falling back to an anonymous access of the room. The effect continues by using the resulting access token to issue a chain of requests to initialize a `Room` object with the guest access token, as displayed in Figure 3. This eventually generates a `GotRoom` input message.

Error Handling

Some of the effects issued by the client may fail to resolve correctly. Specifically, every HTTP request issued has a number of failure modes, including:

- Any number of network failures
- Request timeouts
- Failure to parse the HTTP response
- Any number of Matrix-specific Error responses

In case of a network failure or a request timeout, the web client will wait a number of seconds before re-issuing the same request. The number of seconds is incremented by an exponentially increasing amount on each retry, as per the exponential-back-off principle. This reduces load on both the client and the homeserver in case of failure.

Any non-200 response[13] from a Matrix homeserver will include an error field, and an optional error description.[2] The client parses the Matrix-specific error codes, rather than acting on HTTP status codes. Some of these error codes are handled gracefully by the client:

`M_LIMIT_EXCEEDED` An `M_LIMIT_EXCEEDED` error indicates that a user has exceeded the rate limits enforced by this homeserver. These error responses will have a field named `retry_after_ms`, indicating the time to wait before being allowed to issue another request. The web client will parse this number and wait the given number of milliseconds before re-issuing the same request.

`M_UNKNOWN_TOKEN` This indicates that the provided access token is considered invalid. The web client will invalidate the current `Session` object and issue a guest account registration to the default homeserver.

Any other error Any other error is considered an unrecoverable error, and is displayed to the user. If an error description is present, it is displayed alongside the error code.

Automatic Testing

Along with the implementation of the client, a suite of unit tests have been developed, to aid with verifying the core functionality of the web client as a closed system. Focus was deliberately given to the most critical and the most complex parts of the system, as these are respectively the areas that are most important to get right, and the most difficult to get right.

A total of 25 unit tests were developed. This results in a coverage spanning of 31% of all code branches, or 23% of all function declarations.

[13] "200 OK" is the standard HTTP status code indicating success.

The coverage distribution over the modules of the web client are summarized below. It is apparent that the majority of testing effort was spent on the `ApiUtils` and `FormattedText` modules.

Module	decls	let decls	lambdas	branches
`ApiUtils`	10/10 (100%)	4/4 (100%)	2/2 (100%)	2/2 (100%)
`Editor`	0/7 (0%)	0/12 (0%)	0/3 (0%)	0/4 (0%)
`FormattedText`	8/11 (73%)	2/2 (100%)	7/9 (78%)	31/39 (79%)
`LoginForm`	0/4 (0%)	0/7 (0%)	0/3 (0%)	0/2 (0%)
`Main`	0/6 (0%)	0/10 (0%)	0/7 (0%)	0/28 (0%)
`Member`	0/4 (0%)	n/a	0/5 (0%)	n/a
`Message`	1/9 (11%)	4/11 (36%)	0/7 (0%)	3/23 (13%)
`Room`	0/11 (0%)	0/4 (0%)	0/7 (0%)	0/2 (0%)
`Session`	0/22 (0%)	0/1 (0%)	0/4 (0%)	0/16 (0%)
total	19/84 (23%)	10/51 (20%)	9/47 (19%)	36/116 (31%)

Styling

A default CSS stylesheet is provided with the Cactus Comments web client. This stylsheet has been designed to optimize for portability across web environments with different applied stylesheets, as well as ease of modification for website developers.

Although Elm does provide ergonomic abstractions for web applications stylesheets, the choice was made to implement the visual styling for the web client using plain CSS exclusively, with no pre-processors or other abstractions. CSS is a W3C native standard, and is supported by every web browser. It is expected that any website implementor will have some familiarity with CSS. Additionally, no tools (aside from a text editor) are required in order to modify the provided stylesheet. Using plain CSS lessens the friction associated with adapting the stylesheet for a specific visual appeal.

In order to test the portability of the web client's default stylesheet, we embedded the web client in three different web pages, each of which use a different stylesheet. Specifically, we tested it in one website using Sakuro CSS[34], one website using mkDocs' default theme[35], and one with no stylesheet at all. This testing led to discovering a series of incompatibilities, which have subsequently been resolved.

Below is a screenshot of the comment system in three different styling contexts:

Figure 3: From left to right: mkDocs' CSS, Sakuro CSS, no CSS.

Distribution

Since the Cactus Comments web client is released under an AGPL license, the website owner is free to make a copy of the compiled Javascript files, and include these in their application.

However, for ease of use with minimal websites, a distribution mechanism was chosen for Cactus Comments. This distribution should allow website implementors to link to the appropriate Javascript and CSS assets, without hosting these themselves.

Cactus Comments' compiled assets are distributed using IPFS. IPFS is a global content-addressable peer-to-peer network for file sharing. We won't go into much depth regarding the distributed properties of IPFS, but being a distributed system, it does share many of the Matrix's fault tolerance properties.[36]

The IPFS support is seeing growing support in browser, with three major web browsers already supporting IPFS natively.[**?**] The browsers which *do* support IPFS natively may choose to fetch the asset from a number of peers on the IPFS network. For the browser that don't an IPFS gateway link can be provided. IPFS gateways are services which proxy HTTP requests onto the IPFS network, effectively acting as a cached proxy for IPFS content.[36][38] Once on the IPFS network, users may access the file via any IPFS gatway. This lets us cache the content on a number of Content Delivery Network (CDN) servers, by leveraging the distributed nature of IPFS.

On every release of the Cactus Comments web client, an automatic continuous deployment process compiles the Elm application into single minified Javascript file, and uploads the resulting Javascript - and the default CSS stylesheet - to pinata.cloud, which is an IPFS pinning service and IPFS gateway.[38] From there, the files can be distributed amongst IPFS peers who use the file.

User Guide

This subsection describes how to embed the Cactus Client web client in any web page. For a usage guide on how to install the appservice, refer to Appendix A.

In order for a website implementor to use the Cactus Comments web client, they need to do the following two steps:

- Include the client's Javascript and CSS files from the webpage, by using the IPFS gateway link.
- Call the `initComments` function with a Javascript object that has the following members:
  - `node`: a DOM node to render the comments inside.
  - `defaultHomeserverUrl`: the URL of a homeserver with guest registration enabled.
  - `serverName`: the server name of the homeserver with a Cactus Comments appservice registered.
  - `siteName`: the moderation namespace registered with the Cactus Comments appservice.
  - `commentSectionId`: a unique identifier for the comment section to load on this page

Usage Example

The following HTML code is an example of a minimal but completely functional web page, which embeds the Cactus Comments web client v0.3.0, instructing it to use the homeserver hosted at `https://cactus.chat:8448`:

```
<script type="text/javascript" src="https://gateway.pinata.cloud/ipfs/QmSJZhJ8zNBQbE2vngLPG6
<link rel="stylesheet" href="https://gateway.pinata.cloud/ipfs/QmSJZhJ8zNBQbE2vngLPG6EAnFpiN
<div id="comment-section">Loading Comments...</div>
<script>
  initComments({
    node: document.getElementById("comment-section"),
    defaultHomeserverUrl: "https://cactus.chat:8448",
    serverName: "cactus.chat",
    siteName: "ExamplePage",
    commentSectionId: "ExampleSection"
  })
</script>
```

Handling Formatted Messages

Several of the instant message `msgtypes`[14] may contain optionally formatted text content[2]. This is how Matrix supports rich text features like bold, italics, quotes, nested lists and code blocks.

[14] 'm.text', 'm.emote' and 'm.notice'

The formatted text content is presented in an optional field in the message event called `formatted_body`. The markup language used is indicated by the associated `format` property.

Currently `org.matrix.custom.html`, a subset of HTML, is the only markup syntax described by the Matrix specification[15], but clients are free to implement whichever new format they wish, or extend the existing format for client-specific features. Notably, there is an active proposal to extend the current format syntax to support rendering LaTeX content, which is already supported by at least one client[9].

Security

Rendering user-provided HTML in the browser is potentially a security vulnerability. If not handled properly, a malicious user could inject javascript code into a victim user's browser, potentially exposing cookies, session tokens or other sensitive information. Cross-Site Scripting attacks are number seven on the OWASP Top Ten list of critical web application security risks[19]. When considering the security of the comments web client, protections against Cross-Site Scripting attacks is a critical focus.

Our web client manages the user's Matrix access token, which is sensitive information. The access token could be used by an attacker to gain access to confidential information, or to forge messages on an attacked user's behalf. The web client also is meant to be embedded into other webpages, in which case the embedding website might expose any number of related security issues, or other sensitive informations. Security is of high importance to the comments web client.

Implementation of the HTML Sanitizer

To sanitize, the client first parses the formatted HTML string into a structured tree, using the Elm package `hecrj/html-parser`[17]. This library parses strings according to the WHATWG's guidelines for HTML parsing[21]. When parsed, we are left with a structured HTML tree of the following type (essentially a *rose tree* data structure):

```
type Tag = String

type Attribute = (String, String)

type Html
    = Element Tag (List Attribute) (List Html)
    | Comment String
    | Text String
```

[15] As of Client-Server API r0.6.1.

The sanitization algorithm traverses this tree recursively, comparing each element tag to the set of tags allowed according to the Matrix Client-Server specification[16].

If the tag *is not* on the whitelist in the set of allowed tags, replace the tag with `div`, and remove all of its attributes. If the tag *is* on the whitelist, keep the tag, and apply a transformation function to its attributes. In both cases, the element's children are replaced by recursive applications of the sanitization function.

The attribute transformation function are selected based on the tag. Only `font`, `span`, `a`, `img`, `ol`, and `code` tags permit having any attributes. Any other tag will have its attributes replaced with the empty list. An attribute is only kept if it is explicitly allowed according to the sanitization guidelines{matrixspec:cs}, and its value parses in the expected manner.

Some attributes have restrictions on which values are permitted. For instance, `href` URLs must not be relative, and must be one of five specific schema. Color attributes must be 6-character hexadecimal color codes.

Apart from filtering attributes, the attribute transformation functions also transform select attributes into entirely separate attributes. Specifically, the two color attributes are transformed into CSS `style` attributes, and Matrix content URIs (MXC URIs) are replaced with the appropriate HTTPS call to the Client-Server API. For this reason, the HTML sanitization function is not idempotent.

During development, we discovered a CSS injection vulnerability in this last step. It allowed any attacker with access to the Matrix room to craft a malicious message, which injects CSS into some elements of the comments section. We were not able to exploit this vulnerability to any meaningful gain. However, it still might have severe consequences depending on the embed environment. OWASP writes that CSS injections "may lead to cross site scripting or data exfiltration"[20]. The CSS injection vulnerability has been eliminated by strictly parsing the color attribute contents to match only 6-character hex color codes.

Fuzz Testing

In order to solidify confidence in the correctness of the client's HTML sanitization, we wrote fuzzing tests (also known as parameterized tests) for the sanitization functions.

These tests generate structured HTML trees with a mix of valid and invalid attributes and tag values. The fuzzers favor irregular input, such as uncommon characters and extreme list / string lengths. Our test expectations assert that the sanitized output contains all of the valid tags and attributes present in the input, and that the output contains no invalid tags or attributes.

[16] See section 13.2.1.7: m.room.message msgtypes[2]

We immediately discovered an unexpected problem: the sanitization perfoms poorly on large input sizes - especially on wide HTML trees. We observed the javascript repeatedly process exceeding 2 gigabytes of memory usage, inducing a fatal crash. This was not the type of security issue we were expecting to discover, but it is a security issue nonetheless. Since it threatens the availability of the system, it can be classified as a denial-of-service attack. The threat scenario has an attacker submit an sufficiently wide HTML tree in a comment, wasting memory resources of client machines and denying all instances of the web client access to the malicious comment's neighboring content.

Performance Optimizations

Recognizing that Elm as a purely functional language exclusively uses value semantics, it becomes apparent that the memory issue comes from persisting a large number of stack frames - each containing copies of HTML tree values - when traversing a large HTML tree. Seeing as Elm does not allow mutation, we took on the task of re-implementing the function in a tail-recursive manner. This would allow us to make use of tail-call elmination in the Elm compiler, to free unnecessary parent stack frames.

It turns out that traversing a rose-tree data structure in a tail-recursive manner is not a trivial problem. It took a considerable amount of research and development time to arrive at two new implementations, each exhibiting an advanced recursion style:

One implementation is in a continuation-passing style, where the recusive call is given a partially applied function (the *continuation*). The continuation represents the composed transformations to the HTML tree, and is finally executed at the bottom of the tree.[17]

The other implementation simulates a "frame" of already mapped and not yet mapped children of each element, as well as a current value. It employs tail call recusion on iterating over children at each element, as well as a tail recursive descent operation.[18]

With three implementations of the same functionality, and a parameterized testing suite, we can begin to compare their performance. Both of the optimized implementations did show shorter run-times on large inputs than the naive implementation. However, neither of the new implementations could handle the largest tree-size that the fuzz testing suite would generate. When using randomized seeds for the input, the fuzz tests suite would still regularly fail all three implementations on the same memory error.

[17]The source code of this function can be found at commit 'dc343e5' of the git repository in the appendices.

[18]The source code of this function can be found at commit 'e9d7719' of the git repository in the appendices.

Limiting the Problem

Examining the fuzzing test input generator reveals a combinatorial explosion. The entire generator is composed from a number of nested primitive generators. Each individual list generator can generate lists with up to three hundred elements, and each string generator can generate strings with several hundred elements as well. The Matrix specification states that `org.matrix.custom.html` formatted messages can be at nested at most 100 times. In the largest possible case, the fuzzer outputs an html tree with depth 100, at each level an element with 300 children and 300 attributes of a combined 600 characters each.

Assuming only non-unicode characters, each element has a maximum size of ~180 kilobytes, not counting it's children. The entire tree has a branching factor of 300. At depth 100, the total number of elements will 300^{100}, and the total size of the tree is $180kb \cdot (300^{100})$.

This is an extremely large input size[19], and entirely unrealistic to be received in a single HTTPS packet (yet alone contained using computers on earth). This shows that the fatal performance issues was in the fuzzer attempting to generate impossible amounts of data.

When asking "What is the largest possible message event size?", the Matrix spec answers 65536 bytes[2]. Thus, the fuzzing tests were rewritten to limit the input sizes to realistic values.

Selecting an Implementation

In selecting a final formatted message sanitization implementation, the following criteria are taken into account, in prioritzed order:

1. Security correctness / robustness
2. Code maintainability
3. Resource efficiency

In order to select a compare the implementations, all three function were tested again using the new, smaller fuzz testing suite. A total of 10,000 test cases was generated and run against each implementation. All three runs were done the same computer, with no user interaction at run-time. Below are the summarized results:

Implementation	Git commit	Completion time	Average time per test
Naive	`2063dae`	120474ms	12.0474ms
Continuation-passing	`dc343e5`	121781ms	12.1781ms
Frame-based	`f97a1ab`	Failed	Failed

The frame-based implementation was not able to pass all ten thousand test

[19] Totalling some $9.3 \times 10^{241} yottabytes$.

cases, due to select edge cases discovered by the fuzzer. This bug was not readily apparent, and has not been resolved. This implementation was discarded on grouds of not being correct, and of being difficult to maintain (and thus difficult to coerce into a correct state).

Comparing the two remaining implementations shows only a negligible difference in runtime. For the purposes of this comparison, the two implementations are considered equal in performance. Seeing as they both pass all tests, they can be assumed to also be equal in correctness. Finally, we consider the maintainability differences. The naive recursion is far shorter and more easily readable than the continuation-passing style. Having shown a small number of Elm programmers both implementations, all agreed that the continuation-passing style was difficult to understand.

Being equal on all counts except one (on small input sizes), maintainability becomes the deciding factor. Having a more direct and understandable function will make it simpler to extend the functionality in the future - for instance to support upcoming features like LaTeX rendering[9].

For these reasons, the two optimized implementations have been discarded in favor of the original, naive implementation.

Accessibility

The web client uses the libraries `tesk9/accessible-html`[15] and `tesk9/elm-html-a11y`[16] to ensure that the comments section is accessible to persons who are restricted to interacting with it by unconventional means (e.g. using a screenreader).

These libraries contain wrappers around the functions usually used to construct HTML elements. It uses the type functionality in Elm to provide compile-time guarantees about the accessibility of the rendered HTML. For instance, one cannot construct an `img` tag without a description of the image content.

Experiments

A series of experiments have been conducted, in order to confirm the properties of the comment system. The experiments have been designed to illustrate Cactus Comments' properties regarding the six problems identified in the Problem Analysis section of this report.

All seven experiments are qualitative experiments, as we are interested in confirming our assumptions on how the system performs in conjunction with the Matrix network. We are not interested in measuring the performance or practical scalability of the system. The qualitative experiments highlight capabilities afforded to the system by using the Matrix network.

The experiments were designed and conducted in collaboration with Carl Bordum Hansen.

Setup

The following experiments all use the same setup, in which we have three homeservers, with one or two user accounts each. We also have a user with no Matrix account.

Homeserver A A homeserver managed by the website owner. This homeserver's URL is passed to the web client's `defaultHomeserverUrl` parameter. This is also the homeserver which has an instance of the appservice attached. Homeserver A's hostname is `A.com`.

Homeserver B A completely independent homeserver, which initially has never communicated with any users on homeserver A or C. Homeserver B's hostname is `B.com`.

Homeserver C A completely independent homeserver, which initially has never communicated with any users on homeserver A or B. Homeserver C's hostname is `C.com`.

User "Alice" Alice is a user with a Matrix account on homeserver A. Alice's user id is `@alice:A.com`.

User "Bob" Bob is a user with a Matrix account on homeserver B. Bob's user id is `@bob:B.com`.

User "Charlie" Charlie is a user with a Matrix account on homeserver C. Charlie's user id is `@charlie:C.com`

User "Carol" Carol is a user with a Matrix account on homeserver C: Carol's user id is `@carol:C.com`

User "Grethe" Grethe is a user who does not have a Matrix account, and will use the comment system via a guest account.

Experiment 1: Authenticate Using an Existing Matrix Account

Purpose

To show that Cactus Comments does not require users to have identities on any specific server. Users may authenticate using Matrix accounts on any homeserver.

Hypothesis

If a user creates a Matrix account using a client and homeserver unknown to the comment system, that user can log in to the Cactus Comments web client, using their existing Matrix credentials.

Method

1. Charlie accesses a web page with the web client embedded.
2. Charlie clicks the "Log In" button and provides the credentials and homeserver URL for their existing Matrix account.
3. Charlie posts a comment using the web client.
4. Grethe accesses the same web page as Charlie.
5. We observe whether Charlie's comment is visible to Grethe.

Result

A single run of the procedure was executed, yielding results that match the expectation stated by the hypothesis. Charlie's comment is displayed in Grethe's web browser, along with their display picture and display names. The experiment supports the stated hypothesis.

Experiment 2: Convenient Cross-platform Follow-ups

Purpose

To show that the comment system allows for convenient follow-ups, by being available across more than one client on more than one platform.

Hypothesis

When a user has participated in a comment section, they can be notified by further activities in the same comment section, by using other Matrix clients.

Method

1. Alice logs in using the Matrix client *Element*[12] on her phone.
2. Alice logs in using the Cactus Comments web client.
3. Alice posts a comment.
4. Bob posts a comment mentioning Alice.
5. We observe whether Alice is notified of the comment via the *Element* application.

Result

The experiment was conducted once. We observe that Alice is notified of the replying comment. *Element* shows a push notification on Alice's phone, with the contents of Charlie's comment. This supports the stated hypothesis.

Experiment 3: Continued Conversation while the Default Homeserver is Offline

Purpose

To show that the comment system remains available to users who have previously visited a particular comments section - despite service disruption of the default homeserver. The experiment will also show that federation is independent of the default homeserver.

Hypothesis

If two homeservers which have previously joined a comment section exchange messages while the default homeserver is unavailable, and the default homeserver becomes available after the fact, then the Cactus Comments web interface will show new guests the messages which have been exchanged while the default homeserver was offline.

Method

We perform the following steps:

1. Bob logs in using the web client, and posts a comment.
2. Charlie logs in using the web client, and posts a comment.
3. Homeserver A is turned off.
4. Bob responds to Charlie's comment using *Element*[12].
5. Charlie responds to Bob's response using *Fluffychat*[13].
6. Homeserver A is turned back on.
7. Grethe visits the webpage. We record which comments are displayed to Grethe.

Result

The experiment was carried out once. We observe that Grethe's web browser displays the comments posted by Charlie and Bob, while Homeserver A was turned offline. These results match our expectations, and support the stated hypothesis.

Experiment 4: First-time Access by Invite while the Default Homeserver is Unavailable

Purpose

To demonstrate that Cactus Comments is resilient to service disruption of the default homeserver.

Specifically, we demonstrate that a new user can join the comments section for the first time, while the default homeserver is unavailable, if an existing participant explicitly invites the new user.

Particularly interesting is the fact that the new user does not need to know the existing participant. Only the participant needs to know the user id of the new user. The relationship is one-way.

Hypothesis

If a new user is directly invited to participate in a comments section by an existing participant while the default homeserver is down, that user can gain full access to the comments section.

Method

1. Bob logs in using the web client and posts a comment.
2. Homeserver A is turned off.
3. Bob uses the *Fluffychat*[12] client to invite Charlie to the Matrix room.
4. Charlie uses the *Element*[13] client to join the room and post a comment.
5. Homeserver A is turned back on.
6. Grethe accesses the web content with the comment section attached. We record the displayed comments.

Results

When Grethe accesses the webpage with the comments section, her browser displays both Bob and Charlie's comment. This shows that Charlie was able to participate in the comments for the first time, while the default homeserver was unavailable. This outcome supports the stated hypothesis.

Experiment 5: First-time Access via Alternate Alias while the Default Homeserver is Unavailable

Purpose

To demonstrate that Cactus Comments is resilient to service disruption of the default homeserver.

Specifically, to show that the comment section can be accessed by a new user, if the new user can discover an alternate room alias, which points to the comment section room. The key point here, is that no existing participants need to know the new user directly, in order for the new user to join. The relationship is one-way.

Hypothesis

If a new user joins the comment section room using an alternate room alias, while the default homeserver is unavailable, then the new user can access to the comment section.

Method

1. Bob logs in using the web client and posts a comment.
2. Homeserver A is turned off.
3. Bob uses the *Element*[12] client to assign another room alias to the comment section room.
4. Bob broadcasts the new room alias to the public (e.g. via Twitter).
5. Charlie sees Bob's broadcast, and joins the comment section room via the *Element*[12] client.
6. Charlie posts a comment.
7. Homeserver A is turned on.
8. Grethe visits the website with the comment section in use. We record the displayed comments.

Results

The procedure is performed once, yielding the expected results. We observe that both Bob and Charlie's comments are shown in Grethe's browser. This means that Charlie was able to join the room, by using the room alias created by Bob, with only a one-way affiliation relationship to Bob. This outcome support the stated hypothesis.

Experiment 6: Resilience to Application Service Disruption

Purpose

To demonstrate that an existing comment section continues to function while the application service is inaccessible.

Hypothesis

If the application service is inaccessible, existing comment sections remain available and interactive.

Method

1. The Cactus Comments Application Service is turned off.
2. Alice logs in using the web client, and posts a comment.
3. Bob logs in using the web client, and posts a comment.
4. Grethe visits the website, and records which comments are displayed.

Result

The procedure is carried out once, yielding results that match the expectations. In step 4, Grethe's web browser displays both of Alice and Bob's comments. This shows that an existing comment section remains available for reading and posting comments, despite the appservice being unavailable. This confirms the stated hypothesis.

Experiment 7: Application Service Moderation State Persistence

Purpose

To demonstrate that the comment system continues to function, despite service disruption to the application service. Specifically, that the comment system continues to enforce existing moderation policies.

Hypothesis

If a user has been banned from from all comment sections in a registered namespace, and the appservice subsequently becomes unavailable, then the user remains banned from all comment sections in the registered namespace.

Method

1. Bob logs in using the web client and posts a comment.
2. Alice bans Bob from the comment section room, using the *Element* client[12].
3. The application service is turned off.
4. Bob tries to comment in a different comment section, than used in step 1.

Results

When Bob tries to comment in step 4, he is met with an error message:

> M_BAD_STATE: Cannot join user who was banned.

This shows that the homeservers enforce the moderation policies set by the application service, even when the application service itself is inaccessible.

Discussion

This chapter goes over the observed qualities of the Cactus Comments system as a whole, in order to arrive at an evaluation of how well the system tackles the stated problems, and how the system compares to existing solutions.

In the first section, we use the existing experiments as a basis for assessing how failure of various components affects the system as a whole, and what requirements must be met to allow the system to recover.

The second section goes through each of the six identified problems, and discusses whether and how Cactus Comments performs in regards to each.

Finally, we compare Cactus Comments to the three analysed existing Comment Systems, for an overall assessment of its fitness.

Disruption Scenarios

As a distributed system, Cactus Comments has some interesting robustness and redundancy properties. To clarify exactly what these are, we go through three different disruption scenarios. For each scenario, we will describe the reaction of the system as a whole.

Unavailable Application Service

We consider the scenario where the Cactus Comments appservice is offline. This could happen either through hardware failure, network errors, failure of the associated homeserver, software bugs in the appservice, or a number of other reasons.

Experiments 6 and 7, detailed in the *Experiments* chapter, demonstrate that essential features of the comment system continue to function, despite disruption of the appservice. Experiment 6 shows that users can still interact with existing comment sections while the Cactus Appservice is inaccessible. Experiment 7 shows that moderation policies are still enforced while the Cactus Appservice is inaccessible.

We claim that users with sufficient power levels may still manipulate moderation roles, as well as ban, unban or kick users, while the appservice is inaccessible. However, as long as the appservice is inaccesible, these changes will be per-room.

On-demand room creation is not available, while the appservice is offline. If the website owner were to add the web client to a new page (with a new `commentSectionId`), the client would fail to create or access a room, and instead display a human-readable `M_NOT_FOUND` error message, which includes the room alias it tried to access. The site owner can create this room alias manually, if the appservice is entirely dissociated from the homeserver. Otherwise, the alias is in a namespace reserved for use by the appservice.

If moderation changes are made to a room, while the appservice is offline, the homeserver will retry the request (with an exponential back-off timer), until it succeeds. In this way, changes made while offline will still be replicated across other rooms in the namespace, when the appservice comes online again.

The Cactus Comments system only takes a convenience loss, if the appservice is not used. Without the appservice, site owners must create each comment section room manually, and moderation policies can only be set on a per-comment-section basis. The appservice can be considered a non-essential component of the system as a whole.

Unavailable Default Homeserver

Experiments 3, 4, and 5 demonstrate that Cactus Comments is resilient to service disruption of the default homeserver[20].

Experiment 3 shows that users who have already joined the comment section room can continue to participate in the conversation while the default homeserver is down, provided that those users have accounts on other homeservers. Experiments 4 and 5 show that users who have not already joined the comment section (and also use different homeservers from the default homeserver), may join the comment section room, provided that a one-way information flow exists between the new user and an existing participant: either the existing participant knowing the user id of the new user, or the new user knowing a current room alias of the comment section room.

The demonstrated resilience scenarios all require the room to be federated to at least one other homeserver. This happens when at least one user from another homeserver logs in to the web client. Once this condition is met, the entire room history is stored on its entirety on two different servers. If this condition is not met, the data is only stored on the disks of the default homeserver and not replicated to any other homeserver. Thus, the resilience to loss of the default homeserver depends on users using different homeservers.

Unavailable Website

A scenario which was not covered by any of the experiments, is the failure of the embedding website. For instance, if the webserver should fail to serve the web content, the DNS name fail to resolve to the correct address or simply if the website owner decides to permanently shut down the website.

This scenario makes the web client unavailable. Thus, users can not discover the comment section in the same place as the web content - but in this scenario the web content is also not available.

[20] The default homeserver is the homeserver found at the URL provided to the web client as the `defaultHomeserverUrl` configuration parameter. It is used for anonymous access to the room.

Assuming the homeserver is hosted separately from the rest of the website infrastructure, the comment section the continue to be available to existing users and new users who can discover the room. If the homeserver has also become unavailable, the behavior described in the previous section applies.

Federating Comments Through Malicious Homeservers

It is feasible for a Matrix homeserver administrator to intentionally disturb the flow of information to their clients. For instance, a malicious homeserver admin might cease to send events from a specific comment section, if that comment section is found to have disagreeable content. Since Cactus Comments is not (as of v0.3.0) capable of interacting with end-to-end-encrypted rooms, there is no cryptographic integrity measure for individual messages. Thus, a malicious homeserver might also intercept messages and alter their contents.

This scenario is not mitigated by Cactus Comments in any way. However, it will only affect the direct users of the malicious homeserver. Federation events in the Server-Server API *are* cryptographically signed, so the malicious homeserver cannot man-in-the-middle (MITM) messages that are being relayed through it from one homeserver to another.

Tackling the Identified Problems

This section goes through the six problems identified in the *Problem Analysis* chapter, and discusses how Cactus Comments tackles this problem, and to what degree the problem is solved.

Identity Fragmentation

Cactus Comments relies on Matrix accounts for identities. Due to Matrix's federated architecture, a Matrix account can be managed by any homeserver. This means that users have the freedom to decide which homeserver to trust with their credentials.

Accounts are not tied specifically to Cactus Comments, but rather can be used with any application compatible with Matrix. Because Matrix has been designed with interoperability in mind, this includes a wide range of other social networks.[30]

Using Cactus Comments reduces identity fragmentation, simplifies identity reuse, and gives the user a greater degree of control over where their account data and credentials are stored and verified.

Inconvenient Follow-ups

Cactus Comments is able to leverage a wealth of existing Matrix clients, allowing the user to interact with the comment section from almost any other native Matrix client. Many of these support notifying users of new events. [**?**]

Using Cactus Comments makes following up on a comment section conversation more convenient. It does this by giving the user several options on how to interact with the comment section, and by using a platform that supports several mechanisms for push notifications.

Fragility

In the event of a service disruption to the Cactus Comments application service, all existing comment sections remain available to existing and new users. If the appservice is not accessible, the system as a whole only loses the ability to create new comment sections on-demand, and the convenience feature of automatically replicating moderation policies across all the comment sections of a site.

In case the default homeserver should become inaccessible, the comment section remains available to all existing users whose homeservers are still in a functioning state. New users may also join the conversation during a homeserver outage, provided that some prerequisite conditions are met.

Cactus Comments is highly resilient to service disruption.

Data Disappearance

Comment sections in the Cactus Comments system are also resilient to the permanent disappearance of the entity proving the web content. As long as there are homeservers federating the room, the data persists. The continued existence of comments exchanged using the Cactus Comments system is not dependant on the continued functioning of any component of Cactus Comments.

Censorship

In Cactus Comments, moderation control is given to the registrant of the moderation namespace. Website owners are not forced to give any single party complete control over the allowed content in their comment sections.

However, we cannot rule out the possibility of federating with a homeserver which could censor or alter the comments being displayed to their direct users. So we cannot guarantee that no external filtering or manipulation of the content is taking place. However, this only manipulation would only affect a subset of the users and not all users. In the end, the users may chooses which homeservers to trust with their data.

Cactus Comments only partially solves the problem of external censorship. Although no external party can force censorship of comments for all potential users, an external party might be able to perform a man-in-the-middle attack to censor content for a specific subset users, who have explicitly placed trust in said censoring party.

Limited Interaction Options

Because Cactus Comments builds on an open standard, there is no hard limit to the number of compatible clients that may exist. There is already a number of existing compatible clients, with different accessibility advantages. Users with special needs have a high degree of freedom in choosing how they want to interact with the comment sections.

Because Cactus Comments is built on Matrix, it already has a high number of existing interaction options, and there is no limit to the number of interaction options that may exist.

Comparison to Existing Solutions

.	Disqus	Facebook	Commento	Cactus Comments
Identity Fragmentation	X	X	X	
Inconvenient Follow-ups	X		Email-based	
Fragility	X	X	X	
Data will disappear	X	X		
3rd party censorship	X	X		Homeserver MITM
Limited interaction options	X	X	X	

Conclusion

We investigated the current state of embeddable web comments systems, and analyzed three specific comment systems to identify six widespread problems. We set out to design a solution that solves or reduces the severity of these problems. A design was made for a decentralized web comments system, which can replicate comment data to users' servers, by federating messages over the Matrix network. The system, named Cactus Comments, was implemented and tested. Seven experiments were conducted to verify correct functioning and fault tolerance of the system. We find that Cactus Comments solves or improves all six of the identified problems, and argue that Matrix is a good fit for embeddable web comments.

Future work

Other Applications of Matrix for Web-Embedded Communications

The problems burdening the existing market for web comments might also affect other use cases where embeddable web communications systems are used. Similar problems might gain similar advantages from solutions built using Matrix. Similar use cases include chats sections for live video streams, customer support chat systems, product reviews sections for e-commerce websites. In general, systems that have humans communicate in an ad-hoc manner via an embeddable web client might be suitable applications for the Matrix protocol.

Message-level Cryptographic Secrecy and Integrity

The Cactus Comments was designed to not use any of the end-to-end encryption features that Matrix provides. Comment sections in Cactus Comments are inherently public data, so it was assessed that encryption was not a necessary feature.

However, this also leaves Cactus Comments with no mechanism for verifying that individual messages have not been manipulated by the direct homeservers of a message's sender or receiver. The Matrix standard does specify APIs for out-of-band key verification, as well as a standard for signing and encrypting messages.[2]

Cactus Comments could make use of these to mitigate the possibility of homeserver admins forging messages sent to its' users. Additionally, the encryption features could be used to allow comment section rooms to be private, for instance to support secret internal comment sections, as part of an intranet system.

MSC1772: Matrix Spaces

Currently, one of the Cactus Comments appservice's responsibilities is to propagate power levels and ban lists across a range of associated Matrix rooms. There is work in place to make this appservice functionality obsolete, by adding similar behavior to homeservers.

The Matrix Specification Change proposal 1772 (MSC1772)[11] introduces the concept of Matrix Spaces. A Space is hierarchical structure of rooms, where relationships between rooms are defined through `m.space.child` and `m.room.parent` events. The proposal introduces a mechanism for managing power levels and ban lists across all of the rooms associated with a space.

MSC1772 is seeing a lot of support from the Matrix community, and the consensus seems to be that the change will be accepted in the first half of 2021. Once this lands in the Matrix specification and common homeservers, Cactus Comments should begin to support managing moderation via Spaces, and the previous custom logic should be deprecated. This will again increase compatability with existing software developed for Matrix, by managing moderation cross rooms in a standardized way.

MSC2753: Peeking Via Sync

A current restriction of the Cactus Comment system is that the appservice must be registered with the default homeserver, for anonymous read access to the comment section room to work.

This related to anonymous users not joining the room, but just "peeking" or "previewing" it. Peeking into a room lets users read the messages of a public room without joining it. This is essential for guest access. The alternative is letting each guest user join the room, which would generate a membership event for each page load of the website which contains the comment section. Apart from bloating storage use, this renders the comment section room unreadable in most existing clients. It is essential for guest users to peek into the room, without joining it.

However, Matrix does not currently support peeking over federation. This means that guest access must happen via a homeserver which is guaranteed to have already federated the room. MSC 2753 sets out to resolve this issue, by implementing logic that allows homeservers to support peeking into rooms over federation.[10][2]

When this specification change is implemented, it should allow the web client to to read the comment section rooms anonymously via any homeserver that supports guest registration. The configuration API is already designed with this separation between the default homeserver URL, and the server name to be used when determining the room alias.

References

[1] Matrix.org
Matrix Specification
https://matrix.org/docs/spec/
(last accessed 2020-09-25).

[2] Matrix.org
Matrix Specification: Client-Server API r0.6.1
https://matrix.org/docs/spec/client_server/r0.6.1
(last edited 2020-05-27, last accessed 2020-09-25).

[3] Matrix.org
Matrix Specification: Application Service API r0.1.2
https://matrix.org/docs/spec/application_service/r0.1.2
(last edited 2019-06-30, last accessed 2020-09-25).

[4] Matrix.org
Matrix Specification: Appendices
https://matrix.org/docs/spec/appendices
(last accessed 2020-11-24).

[5] Matrix.org
Matrix Specification: Application Service API r0.1.2 Registration Section
https://matrix.org/docs/spec/application_service/r0.1.2#registration
(last edited 2019-06-30, last accessed 2020-09-25).

[6] Matrix.org
Matrix Specification: Application Service API r0.1.2 Authorization Subsection
https://matrix.org/docs/spec/application_service/r0.1.2#authorization
(last edited 2019-06-30, last accessed 2020-09-25).

[7] Matrix.org
Matrix Specification: Application Service API r0.1.2 Pushing events Subsection
https://matrix.org/docs/spec/application_service/r0.1.2#put-matrix-app-v1-transactions-txnid
(last edited 2019-06-30, last accessed 2020-09-25).

[8] Matrix.org
Matrix Specification: Federation API r0.1.4
https://matrix.org/docs/spec/server_server/r0.1.4

(last edited 2020-05-27, last accessed 2020-09-25).

[9] Matrix.org
Matrix Specification Change Proposal MSC2919: Markup for mathematical messages
https://github.com/uhoreg/matrix-doc/blob/latex/proposals/2191-maths.md
(last edited 2020-10-24, last accessed 2020-11-17).

[10] Matrix.org
Matrix Specification Change Proposal MSC2919: Peeking via sync (take 2)
https://github.com/matrix-org/matrix-doc/blob/matthew/msc2753/proposals/2753-peeking-via-sync-v2.md
(last edited 2020-11-26, last accessed 2020-12-20).

[11] Matthew Hodgson et al.
MSC1772: Matrix spaces
https://github.com/matrix-org/matrix-doc/pull/1772
(last updated 2020-11-17, last accessed 2020-11-18).

[12] Element
Element.io Webpage
https://element.io
(last accessed 2020-11-18).

[13] Fluffychat
Fluffychat.im Webpage
https://fluffychat.im
(last accessed 2020-12-16).

[14] Ecmel Berk Canlıer
Use the Fediverse as a comment section
https://ecmelberk.com/fediverse-comments.html
(posted 2020-06-21).

[15] Tessa Kelly
tesk9/accessible-html
https://package.elm-lang.org/packages/tesk9/accessible-html/latest/Accessibility
(last updated 2018-09-24, last accessed 2020-10-28).

[16] Tessa Kelly (tesk9)
tesk9/elm-html-a11y
https://package.elm-lang.org/packages/tesk9/elm-html-a11y/latest/
(last updated 2018-09-24, last accessed 2020-10-28).

[17] Héctor Ramón (hecrj)
hecrj/html-parser 2.4.0
https://package.elm-lang.org/packages/hecrj/html-parser/2.4.0/
(last updated 2020-08-07, last accessed 2020-11-05).

[18] Evan Czaplicki
elm-lang.org
https://https://elm-lang.org/
(last accessed 2020-11-25).

[19] The OWASP Foundation.
OWASP Top Ten
https://owasp.org/www-project-top-ten/
(last accessed 2020-11-05).

[20] The OWASP Foundation.
WSTG - v4.1: Testing for CSS Injection
https://owasp.org/www-project-web-security-testing-guide/v41/4-Web_Application_Security_Testing/11-Client_Side_Testing/05-Testing_for_CSS_Injection
(last accessed 2020-11-05).

[21] Web Hypertext Application Technology Working Group (WHATWG).
HTML Section 13: Syntax.
https://html.spec.whatwg.org/multipage/syntax.html
(last updated 2020-11-04, last accessed 2020-11-05)

[22] Facebook.
Facebook for Developers: Comments.
https://developers.facebook.com/products/social-plugins/comments/
(last accessed 2020-12-02).

[23] Disqus.
Disqus.com
https://disqus.com/
(last accessed 2020-12-02).

[24] Commento.
Comment.com
https://commento.io/
(last accessed 2020-12-02).

[25] Disqus.
Disqus Incident History.
https://status.disqus.com/history
(last asccessed 2020-12-02).

[26] Disqus.
Disqus Terms of Service.
https://help.disqus.com/en/articles/1717102-terms-of-service
(last asccessed 2020-12-02).

[27] India Today.
Many Indians are moving to Mastodon from Twitter: Why, what is Mastodon and how it works.
https://www.indiatoday.in/technology/features/story/snapdragon-888-5g-from-gaming-to-photography-how-the-new-qualcomm-soc-will-improve-android-phones-in-2021-1746133-2020-12-02
(published 2019-11-8, last accessed 2020-12-02).

[28] Vice.
Gab Is the Alt-Right Social Network Racists Are Moving to
https://www.vice.com/en/article/ywxb95/gab-is-the-alt-right-social-network-racists-are-moving-to
(published 2019-4-5, last accessed 2020-12-02).

[29] The Perspectives Project
The Perspectives Project.
https://perspectives-project.org/
(last accessed 2020-12-05).

[30] Matrix.org
Bridges | Matrix.org
https://matrix.org/bridges/
(last accessed 2020-12-18).

[31] Matrix.org
Clients | Matrix.org
https://matrix.org/clients/
(last accessed 2020-12-18).

[32] Eagain.net
Git for Computer Scientists
https://eagain.net/articles/git-for-computer-scientists/
(last accessed 2020-12-20).

[33] Kript Team
What is a Directed Acyclic Graph
https://medium.com/kriptapp/guide-what-is-directed-acyclic-graph-364c04662609
(last accessed 2020-12-20).

[34] oxalorg
Sakura
https://github.com/oxalorg/sakura
(last accessed 2020-12-20).

[35] MkDocs.org
MkDocs - Project documentation with Markdown
https://www.mkdocs.org/
(last accessed 2020-12-20).

[36] IPFS.io
IPFS Powers the Distributed Web
https://ipfs.io/
(last accessed 2020-12-20).

[37] AreWeDistributedYet.com
Are We Distributed Yet?
https://arewedistributedyet.com/
(last accessed 2020-12-20).

[38] Pinata
Pinata - The Easiest Way to Use IPFS
https://pinata.cloud/
(last accessed 2020-12-20).

[39] Drew DeVault
The unrealized potential of federation
https://drewdevault.com/2020/09/20/The-potential-of-federation.html
(last accessed 2020-12-20).

Appendix A: Application Service Implementation

In this chapter, we will describe how the application service behind Cactus Comments has been implemented. This includes design choices, technology choices, UX decisions and state modelling.

Responsibility and purpose

Remember that application services are passive observers of events on a homeserver. They define namespaces that they have full control over. They listen to all events in their namespace. In our case, the namespace is a set of rooms.

The Cactus Comments application service implements the Application Service API and acts as a client through the Client-Server API.

The purpose of the Cactus Comments application service is to manage comment rooms in collaboration with the homeserver. Its primary responsibility is to create rooms on-demand and to make sure the right users are moderators in the right rooms.

Everyone who participates in a comment section are indirectly interacting with the application service. This happens as all events are sent to the application service by the homeserver. Furthermore, the service should provide a user interface for site administrators and room moderators to interact with. We do this in a specially designated moderation room.

Figure 4: Asking our friendly chat bot for help, using the Element client.

Promoting users in the moderation room should promote them in *all* rooms for the site. When a moderator bans a user in one room, that user should be banned across all the rooms for the site.

When a user is the first to join a particular room in our namespace, we do not create the room if it is a moderation room, but we do if its a room for an already

registered site. The new room is created such that the moderation settings match those provided by the owner of the site.

From here on out, the service needs to make sure there are no discrepancies between the room states and the desired moderation policies.

Application service registration

Application services have to be registered with homeservers[**?**]. Because application services have elevated privileges the server admin has to statically configure them. Otherwise, they would be able to snoop on rooms unbeknownst to the participants. Most importantly, Our application service defines a **room alias namespace**. Defining a room alias namespace gives our service privileges to manage any room in the namespace. This namespace is **exclusive** meaning humans and other applications can *not* create and delete rooms in our namespace.

The application service can be registered with any regular expressions. This regular expressions describes which room aliases are in our namespace. While it can be anything, the default is `#comments_.*` and we will use that in examples. It is configurable.

User experience: initial setup

When designing Cactus Comments, it was important for us that new comment sections could be created dynamically, so a site admin does not need a workflow when creating a new subsite. To this end, we designed our system such that the site admin only needs to register their site the first time. This one-time procedure requires the admin to send a message to our chat bot interface. After typing "register demosite", the bot will invite the admin to a new room, send a helpful message and promoto the user to admin. This is an image of Element after the admin joins the new room:

Figure 5: A website owner registering a site with the application service, using the Element client.

After this, any room joined with the sitename in the alias prefix will be owned by the site administrator and whomever they promote.

Technology Choice

The Cactus Comments application service implements the Application Service Specification[3]. This means we have to expose a HTTP web server. We implemented our web server in the flask microframework and the entire application in Python 3.9. We interact with the Matrix homeserver with pure HTTP calls and not through an SDK using the requests HTTP library. The application is containerized with docker.

The development environment is built with docker-compose and contains only two external services: a local Synapse (Matrix homeserver) accessible at localhost:8008 and a Matrix client (Element) at localhost:8085. Furthermore, 3 users have been pre-registered to make development obstacle-free. The appservice is registered with Synapse and runs as the third service.

State

The application service keeps some application state: the list of registered sites, their owners and moderation configuration. For reliable operations, these need to be stored in a persistent manner.

When we initially drafted our system, we were prepared to use a traditional RDBMS or a key-value store. Ultimately, we ended up carefully modelling all our state in Matrix events and rooms.

To model all state as matrix events and rooms, we need the notion of a moderation room and a comment section room. A comment section room alias is the comment section name appended to the site name. For instance, for the comment section room alias `#comments_demoblog_site405` the site name is `demoblog`, the moderation room is `#_comments_demoblog` and the comment section name is `site405`. As we use underscore as a special seperator, it is not allowed in site names.

The moderation configuration is modelled by the same events (called power levels) used for restrictions in normal chat rooms. When a moderation room initially is created, the creator is invited and promoted to admin.

Now that we have a model in place, we can persist state. Given a room alias, we can look up its id, look up that id and if it already exist, the comment section is in use. The owner is already admin in the room. We can get the moderation configuration by removing the comment section name from the alias and looking up the state of the moderation room. To get a list of all registered sites, we can ask for all joined rooms and filter on alias for moderation rooms. To check if a site has already been registered, we ask if the moderation room exists.

Overall, this makes our service easier to deploy and operate. It is noteworthy

that we modelled everything in Matrix without custom event types. This means that all standard-compliant clients will understand what is going on.