Plex Share

Networking Module - Design Specifications

CS5617: Software Engineering
September 2022

Mohammad Umar Sultan
111901031

Table of Contents

Team

111901016 Anish Bhagavatula (Team Lead)

111901031 Mohammad Umar Sultan (Team Member)

Overview

The networking module is responsible for facilitating communication between clients and the server. It ensures that each module can send data to the server and to every other client in the room, and that other modules can send and receive data without starving by using two priority queues, one for high priority data and other for low priority data. It makes sure that on the receiving end, any module that has subscribed for notifications only receives the data that was sent by the same module. This module does not depend on any other module, while other modules which need to send data over the network depend on this module, like content, screen-share and whiteboard modules.

This module is implemented with the assumption that the application will run over a LAN.

Design Patterns

Factory Pattern

In the Factory Pattern, we create an object without exposing the creation logic to the user and provide access to the newly created object using a common interface. By using Factory pattern in the networking module we can abstract the communication medium from other modules.

The networking module will provide a `GetCommunicator` factory method which initializes the communicator object and returns an object of type `ICommunicator`.

Singleton Pattern

The Singleton pattern involves a single class which is responsible for creating an object while making sure that only a single object gets created. This class provides a way to access its only object which can be accessed directly without having to instantiate the object of the class.

In the networking module, as there is only a single socket to send the data, we will use the Singleton pattern so that only one instance of the `ICommunicator` object is created.

Publisher-Subscriber(Pub-Sub) Pattern

In the Publisher-Subscriber pattern, publishers are senders of messages and the subscribers are the receivers of the messages. But the publishers do not program the messages to be sent directly to the subscribers, but instead they categorize the published messages into classes, without knowledge of subscribers. Similarly, subscribers express interest in one or more classes of messages and only receive messages that are of interest to them, without knowledge of publishers.

The networking module will act as a publisher for the messages received from the network. The networking module will provide the `INotificationHandler` interface which the other modules who want to receive messages need to implement and subscribe to the networking module by calling the `Subscribe` method of the `Communicator` object. When the networking module receives a message, it will call the `OnDataReceived` method of the handler provided by the subscribed module.

Submodules

Sockets

The Sockets submodule on the server side finds a free port and starts listening on it. On the client side, It takes data from the sending queue and sends it to the server.

It listens to the sending queue on the sender's end and sends the data from the queue to the receiver. And on the receiver’s end it pushes the received data to the received queue.

As queues are managed by the Queues submodule, this submodule depends on the Queues submodule.

Communication

The communication submodule is responsible for implementing the communicator in a singleton factory design pattern and encapsulating the internal structure of the networking module.

The networking module is used by other modules for communication, they can use this submodule to interact with the networking module.

Other modules receive a communicator object from the communication submodule, which serves as the main channel for sending and receiving messages over the network. It is responsible for sending the data for other submodules, and when a message is received, it calls the hadler of the respective submodule.

Serialization and Queues

The specs for these submodules are written by Anish.

Sockets

About

A socket is a software component within a computer network node that acts as an endpoint for sending and receiving data. Sockets are generally used in client server architecture. The server creates a socket, attaches it to a network port address and then starts listening. The client creates a socket and then connects to the server socket and then it can communicate with the server.

Responsibility

The Sockets submodule on the server side finds a free port and starts listening on it. On the client side, it takes data from the sending queue and sends it to the server.

It listens to the sending queue on the sender's end and sends the data from the queue to the receiver. And on the receiver’s end it pushes the received data to the received queue.

Socket protocol

We have 2 choices for the socket protocol:

1. TCP protocol
2. UDP protocol

TCP protocol

TCP stands for Transmission Control Protocol. It is a connection-oriented network communication protocol. Connection-oriented means that the client and server should establish a connection before the communication and should close the connection after the communication. TCP is a reliable protocol, which means that it guarantees the delivery of data packets. The disadvantage of TCP is that it is slower than the UDP.

UDP protocol

UDP stands for User Datagram Protocol. The UDP protocol allows the client and server to send the messages in the form of datagrams over the network. UDP is a connectionless protocol and it does not provide any guarantee for the delivery of the packet. So UDP is an unreliable protocol but the advantage of UDP is its fast transmission compared to TCP.

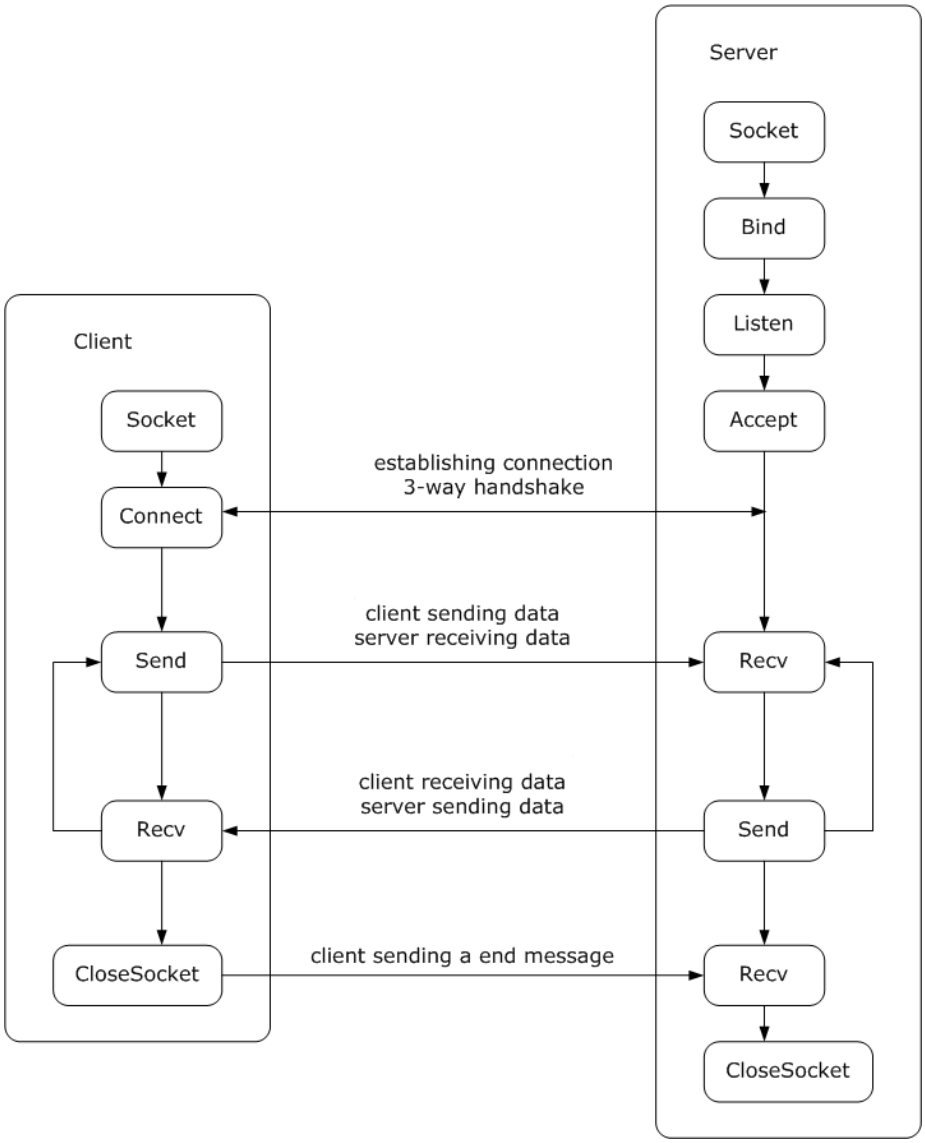
Choice: TCP protocol

As TCP guarantees the delivery of packets, we will use TCP for file sharing, chat, and whiteboard. For screen sharing UDP would be better, but as we are already using TCP for other modules, we will use TCP for screen sharing also. Later if we get more time we will add support for UDP for screen sharing.

Fragmentation

Only 65,535 bytes can be sent in a single packet using TCP. Therefore, if the amount of data to be sent exceeds this size, we split the message into parts that can be transmitted through TCP over the network.

TCP Socket flow diagram



Threads

SendQueueListener

This thread continuously listens to the sending queue, whenever there is some data pushed into the queue, it takes the data from the queue and sends it over the network.

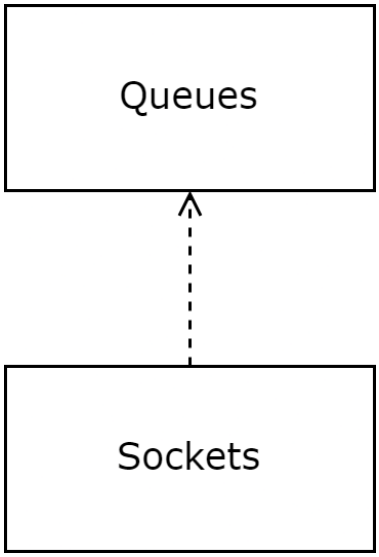
SendQueueListener
- queue: IQueue
+ SendQueueListener(IQueue queue) + Start(): void + Stop(): void

SocketListener

This thread continuously listens to the incoming data from the network and when some data is received it pushes the data into the received queue.

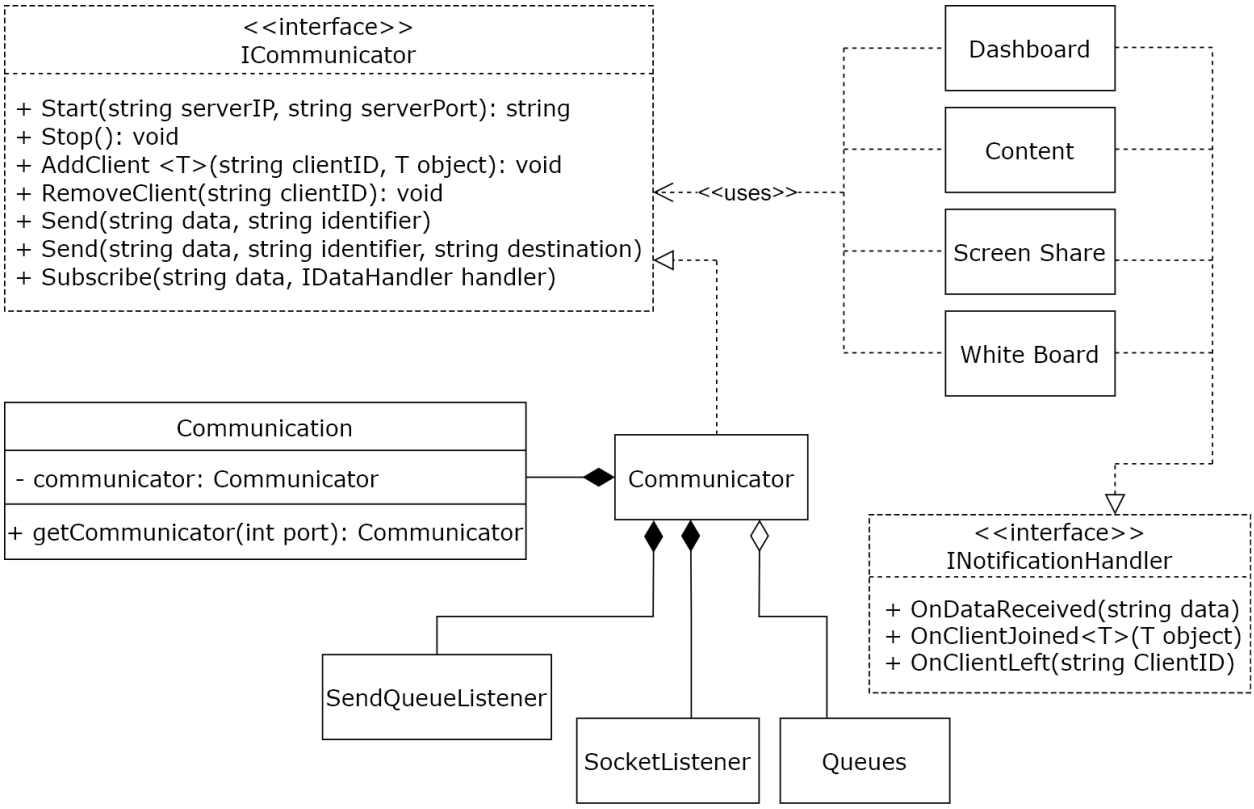
SocketListener
- port: int - queue: IQueue - serverSocket: ServerSocket
+ SocketListener(int port, ServerSocket serverSocket, IQueue queue) + Start(): void + Stop(): void - PushToQueue(string data, string moduleIdentifier): void

Dependency Diagram



The Sockets submodule depends on the Queues submodule as the `SendQueueListener` thread continuously listens to the sending queue, whenever there is some data pushed into the queue, it takes the data from the queue and sends it over the network.

Class Diagram



Program Flow

Client Side

The `SendQueueListener` thread is continuously listening to the sending queue. And whenever data is pushed into the sending queue, this thread takes the data from the queue and sends it to the server.

The `SocketListener` thread is continuously listening to the incoming data from the network and when some data is received it pushes the data into the received queue.

Server Side

The Sockets submodule on the server side finds a free port and starts listening on it.

The `SocketListener` thread is continuously listening to the incoming data from the network and when some data is received it pushes the data into the received queue.

The `SendQueueListener` thread is continuously listening to the sending queue. And whenever data is pushed into the sending queue, this thread takes the data from the queue and sends it to all the clients.

Communication

About

The communication submodule is responsible for implementing the communicator in a singleton factory design pattern and encapsulating the internal structure of the networking module.

The networking module is used by other modules for communication, they can use this submodule to interact with the networking module.

Other modules receive a communicator object from the communication submodule, which serves as the main channel for sending and receiving messages over the network. It is responsible for sending the data for other submodules, and when a message is received, it calls the hadler of the respective submodule.

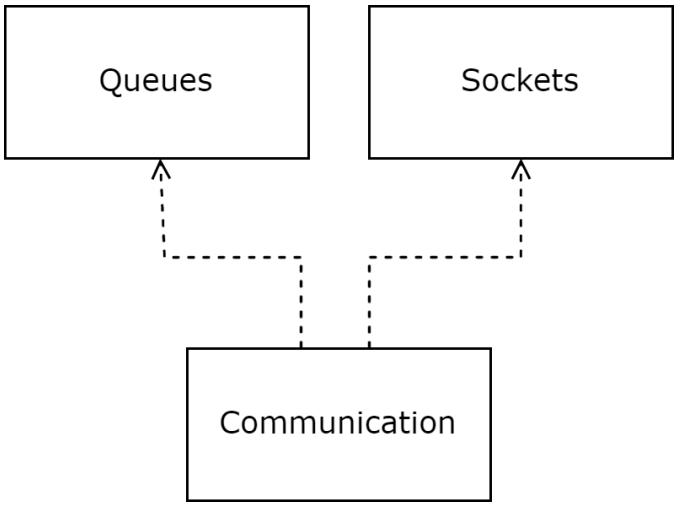
Design Pattern: Singleton Factory Pattern

By using Factory pattern in the networking module we can abstract the communication medium from other modules. The networking module will provide a GetCommunicator factory method which initializes the communicator object and returns an object of type ICommunicator.

In the networking module, as there is only a single socket to send the data, we will use the Singleton pattern so that only one instance of the ICommunicator object is created.

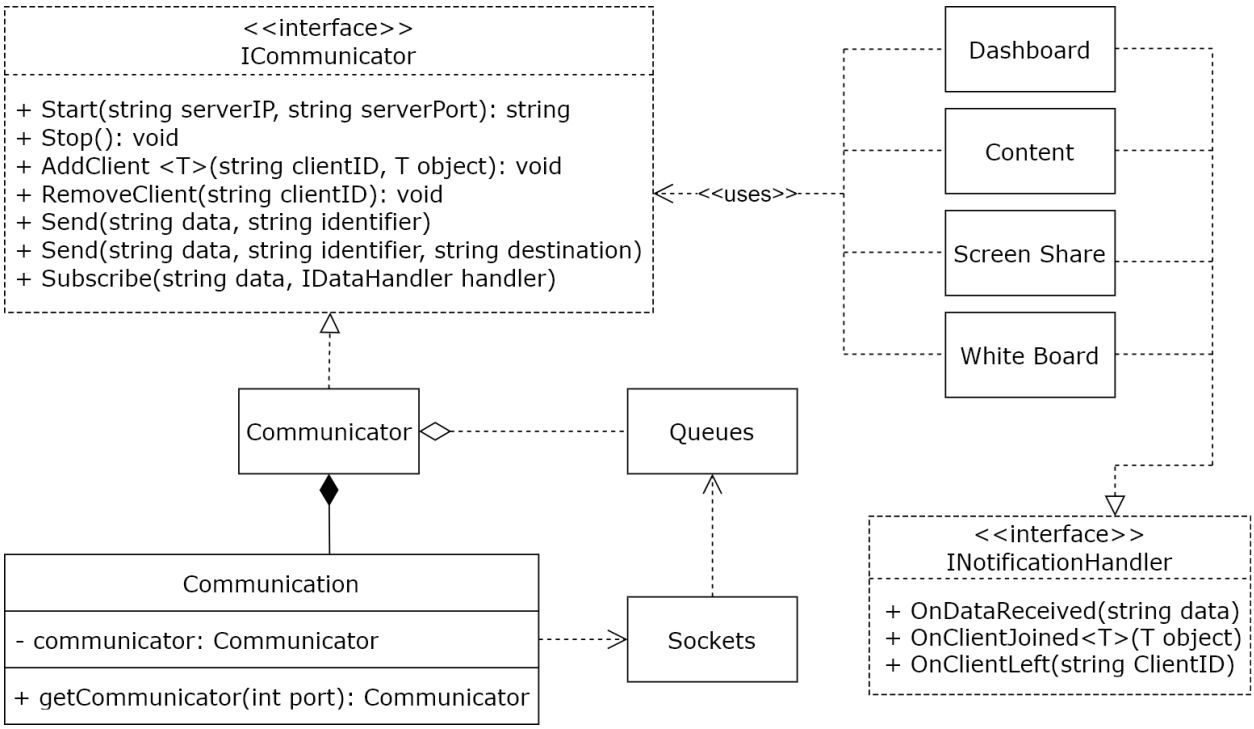
The communication submodule is responsible for implementing the communicator in a singleton factory design pattern.

Dependency Diagram

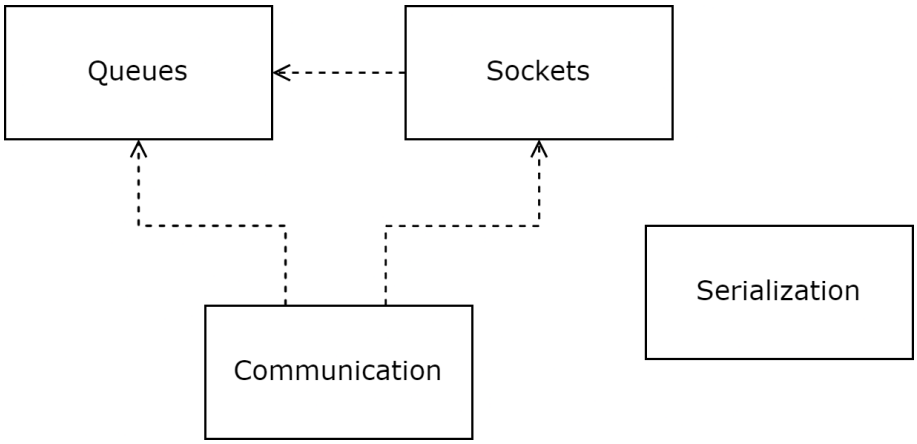


The Communication submodule depends on the Queues submodule because to send the data the communicator pushes the data into the sending queue. And it also depends on the Sockets submodule to send the data using sockets.

Class Diagram



Final Dependency Diagram



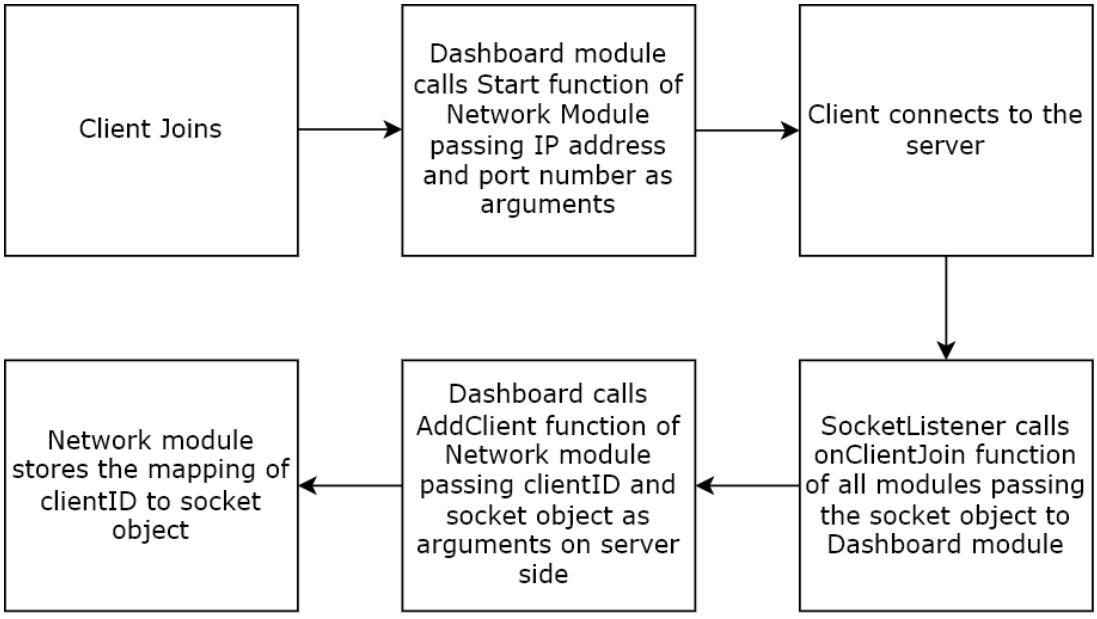
The Sockets submodule depends on the Queues submodule as the `SendQueueListener` thread continuously listens to the sending queue, whenever there is some data pushed into the queue, it takes the data from the queue and sends it over the network.

The Communication submodule depends on the Queues submodule because to send the data the communicator pushes the data into the sending queue. And it also depends on the Sockets submodule to send the data using sockets.

The Serialization and the Queues submodule does not depend on any other module.

Client joining and leaving

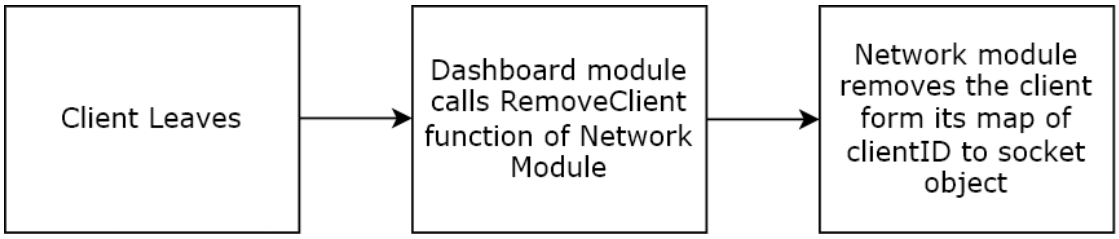
On client join



When a client joins, the dashboard module should call the Start function of the networking module passing the IP address and port number of the server as arguments. Then the networking module, on the client side, tries to connect to the server and once the connection is established, data can be sent between the client and server.

For the server to be able to communicate to a unique client, each client's socket needs to be identified by the server. For this when the client connects to the socket, the Networking module calls the OnClientJoined handler of all the modules, so it will pass the socket object as an argument to the Dashboard module. The Dashboard module gets this object and then it calls the AddClient function of the Communicator object passing ClientID and the socket object as arguments. Now the Networking module can store this as a mapping of ClientID to socket objects. This way the server can identify clients uniquely to send messages to a unique client.

On client leave



When a client leaves the room, the dashboard module should call the RemoveClient function of the Network module. Then the Network module will remove this client from its map of ClientID to the socket object.