SimpleChat Documentation

CS 550 Advanced OS - Programming Assignment - Part 1
Harun Pekacar - A20607262
Github Link: https://github.com/0hr/SimpleChat

Introduction

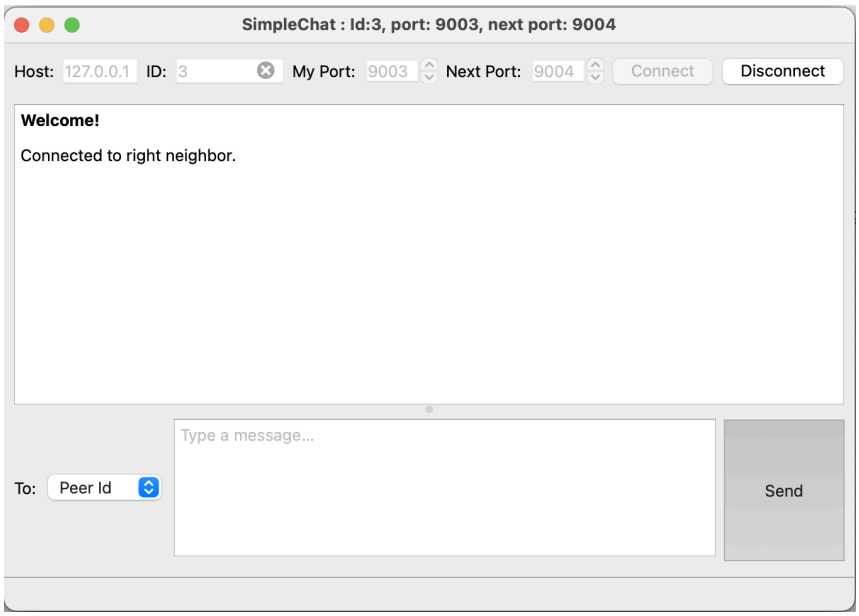
SimpleChat is a peer-to-peer (P2P) messaging application that demonstrates a ring network. It's built in C++ and the Qt6 framework. This document provides an overview of the application's design, features, and building, implementation, and basic test cases.

Core Requirements

GUI Implementation

The application provides a basic graphical user interface (GUI) for users to interact with the chat system. The GUI is implemented using the Qt6 framework. GUI has the following requirements;

- A display area shows the history of the chat, including messages sent and received.
- A multi-line text input field allows users to compose and send messages.
- The chat log area supports word wrap to handle long messages.
- The text input area is automatically focused when the application is launched.



Network Communication

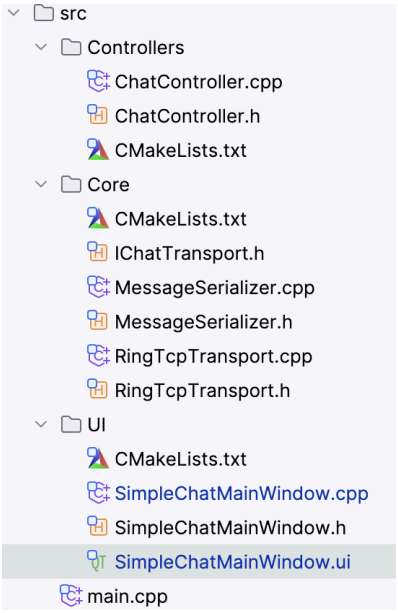
The chat system uses a TCP-based messaging protocol for reliable communication between peers.

- QTcpSocket is used for network communication, ensuring reliable, stream-based data transfer.
- Messages are serialized into a binary format for network transmission and deserialized. This is handled by the MessageSerializer class, which uses QVariantMap to represent messages before serialization.
- Each message is a QVariantMap containing the following fields
    - **ChatText**: The actual message content. Its type is QString.
    - **Origin**: A unique identifier for the SimpleChat instance that originally sent the message. Its type is QString
    - **Destination**: The unique identifier of the recipient SimpleChat instance. Its type is QString
    - **Sequence**: A sequence number for ordering messages from a specific origin. Its type is qulonglong.
- The chat instances are connected in a ring. Each instance maintains a connection to the "next" peer in the ring. When a message is sent, it is passed along the ring from one peer to the next until it reaches its intended destination.

Project Design

The application is divided into three distinct layers: the User Interface (UI), the Controller, and the Core. This separation makes the codebase easier to understand, maintain, and extend. The UI is only responsible for displaying data and capturing user input, the Controller for the application logic, and the Core for low-level networking and data serialization. [1]

The application is built on Qt's signals and slots mechanism, which allows for a highly decoupled and event-driven architecture. Components communicate with each other by emitting signals, which are then handled by slots in other components.



Code Structure and Components

The UI Layer

The UI layer is implemented in the SimpleChatMainWindow class. Its responsibility is to present the user interface and relay user actions to the controller.
SimpleChatMainWindow.ui is a Qt6 UI file that defines the layout and widgets of the main window. It contains, for example, the chat log display (QTextBrowser), the message input field (QLineEdit), and the send button (QPushButton)
SimpleChatMainWindow.cpp file is a class that connects the UI elements to the application logic. It receives a pointer to the ChatController in its constructor and uses it to send messages. It also provides public slots that the controller can use to display messages in the chat log.

The Controller Layer

The ChatController mediates between the UI and the Core networking layer and implements the core logic of the chat protocol.

The Core Layer

The Core layer provides the low-level networking and serialization functionality.

RingTcpTransport: This class implements the IChatTransport interface and is responsible for managing the TCP connections in the ring network.
It uses a QTcpServer to listen for incoming connections from the previous peer in the ring and a QTcpSocket to connect to the next peer. A QTimer is used to periodically attempt to reconnect to the next peer if the connection is lost.

MessageSerializer: This is a utility class for serializing and unserializing messages. The serialize static method takes a QVariantMap and serializes it into a QByteArray. The unserialize static method deserializes a message from a QByteArray buffer to a QVariantMap.

The Ring Network

In a ring network, each chat application links to a neighbor node, and data is passed from node to node around the ring.

Establishment of the Ring

In SimpleChat, the ring is established at startup based on command-line arguments:

- **--port:** Specifies the port on which the current instance will listen for incoming connections.
- **--next:** Specifies the port of the next peer in the ring to which the current instance will connect.
- **--peers:** Comma-separated list of other peers.

Each instance acts as both a server and a client. Also, the ring network can be established on the GUI.

Message Propagation

When a user sends a message,

1. The ChatController creates a message with the Origin, Destination, ChatText, and Seq fields.
2. The message is handed to the RingTcpTransport.
3. The transport serializes the message and sends it to the next peer in the ring.
4. When the next peer receives the message, its ChatController inspects the Destination field.
    - If the destination matches the current peer's ID, the message is delivered to the UI.
    - If the destination does not match, the message is forwarded to the *next* peer in the ring.
5. This process repeats until the message reaches its destination.

Advantages

- There is no central server. So the network is decentralized.
- Data flows in one direction, simple and easy to implement.

Disadvantages

- A message may have traveled through many nodes to reach its destination. Thus, it increases latency.
- If one node goes down, the entire ring is broken, and messages can no longer be propagated.

Build and Run the Project

A helper script, build_run.sh, is provided to build and run 4 nodes. They will be connected in a ring with IDs 1, 2, 3, and 4 on ports 9001, 9002, 9003, and 9004.

1. **Make the script executable:** chmod +x build_run.sh
2. **Run the script:** ./build_run.sh

Build Manually

1. **Create build directory:** mkdir build
2. **Go into "build" directory:** cd build
3. **Generate build files:** cmake ..
4. **Compile the project:** cmake --build .

Run the project

To run an instance of the application, you need to specify the user ID, the port, and the port of the next peer in the ring, and other peers.

./build/SimpleChat --id <id>
--port <port>
--next <peer-port>
--peers <comma-separated-list-of-other-peer-ids>

For example, to start a chat client with ID "1" on port 9001, which connects to a peer on port 9002, you would run:

./build/SimpleChat --id 1 --port 9001 --next 9002 --peers 2,3,4

To create a ring of 4 peers, you would run the following commands in separate terminals:

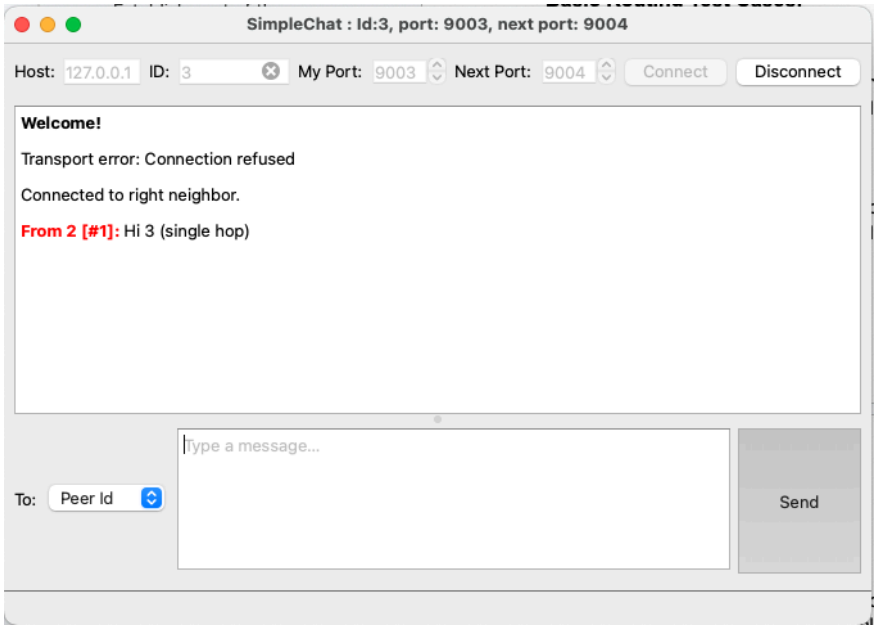
./build/SimpleChat --id 1 --port 9001 --next 9002 --peers 2,3,4

./build/SimpleChat --id 2 --port 9002 --next 9003 --peers 1,3,4
./build/SimpleChat --id 3 --port 9003 --next 9004 --peers 1,2,4
./build/SimpleChat --id 4 --port 9004 --next 9001 --peers 1,2,3

Basic Test Cases

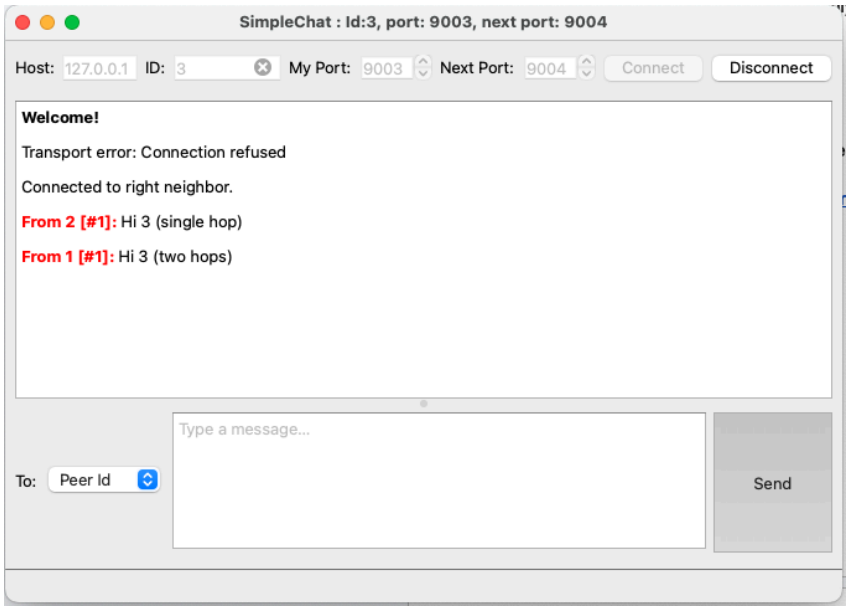
Basic Routing Test Cases:

Test Case 1
Test Case: Peer 2 sends "Hi 3 (single hop)" to Peer 3
Expect: Message Shows on Peer 3 only. Path 2 to 3



Test Case 2

Test Case: Peer 1 sends "Hi 3 (two hops)" to Peer 3
Expect: Message Shows on Peer 3 only. Path 1 to 2 to 3

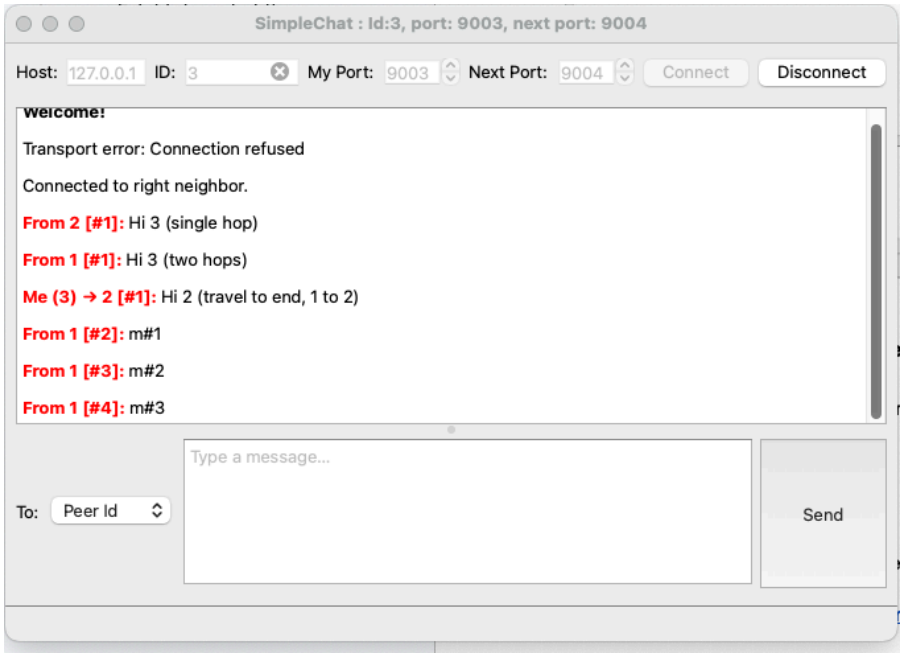


Test Case 3

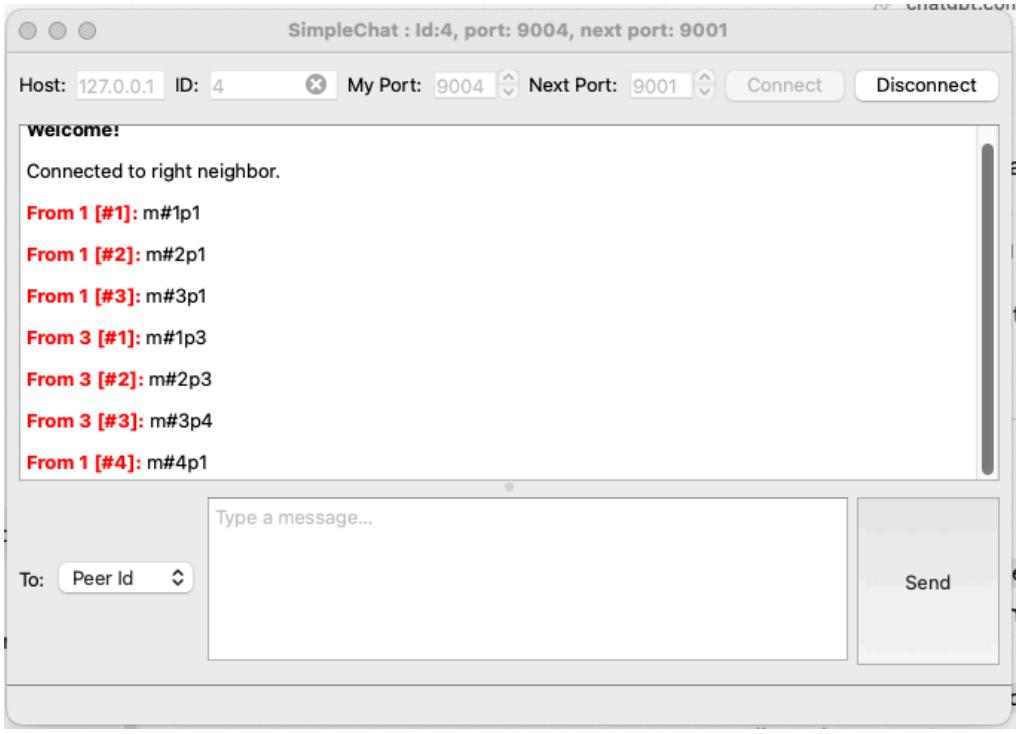
Test Case: Peer 3 sends "Hi 2 (travel to end, 1 to 2)" to Peer 2
Expect: Message Shows on Peer 3 only. Path 3 to 4 to 1 to 2

Ordering & Sequencing Test Cases

Test case: Peer 1 sends messages to Peer 3 quickly m#1, m#2, m#3
Expect: On Peer 3, messages appear m#1, m#2, m#3 in order from Peer 1.

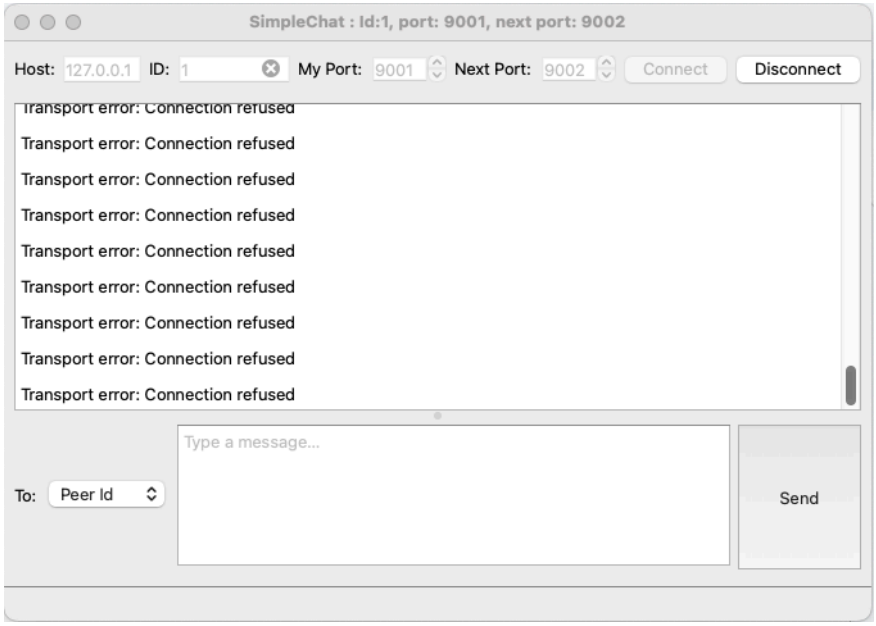


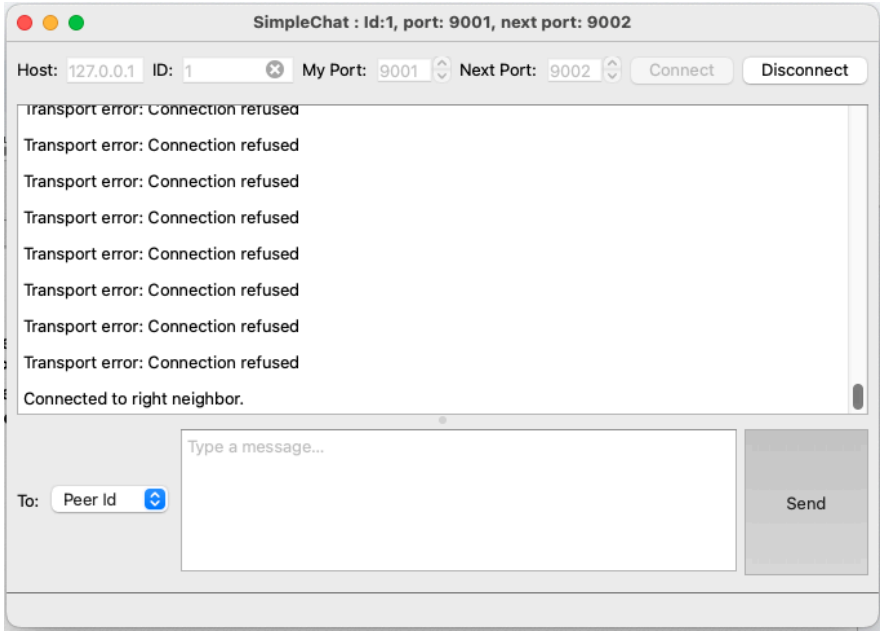
Test case: Peer 1 quickly sends to Peer 4: m#1p1, m#2p1, m#3p1; then Peer 3 sends to Peer 4: m#1p3, m#2p3, m#3p3; finally, Peer 1 sends to Peer 4: m#4p1.
Expect: On Peer 4, messages appear m#1p1(sequence: 1), m#2p1(sequence: 2), m#3p1(sequence: 3), m#1p3(sequence: 1), m#2p3(sequence: 2), m#3p3(sequence: 3), m#4p1((sequence: 4) in order.

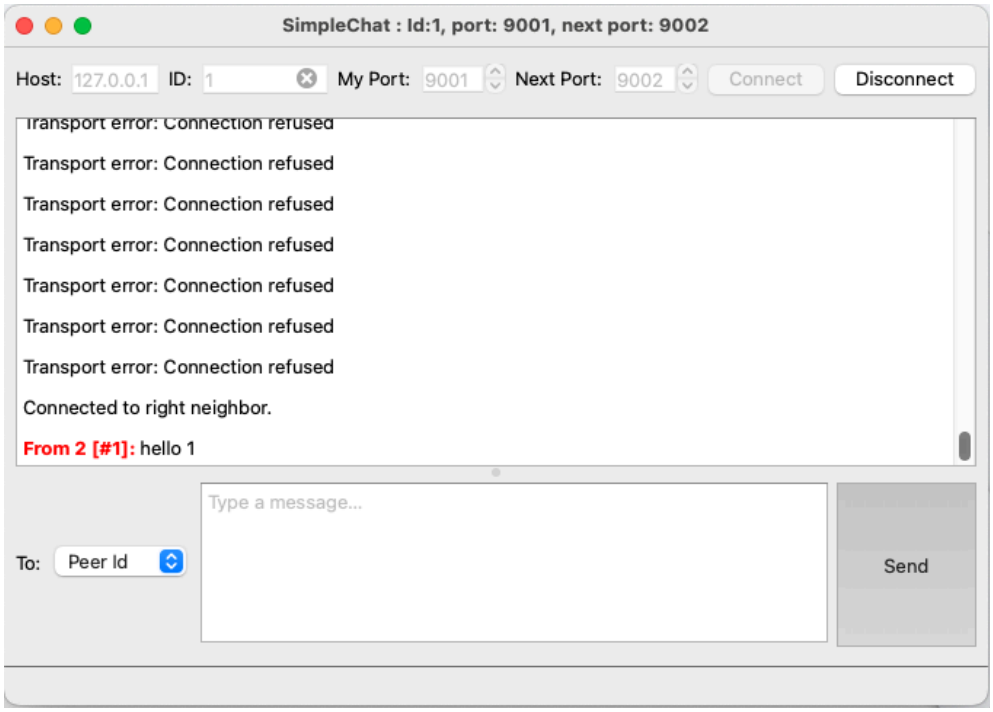


Fault Test Case

Test Case: Peer 2 disconnects from the ring
Expect: Peer 1 shows message,
Test Case: Peer 2 connect to the ring again
Expect: Peer 2 connect to neighbor
Test Case: Peer 2 sends "hello 1" to peer 1
Expect: Message Shows on Peer 1.







References:

[1] K. Pokharel, “How to structure projects in C++ or Qt: Cmake and directories,” DEV Community, https://dev.to/kastuv/how-to-structure-projects-in-c-or-qt-cmake-and-directories-5bbm