WHITEBOARD C

Submitted by:

Amirhosein Rajabi

Petteri Pulkkinen

Sonika Baniya

Chapter 1: System architecture

1.1 High level Architecture of System

We have a strong system designed to handle the concurrency and reliability of the collaborative whiteboard tool. At the core of our system, we have Erlang as the backend supervisor which manages board supervisor(s) and board manager. Clients enter whiteboards with web socket connections. And the board controller manages these web socket connections for real-time communication. The database server serves the purpose of data storage while cache server enhances performance. Furthermore, the architecture consists of HTTP REST component which follows the Cowboy REST flowchart (the framework used for implementation of this protocol in project).

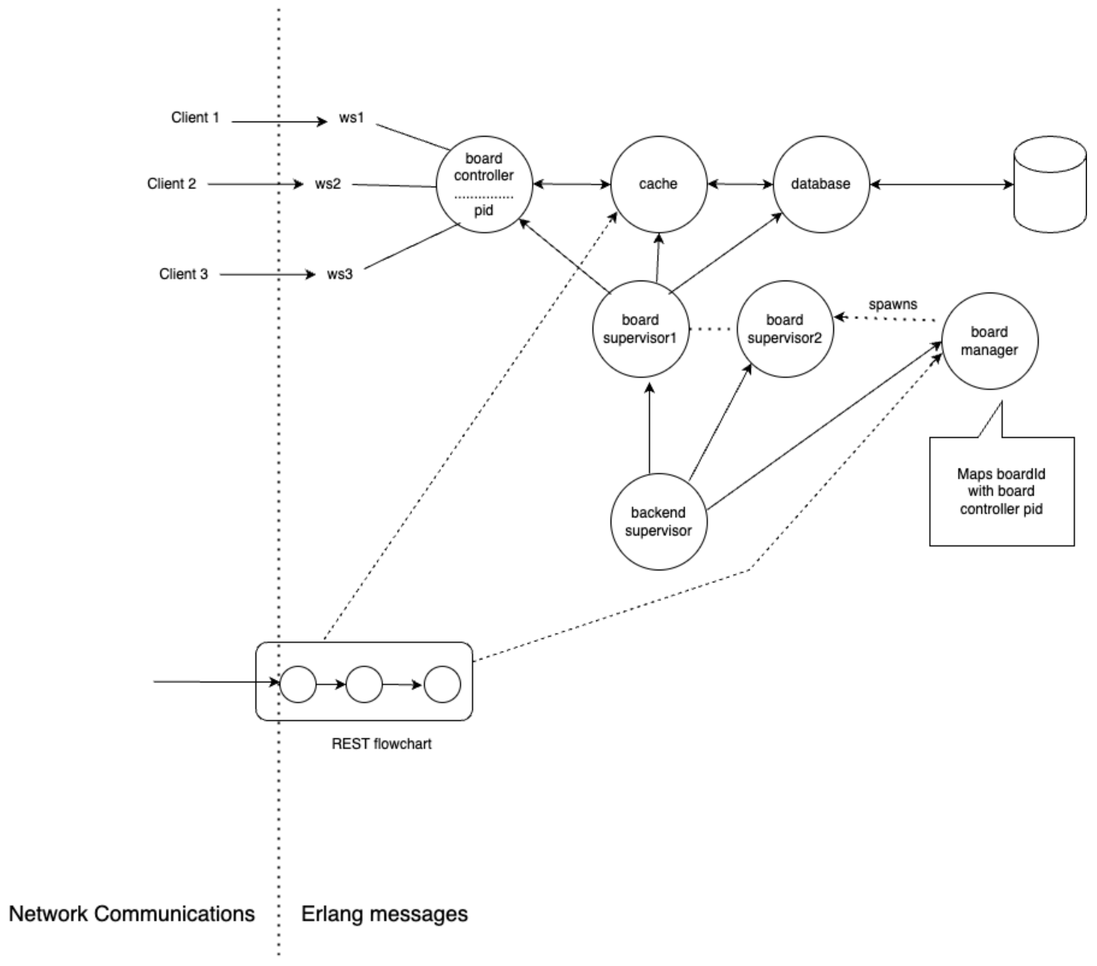


Fig: High level architecture diagram

1.2 Functional Requirements

The following list outlines the functional use cases for our whiteboard system. These use cases cover various aspects of creating, sharing, and collaborating on whiteboard sessions, as well as specific features such as drawing, adding images, and utilizing sticky notes. Each use case describes a specific action that users can perform within the system, ranging from basic functionalities like creating and joining sessions to more advanced features such as real-time collaboration and exporting boards.

- Creating a whiteboard
- Open an existing whiteboard / Join whiteboard session
- Sharing a whiteboard
- General features for whiteboards
- Undo last action
- Redo an action
- List of active users of the (current) whiteboard

- See actions made by other users in real time
- Drawing feature
- Draw on a whiteboard
- Erase drawing on a whiteboard
- Sticky notes feature
- Remove a sticky note
- Edit text on a sticky note
- Moving sticky notes
- Image feature
- Upload an image (to add it on the board)
- Remove an image
- Move an image
- Comment on a canvas object
- View comment on the canvas
- Remove a comment on image
- Export board feature
- Export canvas as JPEG or PNG

The details of each use case is documented in project repository docs/usecases.md

1.3 Non-Functional Requirements

1.3.1 Security

- Joining the session should not be possible without knowing the session id
- Whiteboard URL as the session ID should be hard to guess
- Communication is encrypted

1.3.2 Latency & User Experience

- The users should not feel a lag when drawing or viewing other users' actions
- At least 2 users should interact without interference on each board.

Chapter 2: Design

2.1 State diagram

The diagram below shows the state flow diagram for the object reservation of concurrent communication.

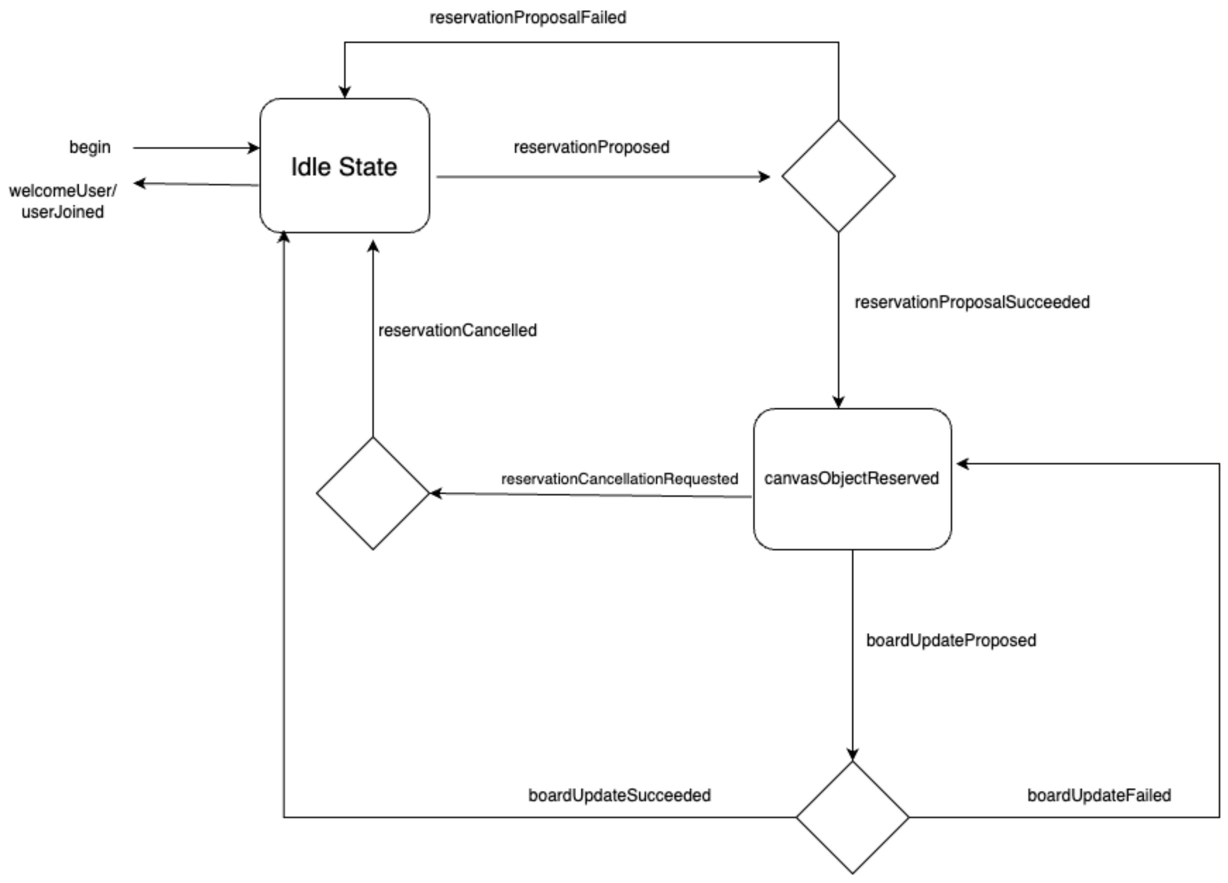


Fig: State diagram in same object canvas of whiteboard

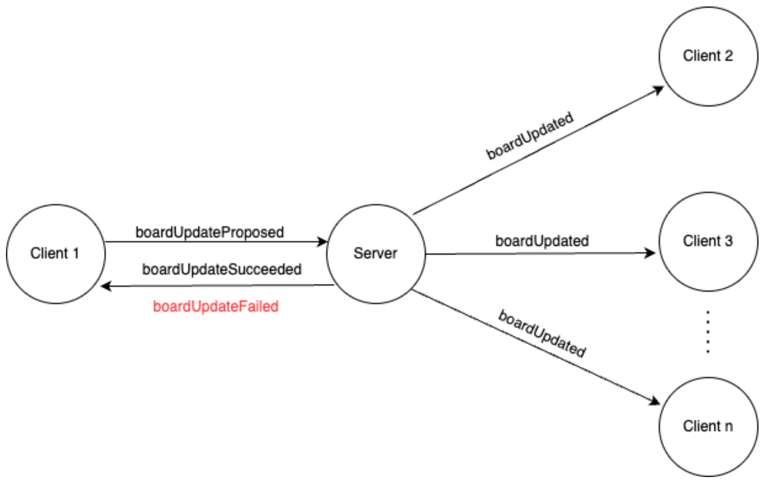
The diagram illustrates a protocol for managing the creation or updating of canvas objects, consisting of two states: the idle state, where the system awaits requests for canvas object creation or updates, and the canvas object reserved state, indicating that a canvas object creation or update process is in progress. Transitions between these states occur through various communication messages, including reservationProposal, reservationProposalSucceeded, reservationProposalFailed, reservationCancellationRequested, reservationCancelled, reservationExpired, canvasObjectReserved, boardUpdated, boardUpdateProposed, boardUpdateFailed, and boardUpdateSucceeded, facilitating the coordination and synchronization of actions within the protocol.

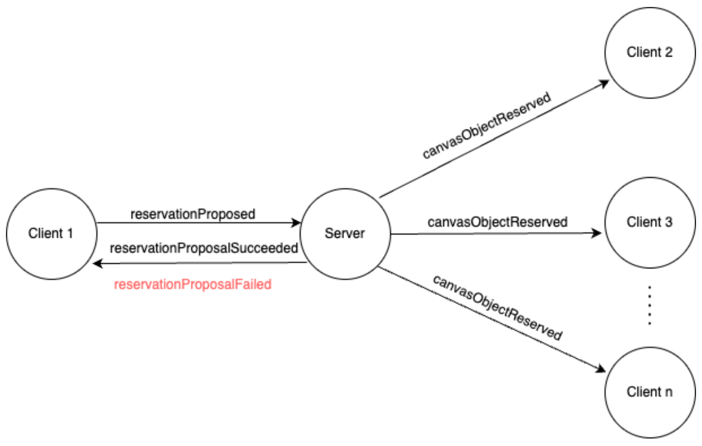
2.2 Communication messages

The supervisor (Erlang, Supervisor) is responsible for starting, stopping, and monitoring its child processes. The basic idea of a supervisor is that it must keep its child's processes alive by restarting them when necessary.

Our asynchronous communication has 16 distinct messages, each serving a specific purpose within the system. These messages include: "begin," "reservationProposed," "reservationProposalSucceeded," "reservationProposalFailed," "reservationCancellationRequested," "reservationCancelled," "reservationExpired," "canvasObjectReserved," "userJoined," "userLeft," "welcomeUser," "boardUpdated," "boardUpdateProposed," "boardUpdateFailed," "boardUpdateSucceeded," "undoRequested," and "redoRequested." These messages are handled across multiple clients and servers, with each message defined as an event according to our coding standard. They inherit common properties of events, facilitating consistent handling and processing. The communication payload associated with each message is defined in our API specification file located at api/api-ws.yaml, where properties and response formats are specified for clarity and consistency.

We can look communications of each mentioned message detail as below





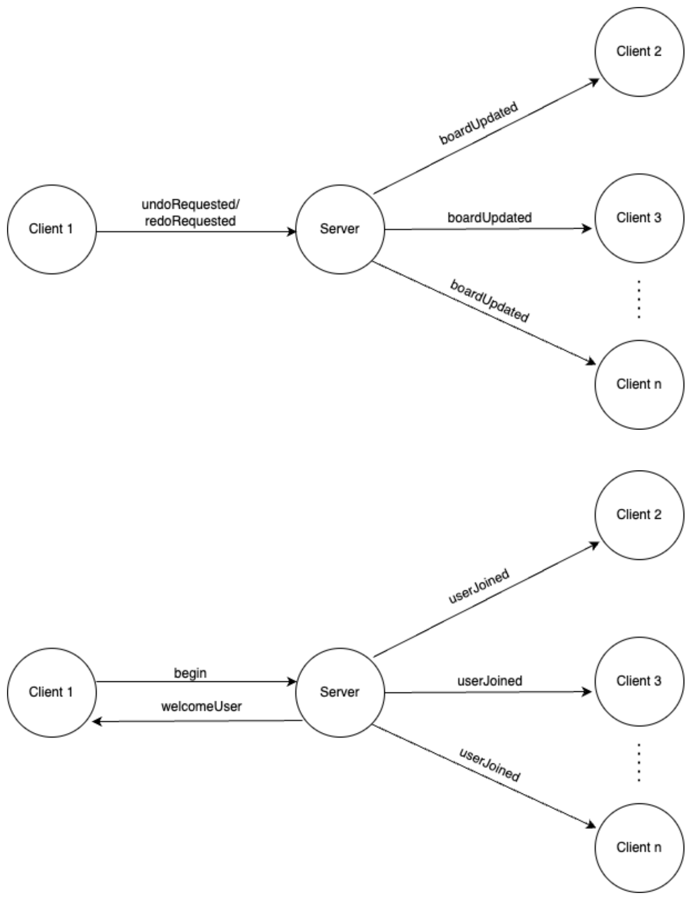


Fig: Messaging communication showing broadcast and response in asynchronous communication

2.3 Sequence diagram

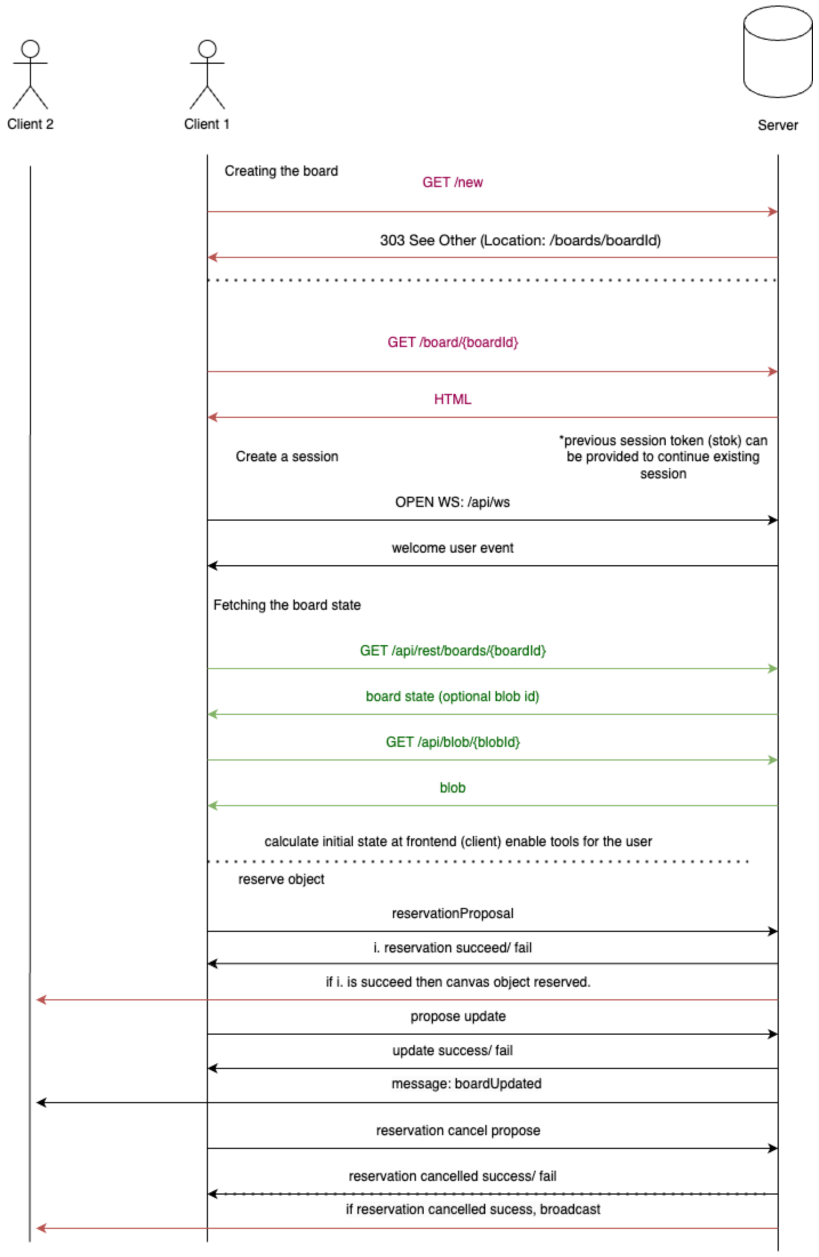

Client 2
Client 1
Server
Creating the board
GET /new
303 See Other (Location: /boards/boardId)
GET /board/{boardId}
HTML
Create a session
*previous session token (stok) can be provided to continue existing session
OPEN WS: /api/ws
welcome user event
Fetching the board state
GET /api/rest/boards/{boardId}
board state (optional blob id)
GET /api/blob/{blobId}
blob
calculate initial state at frontend (client) enable tools for the user
reserve object
reservationProposal
i. reservation succeed/ fail
if i. is succeed then canvas object reserved.
propose update
update success/ fail
message: boardUpdated
reservation cancel propose
reservation cancelled success/ fail
if reservation cancelled sucess, broadcast

Fig: Sequence diagram

2.4 Security Protocol

Our protocol uses the security enhancements of HTTP/2 and WebSockets for the communication. While HTTP/2 offers encrypted connections and header compression, WebSockets provide a secure, bidirectional channel. And the WebSocket handshake initiates over HTTP/1.1, it transitions to the

WebSocket protocol, maintaining security through encryption and allowing for real-time data exchange while minimizing vulnerabilities associated with HTTP/1.1.

To enhance security and protect data transmitted over the network, we have implemented TLS (Transport Layer Security) configuration alongside the existing NGINX setup. TLS ensures encrypted communication between clients and the NGINX server, safeguarding sensitive information from unauthorized access and interception [1]. By integrating TLS into our NGINX configuration, we establish a secure connection for all incoming requests, including those destined for both the front-end and back-end servers.

The NGINX configuration snippet defines three distinct proxy locations to efficiently route incoming requests to different backend servers based on their URL paths. The first location block, designated for requests to /boards/, instructs NGINX to forward these requests to a front-end server listening on port 3000. It also includes directives such as proxy_set_header to transmit pertinent details like the original host, client IP address, and protocol used by the client, ensuring that the backend server can accurately identify and process the request.

The subsequent location block, tailored for requests to /api/rest/, directs NGINX to route these requests to a backend server hosted on port 8080. Like the previous block, it employs proxy_set_header directives to pass relevant HTTP headers to the backend server, aiding in client identification and request processing. Finally, the third location block, targeted at requests to /api/ws/, is configured specifically for WebSocket connections. In addition to specifying the backend server, this block employs proxy_http_version 1.1 to support WebSocket protocol upgrades and proxy_set_header directives to manage the necessary headers for upgrading the connection. Overall, the NGINX configuration optimizes request handling by efficiently directing them to the appropriate backend servers based on their intended destinations while maintaining essential header information for seamless communication.

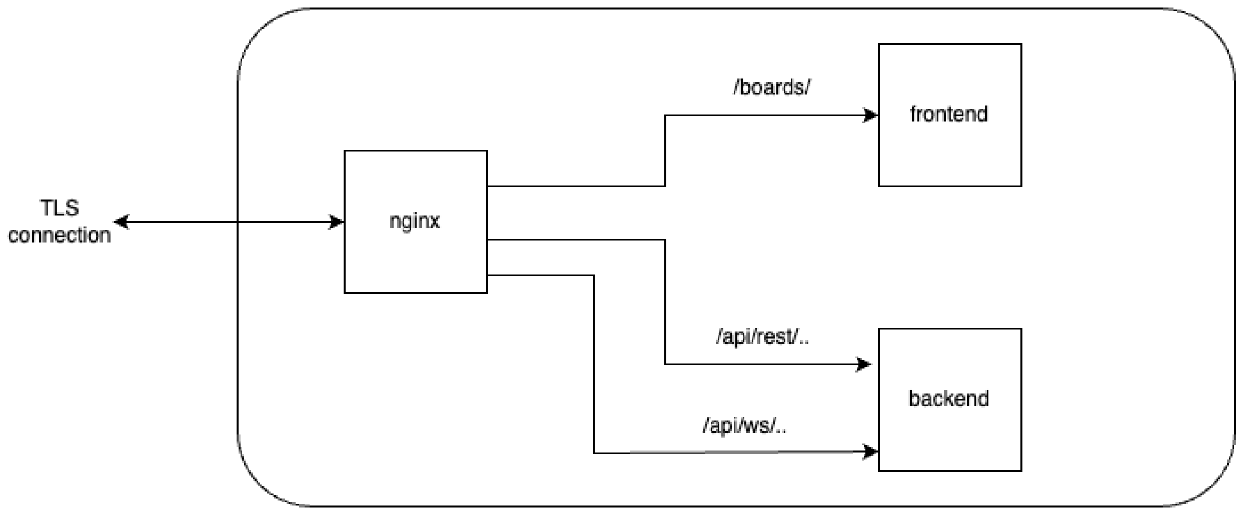


Fig: Implementation of TLS protocol in our system

2.5 Comparison with other communication solutions

2.5.1 In comparison with Design of a shared whiteboard component for multimedia conferencing [6]

In this protocol all the requests are submited but in our protocol the request is only submitted if reservation proposal is succeeded. This ensures optimal resource utilization and minimizes unnecessary network traffic.

Moreover, our protocol exclusively supports text and graphical objects, omitting free drawing capabilities. This streamlined focus enhances clarity and simplifies the user experience.

The changes made in our protocol aren't local copies; they're instantly broadcasted in real-time via WebSockets. This ensures the synchronization across multiple users and eliminates temporary inconsistencies. Additionally, this real-time broadcasting mechanism protects proposed changes from being lost due to simultaneous user actions, enhancing reliability and user confidence in collaborative interactions.

Chapter 3: Implementation and Evaluation

3.1 Implementation Overview

In our system implementation, we've adopted a frontend-backend separation, with the backend developed using Erlang and the Cowboy framework. This choice was driven by Cowboy's seamless support for both HTTP and WebSocket communication, which aligns perfectly with the real-time requirements of our project.

Within the backend repository, we've structured our code around a main parent backend supervisor, which serves as the entry point for managing various components. Specifically, it oversees the operation of multiple parallel board supervisors and a board manager. The board supervisors are responsible for handling individual boards, managing caching, interfacing with the database, and controlling board-related operations.

HTTP requests from users are processed using the REST framework provided by Cowboy. This allows us to define routes and handlers to parse incoming requests, ensuring that our backend can effectively respond to user interactions through standard HTTP methods.

On the front-end side, we've chosen the React framework to develop our user interface. React's component-based architecture and declarative approach make it well-suited for building interactive and dynamic web applications. By leveraging React, we aim to create a modern and user-friendly interface that seamlessly communicates with the backend server implemented in Erlang.

Some of our important implementation decisions for the core functionality of whiteboard were:

- We treat the canvas as several abstract layers on top of each other.
- Only one layer is the ink layer which stores the points because of drawing on the canvas.
- Other layers correspond to the movable objects (sticky notes, comments, images) based on their Z-value.

3.1.1 Important protocol design decision while multiple user tries to access the same resource

Our protocol uses a reservation system for updating objects on the whiteboard. When Client 1 initiates a change, it sends a reservation proposal to the server. Upon receiving a successful proposal confirmation from the server, Client 1 proceeds with the update. However, if Client 2 attempts to update during this period, the server cancels Client 1's reservation proposal, allowing Client 2 to make their changes instead. This ensures orderly and conflict-free object modifications on the whiteboard.

3.2 Experimentation Setup

In our testing framework, we use Gun as the HTTP client to facilitate automated tests directed towards our backend system. These tests are designed to client interactions by sending HTTP requests to various endpoints within the backend infrastructure. To simulate multiple clients concurrently, we use concurrent requests within each test case, utilizing Gun's concurrency features or by spawning Erlang processes. We measure the latency of each request-response cycle to the system's responsiveness under different load conditions. This measurement will be capturing timestamps before and after sending each request, enabling us to calculate the time and assess latency levels.

In our testing framework, we use both the Erlang Common Test (CT) and EUnit test runners to comprehensively evaluate the functionality and reliability of our tool. The Erlang Common Test (CT) framework is employed for conducting integration tests, focusing on verifying the interaction and interoperability of different components within the system. With CT, we use test suites that encompass multiple modules and evaluate their behavior in conjunction, ensuring seamless integration and compatibility across the entire system architecture.

3.3 Test cases

3.3.1 User session management

This scenario simulates multiple users interacting with a whiteboard. Users are created, joined to the whiteboard session, and their interactions are monitored, including joining and leaving events. The scenario also tests the ability to gracefully handle unexpected disconnections and reconnecting users.

3.3.2 Large blobs uploading and downloading

This scenario focuses on uploading and downloading large files, such as images, to and from the whiteboard. It tests the system's ability to handle large data transfers efficiently and verify that uploaded blobs are successfully stored and retrievable.

3.3.3 Database persistence

This scenario tests the persistence of data in the backend database. It involves creating objects on the whiteboard, disconnecting users, and then reconnecting to verify that the board's state remains intact even after disconnections and reconnections.

3.3.4 Blobs database persistence

Similar to the database persistence scenario, this scenario specifically focuses on testing the persistence of blob data (e.g., images) in the database. It verifies that blobs are correctly stored and deleted when necessary, even after users disconnect and reconnect.

3.3.5 Update sticky note

This scenario tests the ability to update objects on the whiteboard, specifically sticky notes. It involves creating, updating, and reserving sticky notes, as well as testing undo and redo functionalities to ensure proper state management and user interactions.

3.3.6 Extend reservation

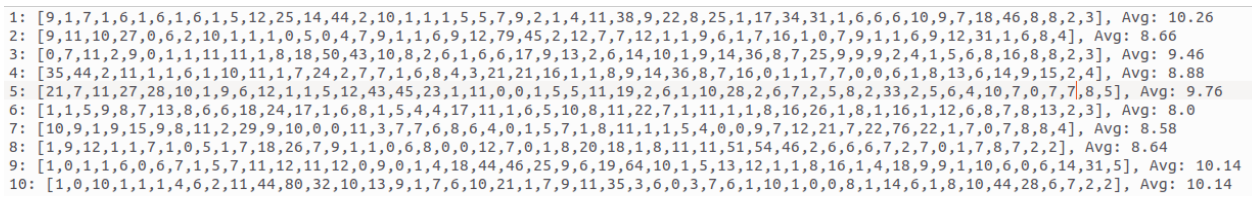
This scenario focuses on testing the reservation system for objects on the whiteboard. It involves reserving an object, extending the reservation, canceling it, and verifying that undo and redo operations behave as expected in the context of object reservations.

3.3.7 Undo redo scenario

This scenario specifically tests the undo and redo functionalities on the whiteboard. It involves creating and updating objects, reserving and canceling reservations, and then performing undo and redo operations to verify that the system correctly restores previous states of the whiteboard.

3.4 Result Analysis

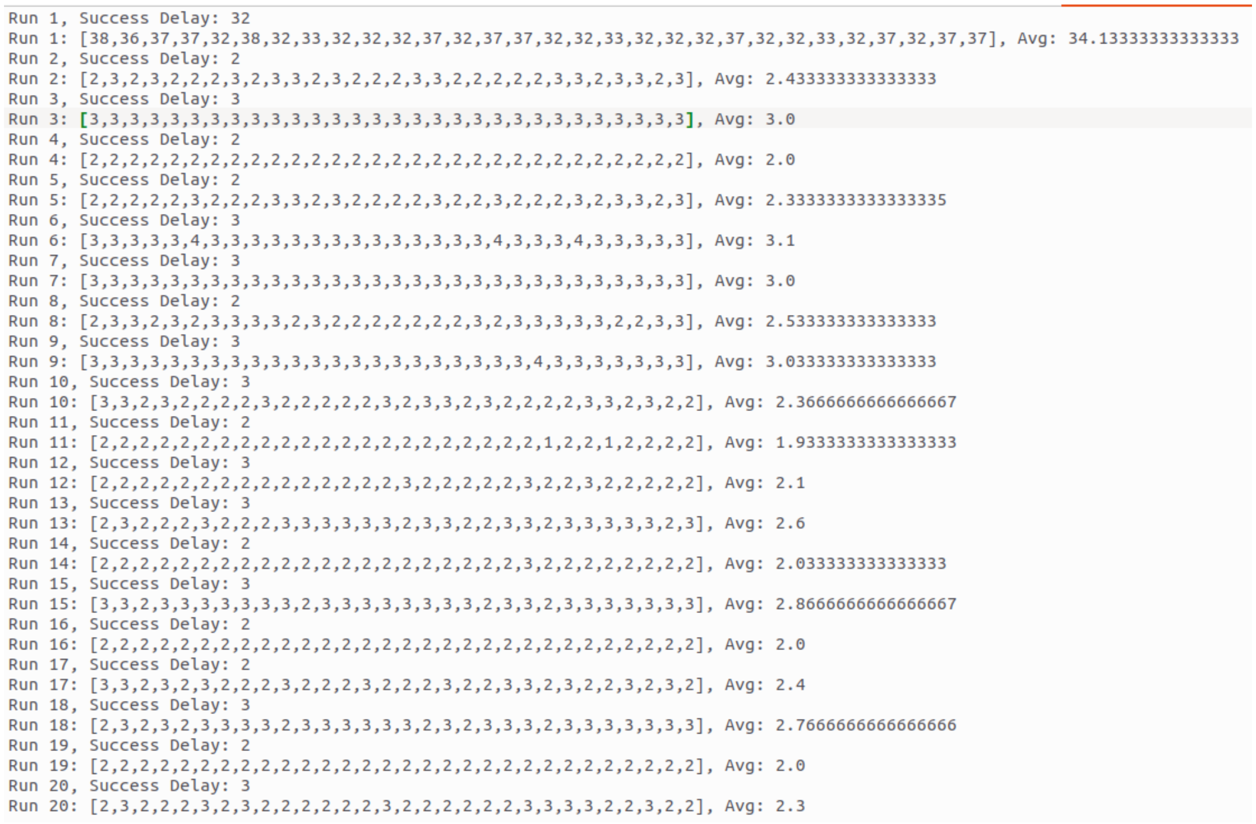
We had 2 benchmark tests. In the first test, we simulate 10 users on the board when they simultaneously draw 50 segments. Since the user is notified of their update success/failure after updating board and informing other board users, we measured the time to receive board success message. We can see the average latency of board updates for each user as below:



```
1: [9,1,7,1,6,1,6,1,6,1,5,12,25,14,44,2,10,1,1,1,5,5,7,9,2,1,4,11,38,9,22,8,25,1,17,34,31,1,6,6,6,10,9,7,18,46,8,8,2,3], Avg: 10.26
2: [9,11,10,27,0,6,2,10,1,1,1,0,5,0,4,7,9,1,1,6,9,12,79,45,2,12,7,7,12,1,1,9,6,1,7,16,1,0,7,9,1,1,6,9,12,31,1,6,8,4], Avg: 8.66
3: [0,7,11,2,9,0,1,1,11,11,1,8,18,50,43,10,8,2,6,1,6,6,17,9,13,2,6,14,10,1,9,14,36,8,7,25,9,9,9,2,4,1,5,6,8,16,8,8,2,3], Avg: 9.46
4: [35,44,2,11,1,1,6,1,10,11,1,7,24,2,7,7,1,6,8,4,3,21,21,16,1,1,8,9,14,36,8,7,16,0,1,1,7,7,0,0,6,1,8,13,6,14,9,15,2,4], Avg: 8.88
5: [21,7,11,27,28,10,1,9,6,12,1,1,5,12,43,45,23,1,11,0,0,1,5,5,11,19,2,6,1,10,28,2,6,7,2,5,8,2,33,2,5,6,4,10,7,0,7,7,8,5], Avg: 9.76
6: [1,1,5,9,8,7,13,8,6,6,18,24,17,1,6,8,1,5,4,4,17,11,1,6,5,10,8,11,22,7,1,11,1,1,8,16,26,1,8,1,16,1,12,6,8,7,8,13,2,3], Avg: 8.0
7: [10,9,1,9,15,9,8,11,2,29,9,10,0,0,11,3,7,7,6,8,6,4,0,1,5,7,1,8,11,1,1,5,4,0,0,9,7,12,21,7,22,76,22,1,7,0,7,8,8,4], Avg: 8.58
8: [1,9,12,1,1,7,1,0,5,1,7,18,26,7,9,1,1,0,6,8,0,0,12,7,0,1,8,20,18,1,8,11,11,51,54,46,2,6,6,6,7,2,7,0,1,7,8,7,2,2], Avg: 8.64
9: [1,0,1,1,6,0,6,7,1,5,7,11,12,11,12,0,9,0,1,4,18,44,46,25,9,6,19,64,10,1,5,13,12,1,1,8,16,1,4,18,9,9,1,10,6,0,6,14,31,5], Avg: 10.14
10: [1,0,10,1,1,1,4,6,2,11,44,80,32,10,13,9,1,7,6,10,21,1,7,9,11,35,3,6,0,3,7,6,1,10,1,0,0,8,1,14,6,1,8,10,44,28,6,7,2,2], Avg: 10.14
```

Latency for each user is around 10ms when 10 users has joined the board and are updating the board at the same time.

In another benchmark test, we measured the latency of other users receiving board updates. We simulated 31 users joining a board where 1 user updates the board. Then we measured the time it takes for others to receive the event. We repeated the experiment for 20 times.



```
Run 1, Success Delay: 32
Run 1: [38,36,37,37,32,38,32,33,32,32,32,37,32,37,37,32,32,33,32,32,32,37,32,32,33,32,37,32,37,37], Avg: 34.13333333333333
Run 2, Success Delay: 2
Run 2: [2,3,2,3,2,2,2,3,2,3,3,2,3,2,2,2,3,3,2,2,2,2,2,3,3,2,3,3,2,3], Avg: 2.433333333333333
Run 3, Success Delay: 3
Run 3: [3,3,3,3,3,3,3,3,3,3,3,3,3,3,3,3,3,3,3,3,3,3,3,3,3,3,3,3,3,3], Avg: 3.0
Run 4, Success Delay: 2
Run 4: [2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2], Avg: 2.0
Run 5, Success Delay: 2
Run 5: [2,2,2,2,2,3,2,2,2,3,3,2,3,2,2,2,2,3,2,2,3,2,2,2,3,2,3,3,2,3], Avg: 2.3333333333333335
Run 6, Success Delay: 3
Run 6: [3,3,3,3,3,4,3,3,3,3,3,3,3,3,3,3,3,3,3,3,4,3,3,3,4,3,3,3,3,3], Avg: 3.1
Run 7, Success Delay: 3
Run 7: [3,3,3,3,3,3,3,3,3,3,3,3,3,3,3,3,3,3,3,3,3,3,3,3,3,3,3,3,3,3], Avg: 3.0
Run 8, Success Delay: 2
Run 8: [2,3,3,2,3,2,3,3,3,3,2,3,2,2,2,2,2,2,2,3,2,3,3,3,3,3,2,2,3,3], Avg: 2.533333333333333
Run 9, Success Delay: 3
Run 9: [3,3,3,3,3,3,3,3,3,3,3,3,3,3,3,3,3,3,3,3,3,4,3,3,3,3,3,3,3,3], Avg: 3.033333333333333
Run 10, Success Delay: 3
Run 10: [3,3,2,3,2,2,2,2,3,2,2,2,2,2,3,2,3,3,2,3,2,2,2,2,3,3,2,3,2,2], Avg: 2.3666666666666667
Run 11, Success Delay: 2
Run 11: [2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,1,2,2,1,2,2,2,2,2], Avg: 1.9333333333333333
Run 12, Success Delay: 3
Run 12: [2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,3,2,2,2,2,2,3,2,2,3,2,2,2,2,2], Avg: 2.1
Run 13, Success Delay: 3
Run 13: [2,3,2,2,2,3,2,2,2,3,3,3,3,3,3,2,3,3,2,2,3,3,2,3,3,3,3,3,2,3], Avg: 2.6
Run 14, Success Delay: 2
Run 14: [2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,3,2,2,2,2,2,2,2,2], Avg: 2.033333333333333
Run 15, Success Delay: 3
Run 15: [3,3,2,3,3,3,3,3,3,3,2,3,3,3,3,3,3,3,3,2,3,3,2,3,3,3,3,3,3,3], Avg: 2.8666666666666667
Run 16, Success Delay: 2
Run 16: [2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2], Avg: 2.0
Run 17, Success Delay: 2
Run 17: [3,3,2,3,2,3,2,2,2,3,2,2,2,3,2,2,2,3,2,2,3,3,2,3,2,2,3,2,3,2], Avg: 2.4
Run 18, Success Delay: 3
Run 18: [2,3,2,3,2,3,3,3,3,2,3,3,3,3,3,3,2,3,2,3,3,3,2,3,3,3,3,3,3,3], Avg: 2.7666666666666666
Run 19, Success Delay: 2
Run 19: [2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2], Avg: 2.0
Run 20, Success Delay: 3
Run 20: [2,3,2,2,2,3,2,3,2,2,2,2,2,2,3,2,2,2,2,2,2,3,3,3,3,2,2,3,2,2], Avg: 2.3
```

Success delay indicates the time for the user who updated the board to receive the success message. As can be seen, latency for a single update is around 3ms. One might wonder why the first run takes 30ms and it is proportional to the number of users. The reasons is that the whole test is running on a laptop and gun framework (the http client) sends ping/pong messages in the websocket connection automatically. Since there are 30 gun processes, there would be around 30ms delay for responding to these ping/pong messages right before the previous step of me asurement which affected the first run of measurement. (we verified this behavior with the fact that 30ms disappears if we insert timer:sleep right before measurement to let background processing finish.)

3.5 Known shortcomings and ideas for improvements

One limitation of our protocol is the lack of real-time drawing synchronization. Client 2 can only view Client 1's drawing after it's completed, as updates aren't transmitted instantaneously. This restriction might be inconvenient in collaborative drawing experiences where simultaneous viewing and editing are desired.

User authentication and management will be nice enhancement to the protocol. This will allow users to revisit the previous work in whiteboard.

Our protocol currently doesn't address the karma problem directly. However, implementing measures to manage resource usage, such as monitoring and controlling data transmission rates, could be a valuable improvement to consider. The "karma problem" refers to challenges in ensuring fair resource allocation and preventing abuse or overuse within decentralized networks or protocols. By incorporating such enhancements, we can ensure fairer and more efficient utilization of network resources, contributing to a better overall user experience.

Chapter 4: Teamwork

During our project timeline from February to early March, a significant portion of our focus in weekly meetings was dedicated to protocol design. We invested considerable time and effort into protocol specifications to ensure the project requirements. Following this phase, we transitioned into the implementation stage, where we began translating the finalized protocol designs into code and functional components. For communication, we utilized a Telegram group messaging platform. This allowed for quick and efficient exchange of ideas, updates, and clarifications among team members. Additionally, to facilitate discussions and collaborative work sessions, we booked campus rooms for in-person meetings and very few meetings on Zoom.

Amirhosein Rajabi – Backend implementation

Petteri Pulkinen – Frontend implementation

Sonika Baniya – Documenting the system diagrams, reporting.

References

[1] IETF, The Transport Layer Security (TLS) Protocol Version 1.3
https://datatracker.ietf.org/doc/html/rfc8446

[2] The Reliable, High-Performance TCP/HTTP Load Balancer, https://www.haproxy.org/#desc

[3] https://www.hostinger.com/tutorials/nginx-vs-apache-what-to-use/

[4] https://xmpp.org/extensions/xep-0113.html#sect-idm25722820387520

[5] Marten van Sinderen, Phil Chimento, Luis Ferreira Pires. Design of a shared whiteboard component for multimedia conferencing. Proceedings of the 3rd International workshop on protocols for multimedia systems. 1996. https://research.utwente.nl/files/5408604/proms96.pdf