Implementing A Decentralized Messaging Application

Henry Ang
henryang@stanford.edu

Shaohui Guo
shaohui@stanford.edu

Jingyi Bian
jybian@stanford.edu

Abstract

This project implements a decentralized chat messenger application that enables data sovereignty, scalability, and flexibility of either on-premise or cloud deployments. The implementation is based on the Matrix protocol, and we implement the message synchronization mechanism between servers from scratch. Our implementation ensures that the system provides causal consistency, high availability and fault tolerance. Temporary server crashes and network errors do not impact the rest of the system.

I. Introduction

A. Motivation

Most popular chat applications today run in the data centers of big tech companies, and therefore the users have to entrust their data with these companies, hoping that the data is securely stored on the servers that they have no control of. In the recent years, the unpredictable success of cryptocurrencies leads to a revolution in the decentralized economy, and we envision that a decentralized chatting tool may solve the trust and data privacy issues. A decentralized messaging application provides data ownership. All messaging history is on servers owned or trusted by the client. The Matrix protocol is one of the solutions that has gained popularity among corporations and individuals because of its ease of use and abundant security features, and what we decided to base our implementation on this protocol.

B. Background

Matrix protocol is a decentralized, peer-to-peer messaging protocol. [4] [6] It primarily defines two sets of interfaces, the client-server API and the server-server API. [5] [3] The prior allows users to choose the client messaging app that they prefer to use on the Matrix network, e.g. Discord, while the later handles communication and message storage behind the scene. The clear separation of client and server provides great flexibility to the users. Users using different messaging app clients can join the same chat room by connecting to federated Matrix servers. It also allows the user to choose the where the service is hosted. An user can use public hosted servers or set up a private server only for the users' use. In our project, the main target is to explore the properties of Matrix as a distributed system. We implement an application that follows the Matrix server-server specification. For simplicity, we assume that every user hosts a private server and encapsulate both the client and server logic in the same application.

II. Technical Overview of the Matrix Protocol

A. Matrix Protocol

On a high level, Matrix is a protocol for federated servers to achieve decentralized storage of messages. [1] The history of messages is replicated across all the servers that participate in the chat room, and each server equally shares the duty of maintaining the messaging history. Accordingly, there is not a single point of failure in the system. If one server is down, the rest of the servers can still communicate normally, and when the failed server recovers, it can retrieve the missed history from other servers. Behind the scenes, each server maintains the messaging history in an event graph. Later sections will provide more detailed explanation and analysis of event graphs. Whenever a client writes a new message to the client's server, the message is added to the server's event graph locally, and it is then broadcast to all other servers in the federation, each of which will add the event to its own event graph respectively.

B. Event Graph

An event graph represents all the history in a chat room. [4] Each vertex on the graph represents an event in the room, e.g. a message. The edge between events is directed and represents casual and temporary relationship between the events. [7] For example, in fig 1, an edge pointing from event B to event A means that event A is the parent of event B. It implies that, temporally, event B happens after event A, and that, causally, writing the message in event B could be a result of reading the message in event A.

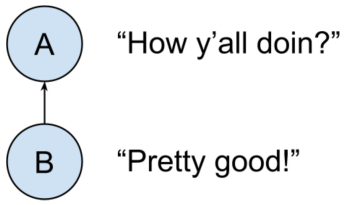

Fig. 1. Vertices and edges in an event graph

Figure2 shows a simple but complete event graph. In an event graph, there is always a single root vertex which has no parent. The first event, Message A, is linked to the root vertex.

Message B and D are sent after two clients see Message A respectively. Message C is sent after a client sees Message B but not Message D. Message E is sent after a client sees both Message C and D. If a new Message F is now sent in the chat room, it will be linked to *all the vertices that do not have children*, which is Message E in this case.

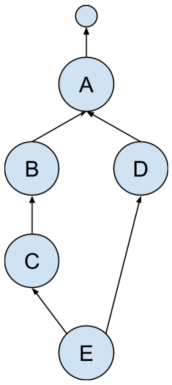

Fig. 2. A simple event graph

C. Concurrency and synchronization of event graph

Normally, an event should only have one parent event, which is the latest message in the chat room. However, when two clients write two different messages to two servers simultaneously, the two servers will temporarily have divergent views of the event graph. Figures 3 and 4 provide an example of how concurrent messages are handled with the event graph.

- In the beginning, only Message A is in the history, and the replicas are identical on the two servers.
- At the next moment, both clients of Server 1 and Server 2 decide to write messages. Client of Server 1 writes Message B and C, while client of Server 2 writes Message D. In each replica's local view, only Message A and the messages just written are in the chat history, thus the two replicas temporarily diverge.
- Then, the two servers broadcast the newly added messages to other servers. Having received the new messages, the two servers now add the new messages from each other to their event graphs, which become in sync again.
- Afterwards, the client of Server 2 sends Message E, which merges the two causal chains in the event graph.

III. Implementation

In this section, we first explain the general architecture of the application and how each module of the application functions. Then we will explain a few key design decisions we made for the system's correctness and optimization.

A. Application Architecture

Our application can be abstracted into the following modules: Event handler, Transport, User interface, Storage, and the

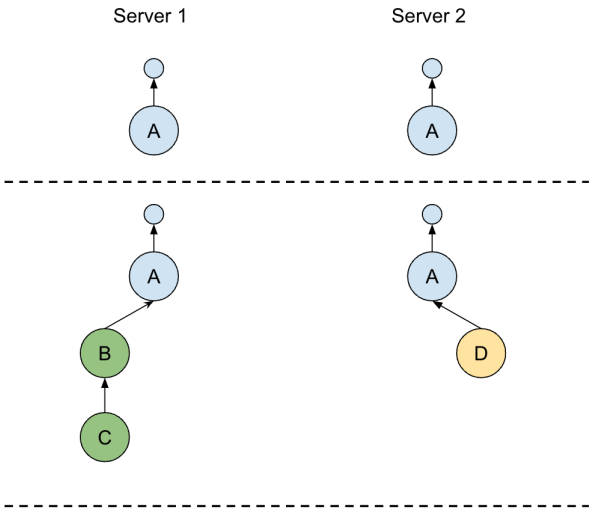

Fig. 3. Current writes (1)

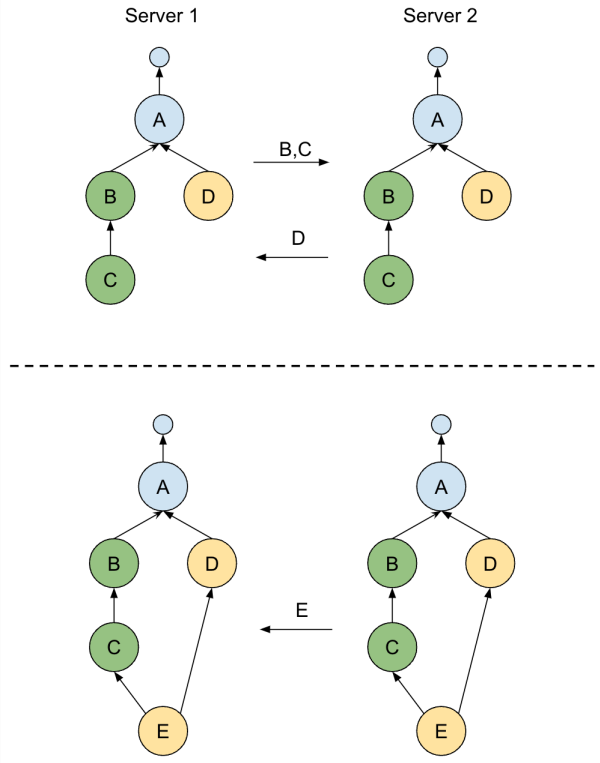

Fig. 4. Current writes (2)

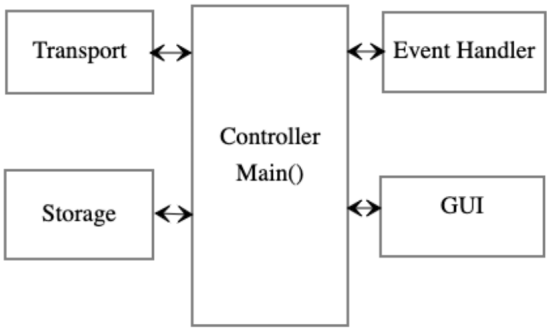

Fig. 5. Application Architecture

Main Controller that coordinates the different modules. Figure 5 illustrates the architecture of the application.

1) **Event Handler** The Event handler contains the core logic of maintaining the event graph and implements the logic for handling all events. It defines the interfaces necessary for the servers to create and synchronize events.
   - AnalyzeMessage: This interface is used to update the event graph. It takes a message and attempts to add it to the event graph. This interface is invoked when the server receives new events from other servers or restores events from logs in storage. For a new event to be appended to the graph, the event's parents must already exist in the event graph. When one or more parents of a new event are not in the graph, it means that the missing parent event might have some issue with transportation, such as order of delivery, or network failure. In this case, the new event will be put in a buffer and wait until all the parent events are added to the event graph. This maintains the invariant that all events that are in the event graph are causally linked. Only events in the event graph can be read by the client.
   - HandleNewMessasge: when the client sends a new message, the Event handler generates a new event that encapsulates the message. It first appends the event to the local event graph, by linking it to all vertices with no children in the event graph. This implies that the new event is causally dependent on its linked parents - that the client sends the message after reading the parent events. It then needs to prepare to send the new vertex and edges (the event and id of its parents) to other servers through the Transport module.
   - GetMessages: this interface queries the event graph and retrieves the desired messages. It may be used for displaying messages in the UI, responding to a request for message history from another server, or updating the local logs. It may retrieve a specific event by the event id, or it could retrieve a range of the event graph. The returned messages will be sorted in the causal order of events. When the event graph splits into multiple branches that are not causally dependent, the events will be ordered by the message timestamp while maintaining the topological order of the graph, so that causal order is never violated.
2) **Transport** The transport module handles communication with other servers and abstracts away the details of sending and receiving events. When sending a new message generated from the local client, the message is broadcast to all the other servers that participate in the chat room. There is no guarantee for the latency or the success of the delivery, but retries could be made if it fails to deliver messages to an active server.
3) **Storage** The storage module is responsible for preserving the event graph in the local disk. Updating the logs on disk for every single new event would be wasteful, so we decide to batch the writes to disk by doing a write every N updates are made to the event graph. The logs in the disk also serve as a starting point for reconstructing the event graph when the application restarts or crashes. It allows the server to recover as much info as possible, before requesting messages from other servers to save network bandwidth. Additionally, all the disk writes are executed asynchronously, so that slow I/O operation would not block other services on the server.
4) **User Interface** The user interface displays real-time messages to the client and collects new messages entered by the client. The display is refreshed each time new messages come and the event graph is updated. The order of displaying follows the order of messages returned by calling the EventHandle's GetMessages interface. When a client sends a message, pieces of metadata are collected, such as the timestamp of the message or the id of the users. They are then passed along with the message to the EventHandler to be appended to the event graph and broadcasted.

B. Design Decisions

Next, we will review a few important design decisions that are crucial to the correctness and performance of the application.

1) **Idempotent updates to the event graph**
   The messages that we transport between servers and write to logs are lists of structured values that encapsulate the message and other metadata that can be translated to idempotent update operations to the event graph. When applying an update operation to the event graph, the application checks if the UUID of the event already exists in the event graph. If it already exists, the operation is a no-op to the graph. If it does not exist, the application checks the validity of the update to decide if the update is applied or rejected. An update operation is valid if and only if all the parents

of the event exist in the event graph. The idempotence of updates is important for a few reasons. First, it eases the requirements for message transportation. When broadcasting new messages to other servers, the server could choose to transmit a few prior messages as redundancy, in case the prior messages were lost in transmission. Also, when servers find missing info and request message histories from other servers, the same message could be received multiple times if the original broadcast eventually arrives. In these cases, the idempotence of the updates ensures that the messages are updated correctly.

2) **Handling of invalid events**
As explained in the previous section, an event may be invalid because its parent events have not been received by the server yet. In our project, we do not assume malicious messages, so missing parent events is the only reason that causes an event to be invalid. When such an event is received, the event handler puts the invalid event in a buffer without updating the state of the event graph. The next time events are received, the events in the buffer and the incoming messages will be used together to update the graph. If the incoming events contain the missing parent events of the events in the buffer, the events in the buffer could be appended to the event graph and removed from the buffer.

3) **Reads of the event graph and causal consistency**
When the user interface refreshes the messages to display to the users, it is formally a read operation from the local event graph replica. The operation only reads the messages that are linked in the graph but not the ones in the buffer. Although it might sometimes be useful to present messages in the buffer to the client before their parent messages arrive, we decided against it to ensure the causal consistency of the system. With the event graph data structure, it is guaranteed that all replicas see the events in the same causal order because a write to the event graph is only valid when its causally proceeding events are in the graph.

4) **Handling of missing events**
Typically, when receiving an invalid event whose parent is unknown to a replica, the reason for the missing event is network delays, and the missing event should arrive later; however, in cases where such an missing event has not been received after a certain timeout, the server will try to request the missing event from other servers. We first attempt requesting it from the server that sends us the invalid event. It is guaranteed that the missing event exists in the event graph of that server.

IV. Evaluation

A. Theoretical Analysis

In this section, we will evaluate the properties of the system theoretically. The CAP theorem states that a distributed system cannot achieve consistency, availability and partition tolerance at the same time. [2] While our system focuses high availability and partition tolerance, it still provides some useful consistency properties - causal consistency. In the previous section, We explained that the use of the causal event graph ensures causal consistency. In this section, we will analyze the availability and partition tolerance.

1) **Availability** The system guarantees that whenever a read or write request is made to an active node, the system will return a non error response. Assume the extreme case where all the servers in our system are down but one, the client can still read from the server because the server contains the complete history in its event graph, which may or may not contain the most up-to-date writes. The client can also make writes to the server, as the server will be able to append the message to the event graph and save it to the disk. The new message cannot be broadcast to other faulty nodes now, but it will eventually be synced when the other nodes recover.

2) **Partition tolerance** When the nodes of our service are partitioned, the connected nodes can still function like a sub-chat room. The partitioned groups will develop divergent causal chains in their event graphs, and those causal chains will be replicated across all the replicas in the same connected group. When the network is recovered, the divergent causal chains will be merged and replicated in all nodes.

B. Empirical Experiment

This section demonstrates the experiments that we conducted to test our implementation.

1) **Network partition** In this experiment, we tested what happens when the nodes are partitioned in our system. We partitioned 3 servers into 2 partitions. We simulated this by blocking one server's communication module, so that it will not be able to communicate with the rest of the two. We see in figure 6 that initially, the three servers were connected and the first three messages were in sync. We then disconnected the right one, and as a result, the rest of messages in figure 6 were out of sync. The left and middle were still in sync because they are connected, while the right one only sees its own messages. Then, Figure 7 shows what happened after we reconnected the right server - the messages sent during the partition were synced to all the servers.
2) **Program Recovery** We also experimented with what happens when application crashes. After sending a few messages into the chat room, we terminated one of the servers in command line. We sent some extra messages

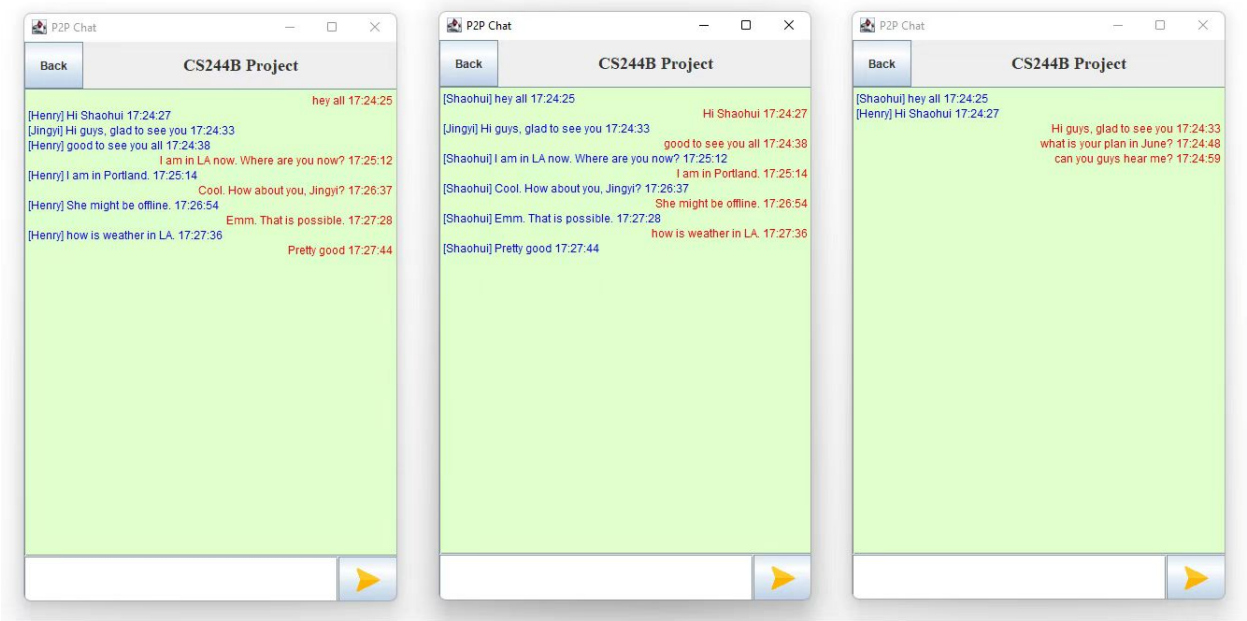

Fig. 6. Partition experiment (1)

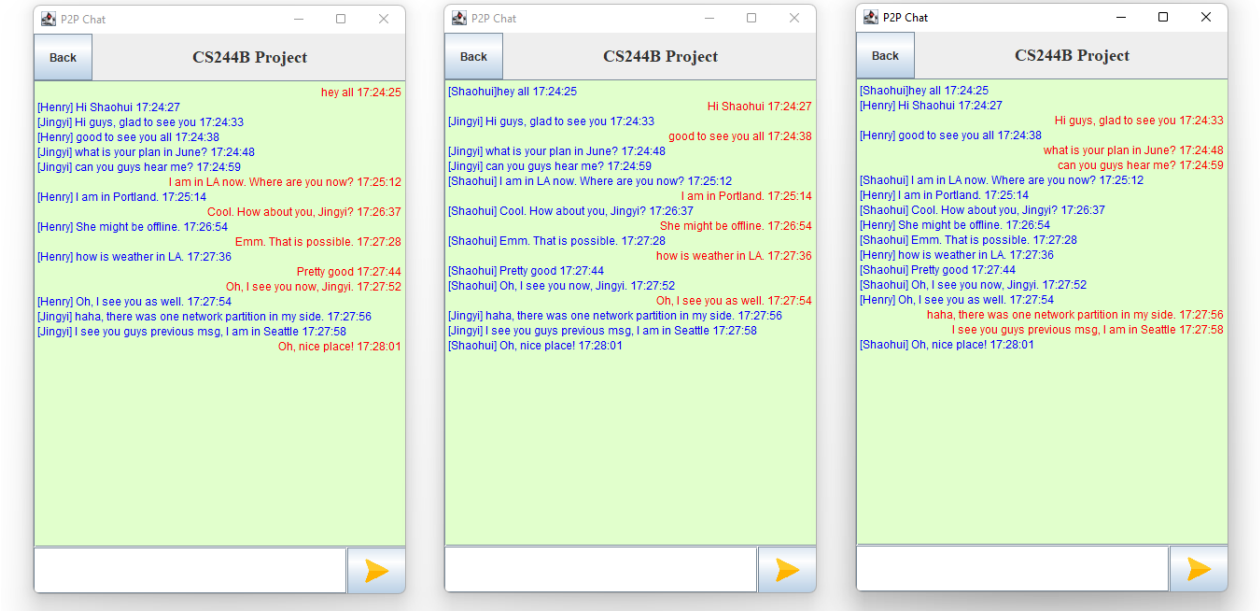

Fig. 7. Partition experiment (2)

on the other servers when this server was still down. When it was restarted, it restored the chat history as expected and received the messages that it missed when it was unavailable.

V. Future works

A. Encryption

In our project, since we focused on building the core synchronization functionalities, we did not have time to implement encryption features for this messaging application. End-to-end encryption would provide great value to the users of this decentralized messaging application, because privacy is one of the main benefits provided by having control over the data.

B. Message Caching

In our implementation, the entire event graph is stored in the memory, which makes the application memory-hungry. One optimization we could make is to only load the recent messages that the client is likely to read. If the client needs

messages that are further back in time, they can be loaded from the disk logs on demand.

VI. Conclusion

In this project we implement a decentralized instant messaging application based on the Matrix protocol. We provided an overview of the protocol and the mechanisms of the event graph. We also explained how we implemented the application and discussed the key design decisions. We evaluated the system and concluded that the system guarantees both availability and partition tolerance. It also provides causal consistency, which is an important property for a messaging application. Decentralized messaging applications have not been widely adapted in the public, but with increasing awareness of data privacy and growing computing resources, it may be a good alternative to the popular messaging apps today.

VII. Code

Link to our Github repo:
https://github.com/heatherbian/CS244Project

References

[1] Alex Auvolat. Making federated networks more distributed. *2019 38th Symposium on Reliable Distributed Systems (SRDS)*, 2019.

[2] Eric A. Brewer. Towards robust distributed systems (abstract). *Proceedings of the nineteenth annual ACM symposium on Principles of distributed computing - PODC '00*, 2000.

[3] Florian Jacob, Luca Becker, Jan Grashöfer, and Hannes Hartenstein. Matrix decomposition. *Proceedings of the 25th ACM Symposium on Access Control Models and Technologies*, 2020.

[4] Florian Jacob, Carolin Beer, Norbert Henze, and Hannes Hartenstein. Analysis of the matrix event graph replicated data type, Nov 2020.

[5] Florian Jacob, Jan Grashöfer, and Hannes Hartenstein. A glimpse of the matrix. *Proceedings of the 20th International Middleware Conference Demos and Posters on ZZZ - Middleware '19*, 2019.

[6] Anirban Kundu. Decentralised indexed based peer to peer chat system. *2012 International Conference on Informatics, Electronics amp; Vision (ICIEV)*, 2012.

[7] Leslie Lamport. Time, clocks, and the ordering of events in a distributed system. *Concurrency: the Works of Leslie Lamport*, 2019.