UNIVERSITY OF MALTA
FACULTY OF INFORMATION & COMMUNICATION TECHNOLOGY
Department of Computer Science

CPS2008 (Operating Systems and Systems Programming II)
Year: 2024

# Title: NetSketch: A Collaborative Whiteboard

**Author(s): Keith Bugeja, Alessio Magro**

## Instructions:

1. This is an **individual** assignment and carries **100**% of the final CPS2008 grade.
2. The report and all related files (including code) must be uploaded to the VLE by the indicated deadline. Before uploading, archive all files into a compressed format such as ZIP. It is your responsibility to ensure that the uploaded file and all contents are valid.
3. Everything you submit must be your original work. All source code will be run through plagiarism detection software. Please read carefully the plagiarism guidelines on the ICT website.
4. Reports (and code) that are difficult to follow due to poor writing style, organisation or presentation will be penalised.
5. Always test function return values for errors; report errors to the standard error stream.
6. Each individual may be asked to discuss their implementation with the examiners, at which time the program will be executed and the design explained. The outcome may affect the final marking.

## Introduction

Collaborative whiteboards have been around for a while, first used in corporate and educational settings for group brainstorming, planning, and idea sharing. With digital advancements, they have moved from the physical to the virtual, becoming key tools for remote learning, online meetings, and creative collaboration worldwide. These tools let multiple users simultaneously draw, write, and modify content on a shared canvas over the internet. They promote teamwork and clear communication, useful for anything from teaching and business planning to creative projects and casual chats. Building a collaborative whiteboard requires tackling real-time network communication, data synchronisation, and designing intuitive user interfaces. It is important to ensure that user drawings and edits are smoothly shared across the network, keeping everyone's work in sync, and providing a seamless experience on various devices and connection speeds.

The aim of this assignment is to create `NetSketch: A Collaborative Whiteboard`. The application will allow multiple users to interact with a shared canvas over a network, demonstrating the principles of network programming and concurrency management. The client component may be implemented in Python or any other language that makes it easy to work with a drawing canvas. The server should be developed in C, C++, or Rust, focusing on handling real-time data synchronisation and network communication efficiently. Your submission will be evaluated based on correctness, structure, style, and documentation. As such, your code must accurately implement the features described, be well-organised into functions and modules, follow good coding practices with clear indentation and descriptive comments (avoiding cryptic variable and function names), and be accompanied by a design document that outlines your implementation approach.

## NetSketch: A Collaborative Whiteboard

Users initiate their NetSketch session by connecting to the server via the command line, providing the server IP address, port, and a unique nickname. This step grants them access to the shared, graphical canvas, where their drawings – made through command-line instructions – will be rendered in real time.

```
$> netsketch --server 192.168.1.66 --port 6666 --nickname artist_name

========================================================
Connected to NetSketch server at 192.168.1.66:6666
========================================================

Nickname artist_name is now active. Ready to draw.
For command list, type ’help’.
```

Listing 1: Connecting to the NetSketch Server

After connecting, users can execute various commands to draw on the whiteboard. Drawing in NetSketch is command-driven, allowing users to create vector graphics such as lines, rectangles, circles, and text.

### Drawing Commands

The user interface supports several commands for creating and manipulating drawings:

1. **help** - Lists all available commands and their usage.
2. **tool** *{line | rectangle | circle | text}* - Selects a tool for drawing.
3. **colour** *{RGB}* - Sets the drawing colour using RGB values.
4. **draw** *{parameters}* - Executes the drawing of the selected shape on the canvas. For text, it requires coordinates and the text string. Each command is assigned a system-wide unique identifier (e.g. a unique positive integer) and is added to a *draw list* – a per-user list of drawing commands.
5. **list** *{all | line | rectangle | circle | text}* *{all | mine}* - Displays issued draw commands in the console, filtered by tool type and/or user.
6. **select** *{none | ID}* - Selects an existing draw command to be modified by a subsequent draw command.

7. **delete** *{ID}* Deletes the draw command with the specified ID.

8. **undo** - Reverts the user’s last action.

9. **clear** *{all | mine}* - Clears the canvas. Using the argument ‘mine‘ clears only the draw commands issued by the user; without it, the entire canvas is cleared.

10. **show** *{all | mine}* - Controls what is displayed on the client’s canvas. Specifying ‘mine‘ will display only the commands issued by the user.

11. **exit** - Disconnects from the server and exits the application.

### Examples

Specific drawing actions are initiated through the *‘draw’* command, with parameters varying based on the selected tool.

```
tool rectangle
colour 0 255 0
draw 100 100 200 150
```

Listing 2: Example of Drawing a Rectangle

This sequence prepares the user to draw a rectangle with the top-left corner at (100, 100) and the bottom-right corner at (200, 150) in green.

```
tool text
colour 0 0 255
draw 150 150 "Hello from NetSketch!"
```

Listing 3: Adding Text to the Canvas

Here, the user decides to add text to the canvas, placing the message “Hello from NetSketch!” at the coordinates (150, 150) in blue.

```
list all mine

[0] => [rectangle] [0 255 0] [100 100 200 150]
[1] => [text] [0 0 255] [150 150 "Hello from NetSketch!"]
```

Listing 4: Displaying draw list

In this instance, the user lists all the drawing operations they have issued so far, with each operation having a unique identifier and encapsulating state such as colour and active tool, in addition to the actual drawing command.

```
select 1

colour 255 0 0
draw 100 100 "Bye!"

list text mine

[1] => [text] [255 0 0] [100 100 "Bye!"]
```

Listing 5: Modifying existing draw commands

Using the *‘select’* command followed by the *‘draw’* command identifier allows subsequent state changes to be applied to an existing command. The ID is deselected once a new draw command is issued that modifies the existing selection. Here, the text “Hello from NetSketch!” is changed to “Bye!” in red and moved to the coordinates (100, 100).

### Additional Notes

Draw commands are automatically appended to the user’s draw list. However, *‘select’* and *‘delete’* may affect draw commands from other users, as they operate on a unique identifier that identifies commands system-wide. Note that *‘clear’* may also affect draw commands from other users, even though it doesn’t take an ID.

When a user disconnects, or is disconnected, any issued draw commands should not be lost. If the user reconnects within one minute of disconnecting, their draw list should be available to them. Otherwise, the server should ’adopt’ their draw commands and delete the user from the system.

## Implementation

Your project comprises the development of two primary components: the server and the client. These components will work together to facilitate a collaborative drawing environment. Below are detailed tasks for each component, designed to guide your implementation process.

### Server Component

1. Server Setup and Network Communication

   (a) Implement server functionality to host the collaborative drawing session, ensuring it is accessible to clients across the network.

   (b) Establish robust TCP/IP connectivity between the server and clients, facilitating real-time data exchange.

   (c) Integrate a timeout mechanism to disconnect clients after a predefined period of inactivity (e.g., 10 minutes) to maintain system efficiency.

2. Command Handling and Canvas State Management

   (a) Develop a command processing module on the server to interpret and execute drawing commands received from clients.

   (b) Implement a dynamic canvas state management system capable of handling concurrent modifications by multiple clients while ensuring data consistency and integrity.

   (c) Design a mechanism to preserve the draw list of disconnected clients, adopting their commands if they do not reconnect within a specific timeframe.

3. Server Stability and Fault Tolerance

   (a) Enhance server resilience by implementing strategies to handle unexpected client disconnections without compromising the canvas state.

   (b) Assess and improve server performance under scenarios of high demand and concurrent access to identify and rectify potential bottlenecks.

**[30 marks]**

### Client Component

4. Graphical User Interface and Display
   (a) Create a user-friendly graphical interface to display the shared canvas, enabling users to view their contributions and those of others in real-time.
   (b) Incorporate features allowing users to filter and control the visibility of drawings on the canvas based on their preferences or needs.

5. Command Input and Execution
   (a) Implement a command input interface that enables users to easily select drawing tools, set colours, and execute drawing commands.
   (b) Ensure efficient communication between the client and server to process and reflect drawing commands on the shared canvas promptly.

6. Canvas Synchronisation and Updates
   (a) Develop a real-time canvas update system to ensure that changes made by any user are immediately visible to all participants.
   (b) Implement synchronisation mechanisms to maintain a consistent and up-to-date canvas view across all clients, reflecting concurrent edits accurately.

**[30 marks]**

### System Integration and Performance Optimisation

7. System Integration Testing
   (a) Conduct thorough integration testing to ensure seamless interaction between server and client components, verifying communication protocols and user interactions.
   (b) Perform comprehensive system testing to validate overall functionality, usability, and reliability.

8. Performance Evaluation and Optimisation
   (a) Evaluate the system's performance under various load conditions, including peak user concurrency and intensive drawing operations.
   (b) Identify and address any performance bottlenecks, optimising both server and client components for efficient resource use and responsive interaction.

**[10 marks]**

## Report and Video Presentation

Write a report to document your implementation and showcase it using a video presentation.

9. Write a report describing:

    - the design of your system, including any structure, behaviour and interaction diagrams which might help;
    - any implemented mechanisms which you think deserve special mention;
    - your approach to testing the system, listing test cases and results;
    - a list of bugs and issues of your implementation.

    Do **not** include any code listing in your report; only snippets are allowed.

10. Record a video presentation of not more than ten minutes where you showcase and discuss your implementation. Alongside the demonstration, you are free to show snippets of code, discuss shortcomings and features, bugs and accomplishments. Upload the video presentation to your Google Drive (remember to set access permissions to public) and link to it from your report.

**[30 marks]**

## Deliverables

Upload your report and source code, including any unit tests, test clients and additional utilities, through VLE by the specified deadline. Include makefiles with your submission which can compile your system. Make sure that your code is properly commented and easily readable. Be careful with naming conventions and visual formatting. Every system call output should be validated for errors and appropriate error messages should be reported.

*Note:* Ensure your implementation is compatible with Ubuntu 20.04.

## Considerations

Make sure you spend enough time designing your system before starting to code. Think of several scenarios which your game might encounter and design appropriate handling mechanisms. Aim for a system that is free of race conditions and deadlocks, and is still considerably efficient. The following is a non-exhaustive list of considerations you might want to ponder about:

- How is the canvas state represented? What data structures are required?
- Are data structure instances unique to each client or shared across multiple ones? Which approach would be more efficient? What overheads are required to share data structures?
- What happens when two clients attempt to concurrently modify the same data structure?
- How are operations from multiple clients sent to the server?
- Is the server thread-safe?
- Is asynchronous functionality required?
- Are client requests queued and serialised, or is some form of parallelism implemented in the server? What are the benefits of the latter option, if any?
- Is some type of conflict resolution required on the server?
- What happens when the connection between the server and client is lost?
- Are proper byte-ordering mechanisms implemented on both the client and server?
- What would be the impact on system efficiency if the clients cached the current map and only received changes in state from the server rather than the entire map at every update?
- What happens if the server crashes, or the clients drop out?

---