Distributed System 2024 S1
Assignment Two Report

Yongchun Li (MSE) 1378156 yongchunl@student.unimelb.edu.au

Chapter One General 2
1.1 Background 2
1.2 Technology Selection 2
1.3 System Architecture and Behavior 3
1.3.1 Class Structure 3
1.3.2 Creation, Joining, and Visibility 4
1.3.3 Drawing 5
1.3.4 Manager functions 5
1.3.5 System design 5
Chapter Two Technical Details and Features 6
2.1 Streaming Calls and Synchronous Previews 6
2.2 Flow control 6
2.3 Conflict Detection and Rejection 7
2.4 JWT Authentication 7
2.5 Text Insertion and Adaptive Formatting 8
2.6 Other 8

****Note that Pictures may be compressed to low resolution in PDF, please refer to appendix****

Chapter One General

1.1 Background

This project mandates the creation of a multi-user collaborative whiteboard. Developers are free to choose their technology stack but must implement a set of specified core functionalities. In addition to fulfilling these requirements, our design addresses several detailed issues, enhancing user experience significantly.

As no specific format has been mandated for this report, the style adopted here will be more flexible than that of the first assignment report. Besides describing the system architecture, this document will also include preliminary insights and discussions on certain technologies.

1.2 Technology Selection

This section analyses the choices of technology. At the outset of the design, we have broad categories including: RESTful style with HTTP PUT, GET plus routing; Remote Procedure Calls (RPC) commonly used for microservices interaction; and direct, socket-based streaming of character data. Each approach has its unique features. The RESTful approach is revered in the internet engineering community for its suitability for stateless web requests and minimal coupling between components. However, for a collaborative application like a shared whiteboard, RESTful's reliance on URI-based resource representation to carry state transition information imposes considerable communication overheads. This is due to the closely interlinked nature of user interactions within collaborative apps—far from being merely parallel, users actively monitor each other's actions.

Sockets offer the most flexibility and customization potential, but require significant reinvention of the wheel in terms of infrastructure, where mature solutions already exist.

As for Remote Procedure Calls (RPC), they abstract remote services into interfaces, providing strong coupling and scalability—ideal for GUI applications with defined requirements like this project, which require tight frontend-backend integration or even full-stack development. Java RMI, a focus of this subject, enhances RPC for object-oriented languages with native support for almost all Java data types, collections, queues, hash tables, custom types, and enums, making it highly suitable for rapid prototyping of distributed systems. Despite this, RMI is confined to the Java Virtual Machine and does not offer external service APIs. Its underlying dependency on JDK Serializable suffers from performance issues (being half as fast in serialization and eight times slower in deserialization than its strongest competitors), awkward serialization syntax, and security vulnerabilities. Motivated by a spirit of innovation, I have chosen Google's open-source gRPC as the communication protocol for this system

Regarding the UI, leveraging knowledge from Assignment 1, I continue to use Swing as the visualization technology, despite its lack of support for genuinely multi-threaded drawing, employing event dispatchers (EDT) instead, which has restricted development and even caused some bugs in the early stages.

1.3 System Architecture and Behavior

As mentioned, I have selected gRPC for the transport layer abstraction, paired with the high-performance, cross-language serialization protocol Google Protobuf as the IDL (Interface Definition Language) message format. Each gRPC server instance can host multiple service definitions; accessing each service requires constructing one (or a group of) Stubs from a channel (gRPC's connection layer abstraction), including synchronous, asynchronous, and compatible types.

Given the simplicity of the computational tasks for 2D drawing, the design adopts a traditional client-server model but retains the potential to expand to multiple servers or rearchitect into a decentralized distributed system.

This section will present the system design diagram, introduce the class structure, and explain the behavioral logic of the system.

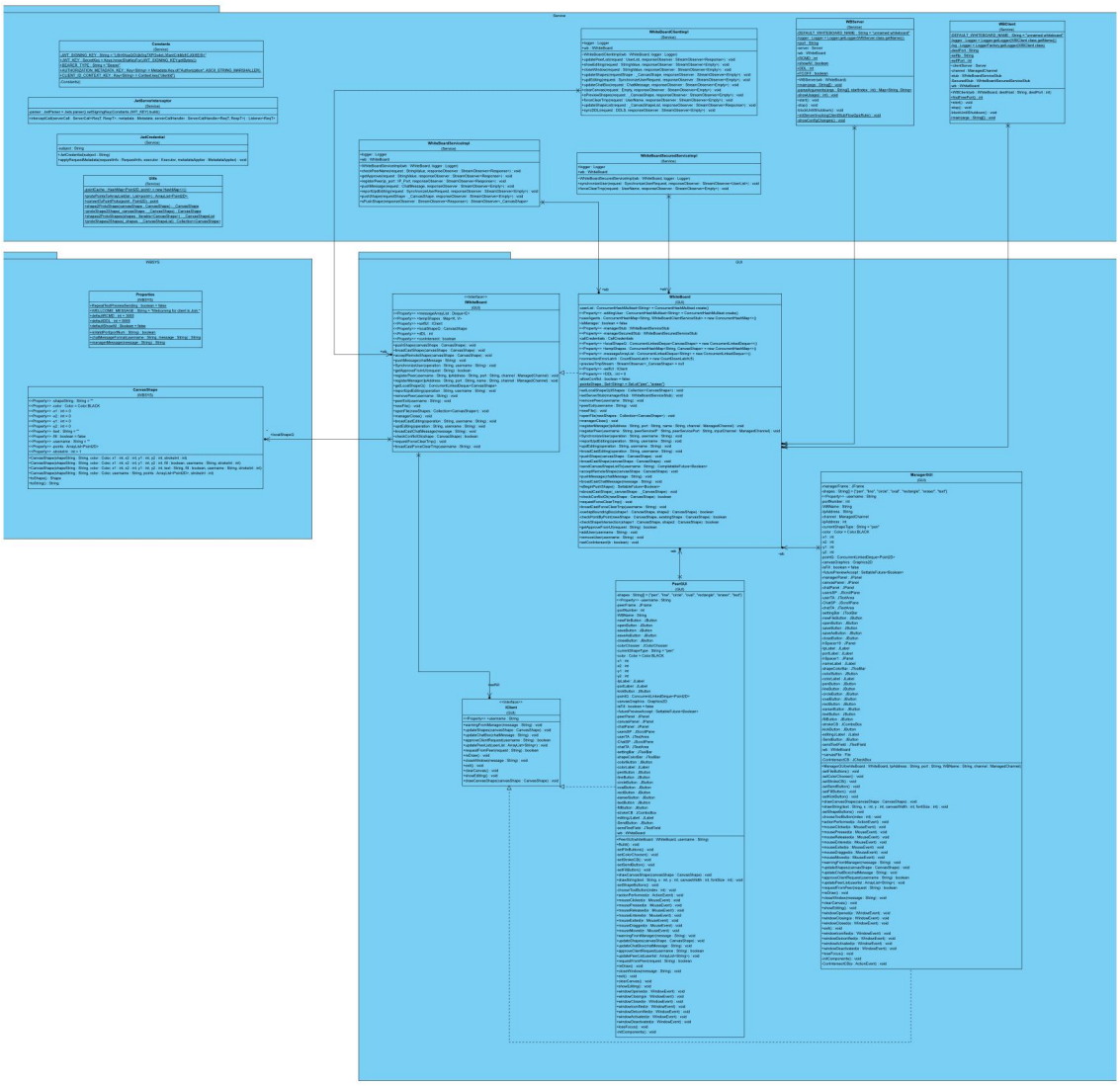

Diagram 1.1 System Class Diagram

1.3.1 Class Structure

The primary classes include two UI codes (Java + Form), which implement the UI layout and associated event listeners. This includes a significant amount of code auto-generated by

JFormDesigner, with the classes themselves handling most of the drawing logic and a small portion of the program logic. WhiteBoard.java acts as the bridge between the UI and the services, serving as both the data class and the primary logic class. These two components are interdependent; the whiteboard class is instantiated at the program's entry point, and the UI is constructed upon successful login. The XXXImpl class implements the RPC method calls as defined in whiteboard.proto. Additionally, there is an authentication component detailed underJWT Authentication .

1.3.2 Creation, Joining, and Visibility

Each client explicitly declares the server IP and port upon joining, establishing a connection to the server's RPC channel, which is then stored as ‘managerStub’ in the WhiteBoard class as a proxy for calling server services.

Per the specifications, the creator of the whiteboard also serves as the manager. Consequently, the server is set up on the manager's terminal using the same IP address—meaning the manager's terminal also functions as a client of the server, similar to other users. It therefore stores the server stub handle in the WhiteBoard class, being uniquely named as "Manager," thereby gaining additional method privileges in the WhiteBoard class.

The server provides services (WhiteBoardService and WhiteBoardSecuredService), and participants are clients of the server (also referred to as peer clients) as well as service providers (WhiteBoardClientService).

Within the WhiteBoard class, there exists a concurrent hash table field, currently dedicated to the manager's instance. This table constructs channels based on the registered client's IP and port, generates stubs, and organizes them using usernames as keys, facilitating efficient traversal and constant-time look-up for initiating calls to specific targets.

Each terminal maintains a user list solely for users display purposes and an editingUser list that is real-time updated with users who are currently drawing. Both are concurrent sets and are synchronized with the server upon joining.

A message list is also maintained where users can only see messages posted after their joining. This feature is inspired by the group design philosophy of social media platforms like WeChat. The display of messages includes the time and the sender (except for some system notifications).

1.3.3 Drawing

Each participant can start drawing from any point and simultaneously preview the movements of all others, including themselves. Due to Swing's lack of support for partial repaints, all graphical updates necessitate a full global repaint. As a result, on high-performance terminals, this leads to flickering updates, while on lower-performance devices, it might cause brief whitening until the preview updates cease and the display stabilizes. To accommodate users on lower-performance systems, the server program can be run with the -RCMD update rate command to limit the rate of broadcast calls, as detailed further in Flow control.

1.3.4 Manager functions

managers possess unique permissions or responsibilities, including the ability to save, save as, open files, close, and command to clear the whiteboard on all terminals, permit or reject the joining request. They can also kick out users by username and set permissions to allow or prohibit concurrent drawing.

Regarding the save and save as functionalities:

- If the current file is newly created, the save and save as functions are equivalent.
- If the file was opened from a local source, the save operation will directly update that file.
- If the user chooses to save to a non-empty target, they will be prompted about overwriting.
- If saving to the current open path, users will be informed, but still makes a successful save.
- Should the save operation fail, users have the option to clear the path and retry.

1.3.5 System interface design

Although proto files are not easily visualized, the automatically generated Impl base code provides a good interface-like method declaration. Here, we present class diagrams for three service Impl classes. These include definitions that have not been implemented, differing from the strong constraints that programming language interfaces impose on implementation classes.

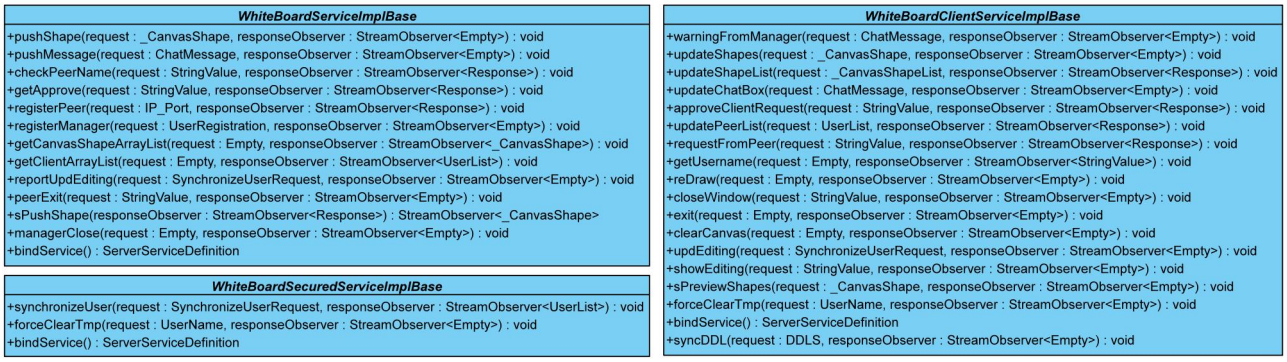

Diagram1.2 gRPC interfaces

Chapter Two Technical Details and Features

This chapter discusses some of the key technical aspects and innovations within the system.

2.1 Streaming Calls and Synchronous Previews

FYR: This section corresponds to the sPushShape() call and the sbroadCastShape method.

I have designed real-time synchronous previews for all users to observe others' work in progress, facilitating efficient collaboration. Most of the gRPC calls in this design are unary and asynchronous. Asynchronous calls are chosen over synchronous due to the latter potentially blocking and inevitably slowing down UI performance. Unary means that each request corresponds to a response, even if it is defined as a null response—in reality, the caller observes this null response's return to execute a callback. However, for previews, there's a twist—during ongoing drawing activities, the system doesn't know when the call should end until the sender ends it voluntarily or the server prematurely rejects it for some reason.

gRPC introduces a new pattern for such scenarios: streaming calls, which can be client-side, server-side, or bidirectional. Though it sounds complex, all other types are derivatives of bidirectional streaming. Given our scenario, client-side streaming is necessary because continuously changing graphics are being sent. Server-side streaming, often used for returning massive data results continuously to prevent poor timeliness of a bulk return, isn’t critical in our system during the preview upload phase—most feedback required is minimal, at most an acceptance or rejection. Thus, the upload phase uses client-side streaming requests with unary server responses.

The preview distribution part triggered more extensive reflection (Note: traditional unary asynchronous methods were used in this design for this part; the following is merely a contemplation). For preview distribution, the server could also use request-side streaming (server calling client’s service, to avoid confusion if still called “client-side”), where once an upload stream is established—or even at the start of the server's setup—the server could simultaneously push to multiple endpoints managed as a collection. However, transmitting rejection or completion statuses becomes problematic, as for the server, the reply of an upload call goes to the uploader (uploader’s WhiteBoard class, UI class, etc.), and stopping the downstream flow would have to rely on nested Future asynchronous results in the upload call’s implementation, which is neither elegant nor practical—because broadcast streams are stateless, allow disorder, and do not require replies. This is why I ultimately used only unary calls for broadcasts.

2.2 Flow control

As mentioned, accommodating weaker performance devices requires lowering the report rate, while effective collaboration necessitates high-speed synchronized views—a trade-off. The system incorporates Alibaba Sentinel as the flow controller. It adopts an annotation paradigm to mark resources and can control the permissible rate of key calls in a blocking manner, thus achieving system rate limiting.

In practice, I made two attempts: initially, intuitively marking server push operations as resources and limiting the rate of stream uploads, which severely blocked the manager's whiteboard response in the UI thread; and the final approach used—designating server whiteboard broadcast calls as the resource to be limited, thus only restricting their broadcast rate and allowing manager to draw at a rate exceeding the broadcast rate.

I have embedded some parameters of the rate limiter in the server program's entry point, as seen in Table 2.1.

Argument	Description	Typical settings & effect
-RCMD	Command rate for synchronizing graphical broadcasts (per second)	200 is barely usable, 800+ is smoother, 2000+ achieves real-time synchronization on an intranet
-SHOWALL	Whether to force broadcast all preview frames	True increases the delay from drawing to graphic effect, False causes intermittent updates at high speeds
-DDL	Drop time (limits the continuation time of a single drawing, also applicable for upload timeout drops)	2 - 8 seconds, depending on network conditions, system sync rate, and the choice of SHOWALL
-FCOFF	Completely turn off flow control, making the above parameters ineffective	Default is false if not entered

Table 2.1 Flow Control Arguments

2.3 Conflict Detection and Rejection

Although all users should have equal editing rights, there's a customary practice in whiteboard collaboration where first-come, first-served rules apply. A user beginning their drawing does not expect to share the space they are outlining until they have completed their design, allowing others to then add or modify without overlapping during creation. Conversely, non-conflicting drawings can proceed concurrently—this is the collaboration model I envision for the whiteboard. Therefore, on the manager's side, I have implemented a checkbox for controlling whether concurrent drawings can overlap. If unchecked, any new drawing that invades an area still being completed by another user will be automatically rejected. Such rejections are cleared from the user's view as soon as the mouse is released.

The specific implementation logic includes an initial boundary box pre-check followed by more precise checks using Area or Shape.intersect. This logic is embedded within the server's gRPC call for preview uploads. If a conflict is detected, the call fails, and the return listener on the client side triggers a boolean Future. This boolean adds a callback to handle the rollback logic, such as deleting and redrawing locally.

2.4 JWT Authentication

Some server-side calls in the system are critical (identified as SecuredService in the protocol) and, if misused, could negatively impact all users' experiences. Reliance on mere message payloads is insufficient to confirm the caller's identity due to potential editable messages (A can claim to be B) and plain-text attacks (A's messages could be tampered with during transmission).

To address this issue, two measures are typically necessary: first, uniquely verifying the caller's identity, and second, securing the transmission (SSL/TLS).

For authentication of critical calls, our design employs JWT (JSON Web Tokens), which offer a level of encryption sufficient for general security needs. SSL encrypts the entire transmission, significantly impacting response speed and throughput (at least one-third slower than plain text). In contrast, JWT merely adds a message header, having minimal impact on system performance.

Constants, JwtCredential, and JwtServerInterceptor comprise a group implementing JWT. Constants defines the content and encryption of trusted stream headers. JwtCredential implements the application of a trusted token to requests, allowing clients to authenticate messages when calling SecuredService on the server. JwtServerInterceptor is the interceptor implementation, added to the server at startup to block critical call requests that lack credentials.

Credentials are generated during the registerPeer call, where the server returns an authentication string to the client, which is then used to attach jwt information based on this string for every critical call according to the agreed encryption algorithm.

Since the server is not remote, the credentials used by the admin terminal for calling secured services are self-issued and thus redundant. However, if the system is expanded such that the server is decentralized, the authentication of the manager will be equally crucial.

2.5 Text Insertion and Adaptive Formatting

This is a minor feature, but one that significantly enhances usability: text insertion is handled through a popup, which automatically formats the text to mitigate concerns of overflowing the screen. The current algorithm adaptively adjusts the line length and shortens it in the final line if necessary to ensure the text ends neatly at the right end of the field block.

2.6 Other

- Centralized Graphic Encoding and Decoding. I have segregated the encoding and decoding of graphics into a separate class. Frequent encoding and decoding constitute a significant overhead in the system. Although protobuf offers robust performance, optimization is an ongoing process. I employ a hashtable to cache already mapped points2D to protoPoint pairs, accelerating the encoding of points.
- The DDL parameter is synchronized to the user side through a call and is slightly increased to ensure that the system's information discarding strategy is broadly consistent.
- The UI layer, service implementation layer, and data layer are separated, each performing their specific roles, which facilitates scalability.

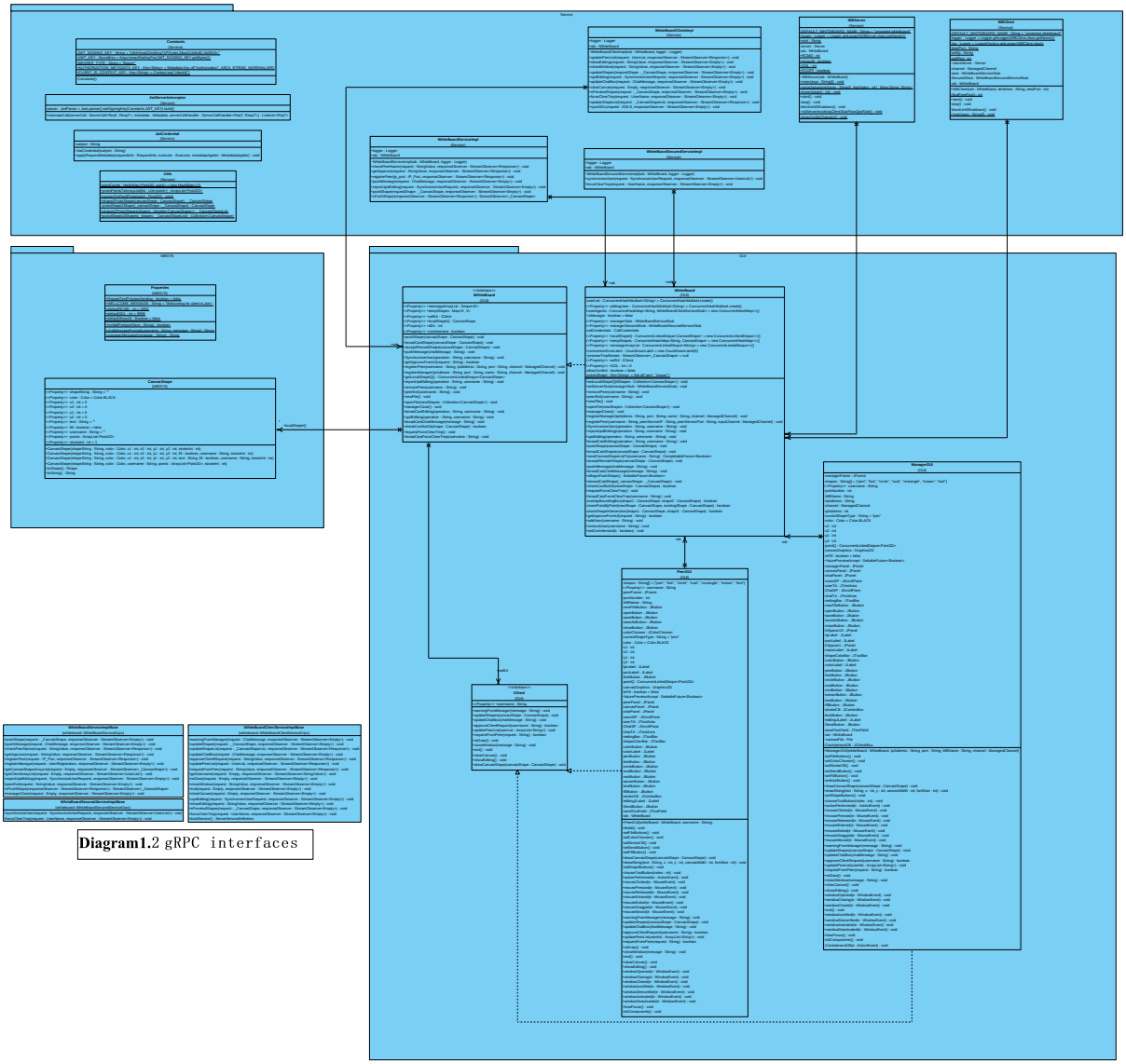

Diagram 1.1 System Class Diagram

Diagram1.2 gRPC interfaces