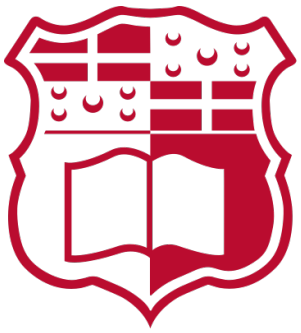

L-Università ta' Malta

Netsketch Assignment Documentation

CPS 2008
Operating Systems & Systems Programming II

Lenise Silvio
220199M

Contents

1 Introduction

This document discuss the implementation of the real-time collaborative whiteboard Netsketch for the CPS2008 course assignment.

The application is built using a client-server architecture, with the server being implemented in C++ and the client being implemented in python.

This document first discuss the system design and implementation details and then delves into how the application was tested. Lastly, the limitations of the systems are discussed.

The following link can be used to access the video presentation:
https://drive.google.com/file/d/1gnWuesciw7mvba5vLu0MSwF1zUQyzH-2/view?usp=sharing

2 System Design & Implementation

This section provides an overview of the system design. The main component details are described along with how they interact with each other.

2.1 Architecture Overview

The main components of the architecture are the following:

- **Server**: the server component manages the client connections, process draw commands and maintains a shared canvas state.
- **Client**: the client component represents the client application which allows the user to draw on the canvas and communicate with the server.
- **Shared Canvas state**: the shared canvas state represents the current state of the whiteboard which is maintained by the server to maintain a consistent and synchronized canvas across all clients.
- **Command Processor**: the command processor module is used on the server-side to interpret and execute drawing commands received by the clients. It updates the shared canvas state to ensure consistency among the clients.

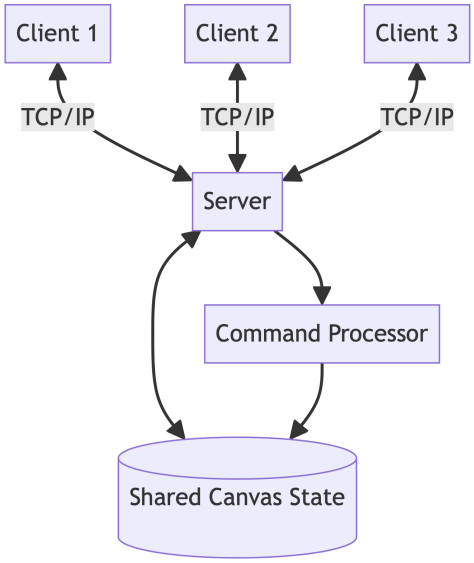

Figure 1: High-level overview of system architecture

The user interacts with the client application to generate draw commands. The client application sends to the draw commands to the server over a TCP/IP connection. The server then processes the commands using the commands module, which accordingly updated the shared canvas state. The server broadcasts changes all connected clients. The client application then updates the user's local graphical canvas based on the commands received from the server.

2.2 Server

The server is responsible for managing client connections, processing draw commands and maintaining the shared canvas state. It uses TCP/IP for reliable and ordered communication between the server and clients.

The server is implemented in C++ and uses multi-threading to handle multiple client connections concurrently.

The server-side has the following main components:

- Server Class
- Client Class
- Commands Class
- Canvas Class

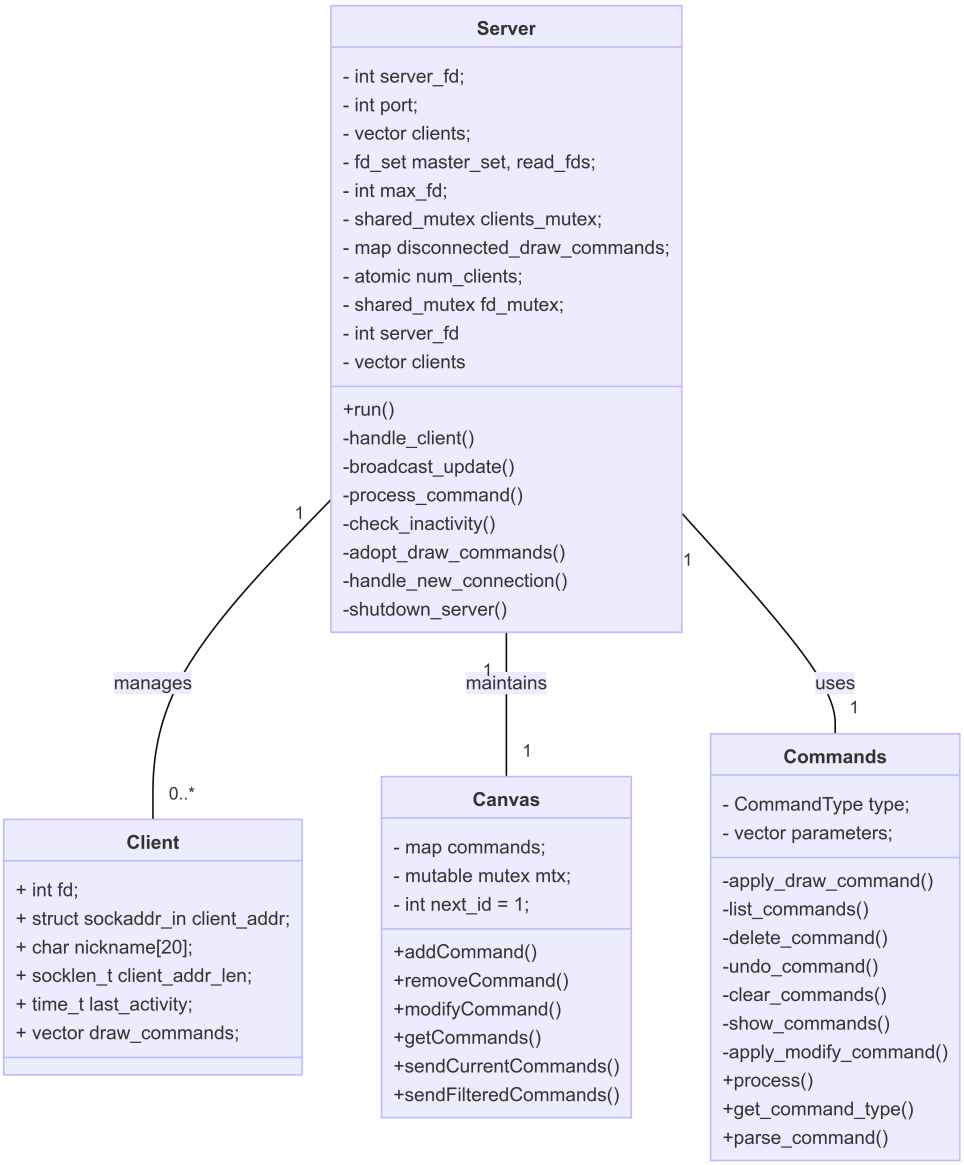

Figure 2: Overview of server-side architecture

These classes will be discussed in more detail next.

2.2.1 Server Class

The server class is the main class of the server. It initializes the server socket, accepts client connections and manages the overall functioning of the server.

To initialize a client connection, the server uses `select()` to be able

to monitor multiple client sockets simultaneously. The use of `select()` allows for non-blocking IO as multiple clients can be managed without dedicating a thread for each. When a new connection is detected, the `handleNewConnection()` function is called to handle the connection and create a new `Client` instance.

```
void Server::run() {
    // Initialize server
    // ...

    while (true) {
        // Wait for activity on sockets
        int activity = select(max_fd + 1, &read_fds, NULL, NULL, NULL);

        if (FD_ISSET(server_fd, &read_fds)) {
            handleNewConnection();
        }

        for (auto& client : clients) {
            if (client.fd != -1 && FD_ISSET(client.fd, &read_fds)) {
                if (!handle_client(client)) {
                    clients_to_remove.push_back(&client);
                }
            }
        }
    }
}
```

When a new command is received from a client application, the server class calls the `handleClientMessage()` function which uses the `Commands` class to handle the command and update the canvas shared canvas state.

```
void Server::handle_client(Client& client) {
    char buffer[1024];
    ssize_t bytes_received = recv(client.fd, buffer, sizeof(buffer), 0);

    if (bytes_received <= 0) {
        // Handle client disconnection
    } else {
        bool success = process_command(client, buffer, bytes_received,  client.fd);
        string response_message = success ? "Command processed successfully." : "Invalid command.";
        send(client.fd, response_message.c_str(), response_message.size(), 0);
        broadcast_update(client, buffer, bytes_received);
```

```
    }
}
```

The server maintains a representation of the canvas state which is then used to send the current state of the canvas each newly connected client.

Moreover, the server ensures thread safety. To do this, the server uses mutex locks to ensure safe access to shared resources.

Additionally, the server implements error handling to manage failure scenarios and log any errors that may occur. Function return values are checked and any errors are reported with appropriate cleanup being performed. For example, the server checks for inactive clients and removes them.

```
void Server::check_inactivity() {
    shared_lock<shared_mutex> lock(clients_mutex);
    time_t now = time(nullptr);
    for (auto& client : clients) {
        if (client.fd != -1 && difftime(now, client.
   last_activity) > INACTIVITY_TIMEOUT) {
            clients_to_remove.push_back(&client);
        }
    }

    // Handle disconnected clients
    // ...
}
```

2.2.2 Client Class

The client class represents a connected client and stores information about the client such as the client's file descriptor, their nickname and a timestamp of when their last command was issued.

2.2.3 Commands Class

The commands class parses received user commands, process them and applies any necessary changes to the shared canvas state.

2.2.4 Canvas Class

The canvas class represents the state of the shared canvas. It maintains a a list all all the issued draw commands and their attributes, including the id, tool, colour and the file descriptor of the user who issued the command.

2.3 Client

The client component is responsible for rendering the graphical canvas, handling user commands and communicating with the server. It is implemented using python with the graphical interface of the whiteboard being implemented using the Tkinter library.

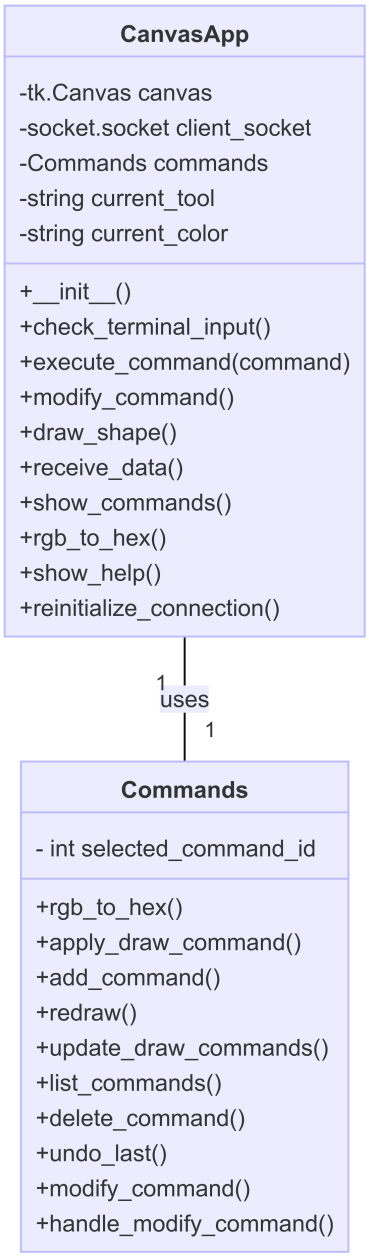

Figure 3: Overview of client architecture

The CanvasApp class and Commands class will be discussed in more detail next.

2.3.1 CanvasApp Class

The CanvasApp class initializes the graphical user interface and manages the canvas through Tkinter. Furthermore, it handles user input and communicates with the server by establishing a socket connection with the server. The main functions of the class will be discussed next.

The `__init__` function initializes the graphical canvas, initializes the network connection, and starts a thread which is used to receive messages from the server.

The function `execute_command()` is used to interpret user commands and appropriately process them. For example, if the user enter the command `tool rectangle`, the function will set the tool being used as `"rectangle"`.

The `draw_shape()` function is used to draw shapes and text on the graphical canvas and then sends the draw command to the server so that it can be broadcast to all other connected clients.

The `receive_data()` function runs in a separate thread and continuously listens for updates from the server. Once it receives a message from the server, it handles it and applies any changes to the canvas.

```
def receive_data(self):
        while True:
            try:
                message = self.client_socket.recv(1024).decode()
                commands = message.split('END\n')  # Split by delimiter
                for command in commands:
                    if command:
                        #print(f"Received command: {command}")
                        self.root.after(0, self.commands.apply_draw_command, self.canvas, command)
            except socket.timeout:
                continue
            except socket.error as e:
                print(f"Socket error: {e}")
                break
            except Exception as e:
```

```
16                 print(f"Unexpected error: {e}")
17                 break
```

Furthermore, the class keeps track of the ids of the user issued commands for filtering. Since filtering the commands does not affect the view of other users, a local cache of ids is kept to avoid having to request the ids from the server.

2.3.2 Commands Class

The commands class is responsible for executing and managing draw commands. It processes commands issued by the user and commands received from the server.

The function `apply_draw_command()` is responsible for interpreting and executing the different types commands and applying them to the canvas.

3 Testing

This section will detail the strategies employed to test the functioning of the system and to evaluate how reliable the application is.

To test the system, unit tests and integration were employed to check for the correctness of the application. These will be discussed next.

3.1 Unit Tests

Testing for the client involved unit testing. Unit tests were conducted for both the `CanvasApp` class and the `Commands` class.

For the `Commands` class, the test suite aims to cover the functionality of draw commands to ensure that the user can draw circle, rectangle, lines and text. The tests also check that commands that modify existing commands (i.e. delete, modify and clear) are executed correctly. Lastly, it also checks that list and filter commands perform correctly.

The `CanvasApp` class tests are designed to check that the commands are executed properly and to ensures that commands are communicated to the server.

3.2 Integration Tests

The integration tests were developed to assess the interaction between the server and the client. The main aim for the integration tests was to verify that the client connection is setup properly, ensure that the server processes user commands correctly and that the server manages an accurate state for the shared canvas.

The class `IntegrationTestSetup` was used in the integration tests suite to manage the client and server processes.

```
class IntegrationTestSetup:
    def __init__(self, server_path, client_path):
        self.server_path = server_path
        self.client_path = client_path
        self.server_process = None

    def start_server(self):
        self.server_process = subprocess.Popen([self.
    server_path])
        time.sleep(2)  # Allow server to initialize

    def stop_server(self):
        if self.server_process:
            self.server_process.terminate()
            self.server_process.wait(timeout=5)

    def run_client(self, commands, timeout=10):
        client_process = subprocess.Popen(['python3', self
    .client_path, '--test'],
                                          stdin=subprocess
    .PIPE,
                                          stdout=
    subprocess.PIPE,
                                          stderr=
    subprocess.PIPE,
                                          text=True)
        stdout, stderr = client_process.communicate(input=
    "\n".join(commands) + "\nexit\n", timeout=timeout)
        return stdout, stderr
```

Due to the fact that some tests were failing as a result of race conditions, delay and retry mechanisms were added to the tests.

3.3 Evaluation

All tests in the unit test suit and the integration test suite successfully pass.

When measuring the coverage for the unit tests, the following results were obtained.

```
Name            Stmts   Miss  Cover   Missing
-----------------------------------------------
canvas_app.py     198    108    45%   40-43, 48, 56-57,
   62-63, 65-66, 71, 78-79, 81, 90-91, 93, 101-102,
   114-115, 117-127, 130-147, 151-152, 155-187, 195-209,
   212-220, 223-237, 240-252
commands.py       163     62    62%   15-22, 24-26, 42-44,
    46-47, 62, 83-88, 91-93, 96-109, 112-117, 142, 146,
   152-153, 157-159, 168, 180-181, 188-194
unit_tests.py     127      3    98%   134-135, 175
-----------------------------------------------
TOTAL             488    173    65%
```

The overall coverage for the unit tests were 65%, suggesting moderate coverage.

The coverage for the integration tests was as follows.

```
Name                    Stmts   Miss  Cover   Missing
-----------------------------------------------------
canvas_app.py             198    181     9%   10-37, 40-43,
    46-127, 130-147, 151-152, 155-187, 190-209, 212-220,
   223-237, 240-252
commands.py               163    151     7%   3-7, 10,
   13-88, 91-93, 96-109, 112-117, 120-129, 133-135, 142,
   145-148, 151-197
integration_tests.py      108     21    81%   28-31, 37-39,
    47-49, 56-58, 68-71, 75-77, 172
-----------------------------------------------------
TOTAL                     469    353    25%
```

The overall coverage was 25%, suggesting that testing efforts should be increased to ensure that the system is reliable. More tests should have been added when testing for multi-user scenarios. Stress tests should be added to simulate a large number of concurrent users and operations.

4 Bugs and Limitations

As mentioned in the previous section, more tests could be conducted to ensure reliable and correct performance. While the current implementation was tested with a small number of concurrent users, it was not tested with large amounts of users and hence more testing also should

be added in that regard.

For this assignment, the application does not support user-inputted nicknames. An attempt was made as follows to include nicknames but was removed from the final implementation as it had bugs and they were interfering with the functionality of the canvas.

`server.cpp`

```
\\ Other functions
\\ ...

void Server::wait_for_nickname(Client& client) {
        char buffer[1024];
        while (true) {
            ssize_t bytes_received = recv(client.fd, buffer, sizeof(buffer), 0);
            if (bytes_received <= 0) {
                if (bytes_received == 0) {
                    cout << "Client disconnected before setting nickname" << endl;
                } else {
                    cerr << "Error receiving nickname from client: " << strerror(errno) << endl;
                }
                remove_client(client);
                return;
            }

            std::string message(buffer, bytes_received);
            std::istringstream iss(message);
            std::string command;
            std::string nickname;
            iss >> command >> nickname;

            if (command == "NICKNAME") {
                bool nickname_taken = false;
                {
                    shared_lock<shared_mutex> lock(clients_mutex);
                    for (const auto& other_client : clients) {
                        if (&other_client != &client && strcmp(other_client.nickname, nickname.c_str()) == 0) {
                            nickname_taken = true;
                            break;
                        }
                    }
                }

```

```
36                 if (nickname_taken) {
37                     send(client.fd, "NICKNAME_TAKEN", 14,
   0);
38                 } else {
39                     strncpy(client.nickname, nickname.
   c_str(), sizeof(client.nickname) - 1);
40                     client.nickname[sizeof(client.nickname
   ) - 1] = '\0';  // Ensure null-termination
41                     send(client.fd, "NICKNAME_ACCEPTED",
   17, 0);
42                     cout << "Client " << client.fd << "
   set nickname to: " << client.nickname << endl;
43                     return;  // Exit the function once the
    nickname is set
44                 }
45             } else {
46                 // If the first command is not NICKNAME,
   send an error message
47                 send(client.fd, "ERROR_SET_NICKNAME_FIRST"
   , 24, 0);
48             }
49         }
50     }
51
52     bool Server::process_command(Client& client, const
   char* buffer, ssize_t bytes_received, int client_fd) {
53     std::string message(buffer, bytes_received);
54     std::istringstream iss(message);
55     std::string command;
56     iss >> command;
57
58     if (command == "NICKNAME") {
59         std::string nickname;
60         iss >> nickname;
61         bool nickname_taken = false;
62         {
63             shared_lock<shared_mutex> lock(clients_mutex);
64             for (const auto& other_client : clients) {
65                 if (&other_client != &client && strcmp(
   other_client.nickname, nickname.c_str()) == 0) {
66                     nickname_taken = true;
67                     break;
68                 }
69             }
70         }
71
72         if (nickname_taken) {
73             send(client.fd, "NICKNAME_TAKEN", 14, 0);
74         } else {
75             strncpy(client.nickname, nickname.c_str(),
   sizeof(client.nickname) - 1);
```

```
            client.nickname[sizeof(client.nickname) - 1] = '\0';  // Ensure null-termination
            send(client.fd, "NICKNAME_ACCEPTED", 17, 0);
            cout << "Client " << client.fd << " set nickname to: " << client.nickname << endl;
        }
        return true;
    }

    // Process other commands as before
    Commands processor;
    return processor.process(client, buffer, bytes_received, client_fd);

    void Server::handle_new_connection() {
        struct sockaddr_in client_addr;
        socklen_t client_addr_len = sizeof(client_addr);
        int new_socket = accept(server_fd, (struct sockaddr *)&client_addr, &client_addr_len);

        if (new_socket < 0) {
            cerr << "Error accepting connection: " << strerror(errno) << endl;
            return;
        }

        // Keep the socket in blocking mode for initial handshake
        fcntl(new_socket, F_SETFL, fcntl(new_socket, F_GETFL, 0) & ~O_NONBLOCK);

        {
            unique_lock<shared_mutex> lock(clients_mutex);
            clients.emplace_back(new_socket, client_addr, client_addr_len, "");
            snprintf(clients.back().nickname, sizeof(clients.back().nickname), "client_%d", new_socket);
            cout << "New connection from client " << clients.back().nickname << endl;
        }

        // Handle nickname setting in a separate thread
        thread nickname_thread(&Server::handle_nickname, this, ref(clients.back()));
        nickname_thread.detach();
    }
}
```

canvasApp.py

```
class CanvasApp:
    def __init__(self, root):
```

```
        self.root = root
        self.user_commands = set()
        self.shape_id_counter = 0
        self.commands = Commands()
        self.current_tool = None
        self.current_color = "black"
        self.selected_id = None

        self.show_var = tk.StringVar()

        # Create canvas
        self.canvas = tk.Canvas(root, width=800, height
    =600, bg="white")
        self.canvas.pack()

        # Setup server connection
        self.client_socket = socket.socket(socket.AF_INET,
     socket.SOCK_STREAM)
        self.client_socket.connect(('127.0.0.1', 6001))

        # Set nickname
        self.set_nickname()

        # Start receiving thread
        self.commands = Commands()
        self.receive_thread = threading.Thread(target=self
    .receive_data, daemon=True)
        self.receive_thread.start()

    def set_nickname(self):
        while True:
            nickname = input("Enter your nickname: ").
    strip()
            command = f"NICKNAME {nickname}"
            try:
                self.client_socket.sendall(command.encode
    ())
                response = self.client_socket.recv(1024).
    decode().strip()
                if response == "NICKNAME_ACCEPTED":
                    print(f"Nickname '{nickname}' accepted
     by the server.")
                    return True
                elif response == "NICKNAME_TAKEN":
                    print(f"Nickname '{nickname}' is
    already taken. Please choose another.")
                else:
                    print(f"Unexpected response from
    server: {response}")
            except socket.error as e:
                print(f"Socket error: {e}")
```

```
                print("Reconnecting...")
                self.reinitialize_connection()

\\ Rest of functions
\\ ...
```

Another issue with the implementation is that it does not support undo commands. One way this could be implemented would be to keep a command history on the server in addition to the the current canvas state so that in case of an undo command, the commands history can be referenced.

In addition to this, a bug occurs with client disconnections. While there is a mechanism to attempt re-connection, this does not work reliably/ This leads to temporary disconnections causing synchronization issues.

Moreover, another improvement could be to serialize the commands between user and server and add persistent storage. This is important as proper serialization and de-serialization methods ensure consistency across different architectures.

One way to optimise the application could be when two clients attempt to modify the same shape concurrently. The server currently processes commands in the order they are received. While this approach is simple, a more sophisticated conflict resolution mechanism could be implemented. One way to deal with this could be to send time stamps along with the commands to ensure that commands are processed in the order they were received.

Further improvement could also be done with regards to the server sending updates to the clients. Currently, the server sends full updates to clients. However, implementing a system where clients cache the current state and only receive delta updates from the server could significantly improve efficiency. Command batching could also be implemented to reduce network overhead.

Lastly, logging and checking for errors could be improved. For example after issuing a draw command, the client could check that the server processed the command correctly or if an error occurs. This would ensure that the canvas state is consistent in case of an error.

5 Conclusion

Overall, the the application developed manages to implement a basic real-time collaborative whiteboard. While the application still has some bugs, the majority of tools are supported and function correctly.