Meet.me

Networking Module – Design Specification

Team
1. Subhash S – Team Lead
2. Abdullah Khan - Serialization
3. Alisetti Sai Vamsi – Queue Management
4. Tausif Iqbal – Socket Management

Table of Contents

Team ... 3
About ... 3
Design Choices ... 3
- Client-Server vs Peer-to-Peer ... 3
- TCP vs UDP sockets for Screen-sharing ... 5
- Design Patterns ... 5

Roles ... 6
- Queue Management – Alisetti Sai Vamsi ... 6
- Sockets – Tausif Iqbal ... 6
- Serialization – Abdullah Khan ... 6
- Communication Manager – Abdullah Khan ... 6

Serialization ... 7
- Overview ... 7
- Design Choices ... 7
- Activity Diagram ... 7
- Class Diagram ... 8

Queue Management ... 9
- Overview ... 9
- Background ... 9
- Problem ... 9
- Design Choices ... 10
- Module Specification ... 11
- Sender Queue ... 11
- Receiver Queue ... 12
- Activity Diagram ... 12
- Dependency Diagram ... 13

Sockets ... 14
- Overview ... 14
- Responsibility ... 14
- About Sockets ... 14
- Design Choices ... 15
- Socket Listener ... 15
- Send Queue Listener ... 15
- Class Diagram ... 16
- Program Flow ... 16

Communication Manager ... 18
- Overview ... 18
- Design Choices ... 18
- Class Diagram ... 18

Dependency Diagram ... 19
Activity diagrams ... 20
Interfacing with other modules ... 23
- Sending an object ... 23
- Client Joins the room ... 24

Team

Subhash S	Team Lead
Alisetti Sai Vamsi	Queue Management
Abdullah Khan	Serialization
Tausif Iqbal	Sockets

About

This module is responsible for the communication between clients. It ensures that each module can send objects to all the other clients in the room and that the modules receive data without starvation by using multi-level priority queues to give higher priority to real-time objects, at the same time not ignoring chat and file objects. On the receiving end, networking module ensures that every other module subscribed for notifications receives only the object which was sent it by itself. This module is at the bottom of the overall dependency diagram and is not dependent on any other modules in the application.

Design Choices

Client-Server vs Peer-to-Peer

Client-Server architecture:

- There is a specific server and multiple clients are connected to the server.
- A client sends the data to the server and the server broadcasts the data to all the other clients in the same room.
- There are no direct connections between the clients.

Peer-to-Peer (P2P) architecture:

- Each client is connected to other clients in the room directly.
- More availability since each client can act as a server and hence even if a central server is down, one can request the data from other clients.
- Harder to implement due to complex peer management and it needs careful implementation to maintain consistency in the global state.

Considering the time constraint and limited knowledge about p2p systems, we chose to stay with the client-server architecture for communication. Once we can complete this implementation satisfactorily, we will try to explore the p2p communication and implement it provided we have enough time.

TCP vs UDP sockets for Screen-sharing

TCP is a connection-oriented protocol whereas UDP is a connectionless protocol. TCP guarantees the delivery of the packet as sent by the client whereas UDP does not provide any guarantee on the same. So, for applications like chat, file-sharing etc., TCP is more suitable since it is important to deliver all the objects without any errors whereas UDP is more suitable in applications like streaming and screen sharing.

The reason for UDP's superior speed over TCP is that its non-existent 'acknowledgement' supports a continuous packet stream. Since TCP connection always acknowledges a set of packets (whether or not the connection is totally reliable), a retransmission must occur for every negative acknowledgement where a data packet has been lost. But because UDP avoids the unnecessary overheads of TCP transport, it's incredibly efficient in terms of bandwidth, and much less demanding of poor performing networks, as well.

Even though the UDP is preferred over TCP for screen-sharing since we have TCP communication for other modules, we decided to stick with TCP based screen-sharing and implementing it over UDP will remain our top priority during phase-2 of the project.

Design Patterns

The networking is independent of all the other modules and it is at the bottom of the dependency diagram.

Factory Method

The Factory Method design pattern defines an interface for creating an object, but let subclasses decide which class to instantiate. This pattern lets a class defer instantiation to subclasses. This ensures that the networking module can abstract the communication medium and would help us to move to communication over the internet without the other modules being concerned about this. The networking module will provide a *GetCommunicator* factory method which initialises the communicator object and returns an object of type *ICommunicator.*

Singleton Pattern

The Singleton design pattern ensures a class has only one instance and provide a global point of access to it.

Publisher-Subscriber Pattern

The Publish/Subscribe pattern, also known as pub/sub, is an architectural design pattern that provides a framework for exchanging messages between publishers and subscribers. This pattern involves the publisher and the subscriber relying on a message broker that relays messages from the publisher to the subscribers. The host (publisher) publishes messages (events) to a channel that subscribers can then sign up to.

The networking module provides the *INotificationHandler* interface which the subscriber must implement and subscribe to the networking module by calling the *Subscribe* method of the *Communicator* object. When the publisher (networking module) receives a message, it calls the *onDataReceived* method of handler provided by the subscriber.

Roles

Queue Management – Alisetti Sai Vamsi

Multiple modules can send and receive objects while the application is running. This submodule provides a queue interface that can be used by other submodules for buffering data before and after sending it over the network.

- Define an interface for the queue.
- Implement multi-level, priority-based queueing.
- Define the IQueue interface and implement it.

Sockets – Tausif Iqbal

This submodule is responsible for defining the way objects are sent across the network. It would depend on the *Queue Management* submodule to manage data storage at both the receiving and sending ends.

- Scan for a free port on the server and start listening.
- Responsible for transmitting data from a sender to all the clients in the same room.
- More specifically, this module listens to the sending queue on the sender's end and pushes the object to the appropriate queue (depending on the module which has sent the object) on the receiver's end.

Serialization – Abdullah Khan

Data received can from the modules can be of the type string, files or images. All that the socket submodule knows is to send *Packet* objects, hence this module is responsible for converting various types into *Packet* objects.

- Serialize input data object into an XML string.
- Deserialize XML string object and return the object for each module.
- Provide *Serialize* and *Deserialize* methods for modules to serialize and deserialize objects respectively.

Communication Manager – Abdullah Khan

Other modules use the networking module for communication, this submodule ensures implementing the singleton and factory design patterns. This can be used by other modules to communicate with the networking module.

- Define the *ICommunicator* interface.
- Define the *INotificationHandler* interface.
- Implement the *CommunicationFactory* class

Serialization

Overview

Serialization is a process of translating in-memory objects or object graphs (references) into a stream of appropriate format (bytes, XML or JSON etc.) for easy storage and transmission. These streams of data are reconstructed later (possibly on a different computer environment) to obtain a semantically identical copy of the original object while also maintaining the state of the object.

Design Choices

.NET framework provides three distinct serialization technologies; Data Contract, XML Serialization, and Runtime Binary Serialization each with its advantages and disadvantages. For our project, we have chosen XML Serialization due to its widespread support, better querying performance and more refined control over the output XML format. It provides better security compared to runtime serializers such as binary and SOAP; while also retaining its independence from the underlying architecture.

For the encoding format, we get the options of XML, Binary or JSON. JSON is a relatively new format with similar serialization performance while also being small and lightweight, but we chose XML due to the abundant support of this document-markup language and security compared to the other two formats.

Activity Diagram

Figure 1: Serializing and Sending an object

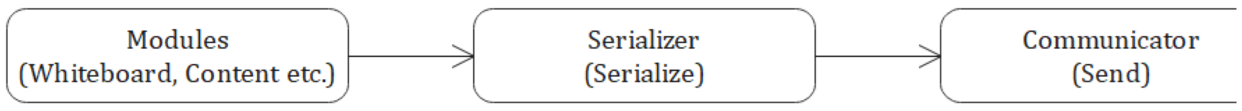
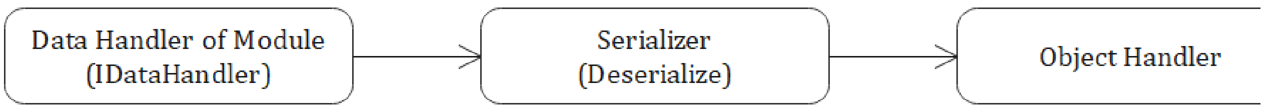
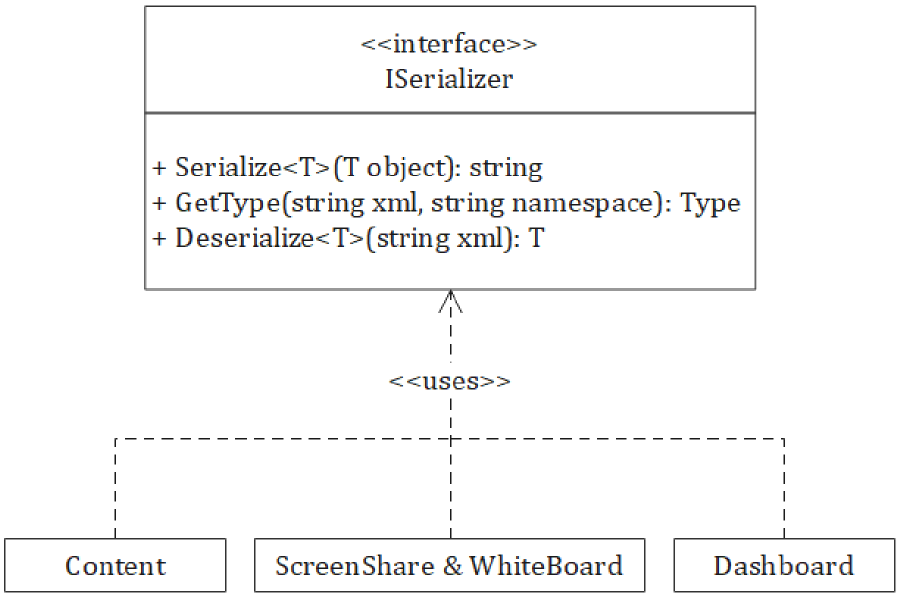


Figure 2: Receiving and Deserializing an object



Class Diagram

Figure 3: Class Diagram for Serialzation


<<interface>>
ISerializer
+ Serialize<T>(T object): string
+ GetType(string xml, string namespace): Type
+ Deserialize<T>(string xml): T
<<uses>>
Content
ScreenShare & WhiteBoard
Dashboard

Queue Management

Overview

Communication between computers has become incessantly imperative in these modern times where everyone owns a device that can be connected to the internet. Most of the applications developed contain a networking module that enables the application to communicate with other devices. Exploring the underlying architecture of such a networking module can be quite travailing. This design spec provides a broad overview of how the networking module functions and specifically focuses on a submodule called Queue Management and its implementation.

Background

Queue Management forms an essential part of a Networking Module. In computer networking, queuing delay forms the majority of the time lag between the exchange of information between two nodes. Queue Management in essence deals with maintaining a buffer in the form of a queue, in a particular node, to store and transmit the incoming packets from different nodes. Queue Management is used in software applications for the same reasons, except that here the nodes from which we receive information are different modules in the application that are trying to communicate with another device. Hence, to increase the performance of our communication system it is imperative to have a fast and robust queue implementation that can buffer all the incoming messages in parallel and service each of them to the listening sockets.

Problem

Consider two systems A (sender machine) and B(receiver machine) running the same application. Assume that the application consists of several modules of which some require communication like file sharing, messaging, screen sharing etc. Now machine A decides to send a message to machine B via the application. A rudimentary approach is when the send function in the application is called, the message is instantly put into the open socket and the send function is blocked until the socket is free. This way whenever the socket is free, the send function is unblocked. This approach causes other modules which want to send a message, to wait that results in superfluous use of the computational resource. Once the message is received on machine B, the communication module should make sure that the appropriate module is receiving this message. Again, another rudimentary approach is to make all the modules listen on the open socket and process the message at the same time to determine which module it belongs to. This again has its conspicuous fallacies, as each module should be processing the message to discern whether it belongs to the module or not. The goal is to deal with this predicament of having multiple modules wanting to send messages and continue their execution on the sending side and to send the message to the appropriate module on the receiving side.

Design Choices

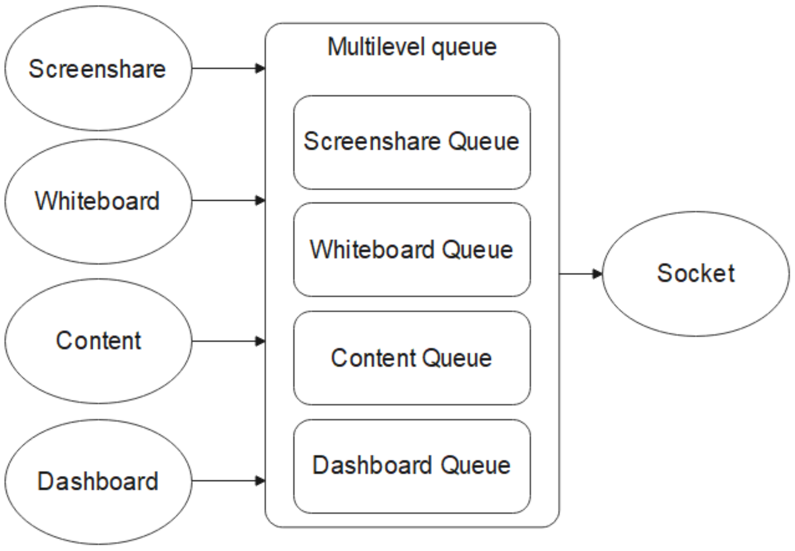
Queue Management plays different roles depending on whether it is a sender/receiver. On the sender side, the task of queue management is to provide a stable data structure to collect the stream of messages from all the other modules that want to send messages. On the receiving side, it needs to collect the message and send it to the appropriate module.

Concurrent Single FIFO Queue

A global queue handles the queue management where each module enqueues items into the global queue. On the other end, the socket keeps listening on this queue and when it finds an item in the queue it dequeues it and transmits it across the network.

Concurrent Multilevel Queues

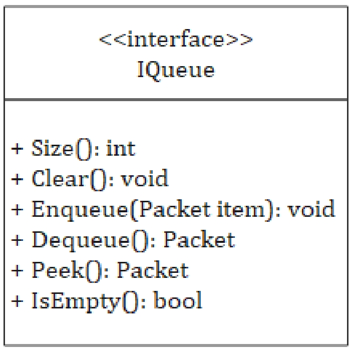
Figure 4: Multilevel Queue



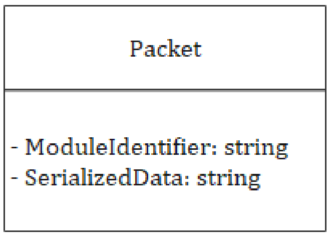
Multilevel queue maintains 4 sets of individual queues for each type of module. Each module takes in the message and enqueues it into the appropriate queue. Each queue is given a certain weight, i.e. for example whiteboard, screen share, files, chats are each given a weight of 4,3,2,1 respectively based on the priority. This means that while dequeuing only after 4 whiteboard dequeues we can dequeue the screen share. Only after 3 screenshare dequeues we can dequeue files. If in any case, the queue is empty, we skip it and go on to deque the next queue. In this way, we define our priority since 4 a regular implementation of priority queue based on heaps will always have the most prioritized one at the top of the heap which will starve the remaining modules. The concurrent nature of the queue system makes it possible for a single thread to enqueue a data item, and a single thread to dequeue a data item.

Module Specification

Queue Interface

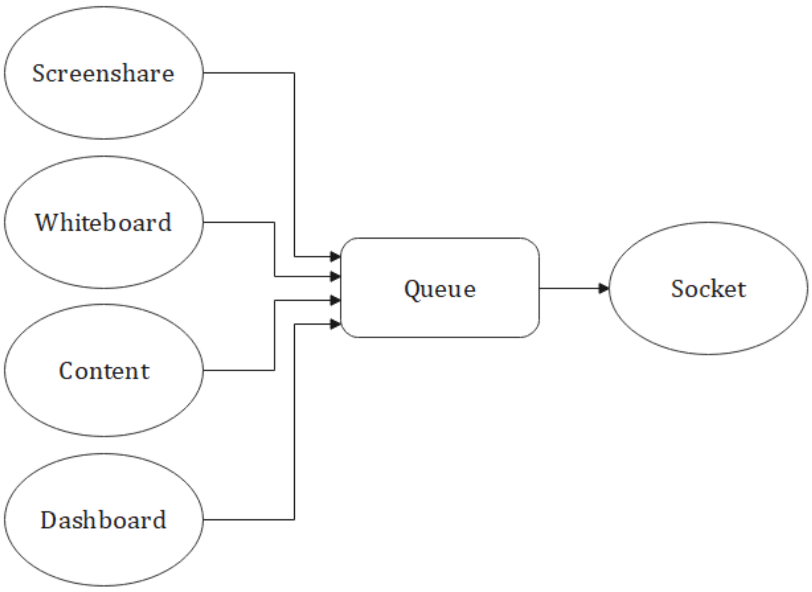


Packet Class



Sender Queue

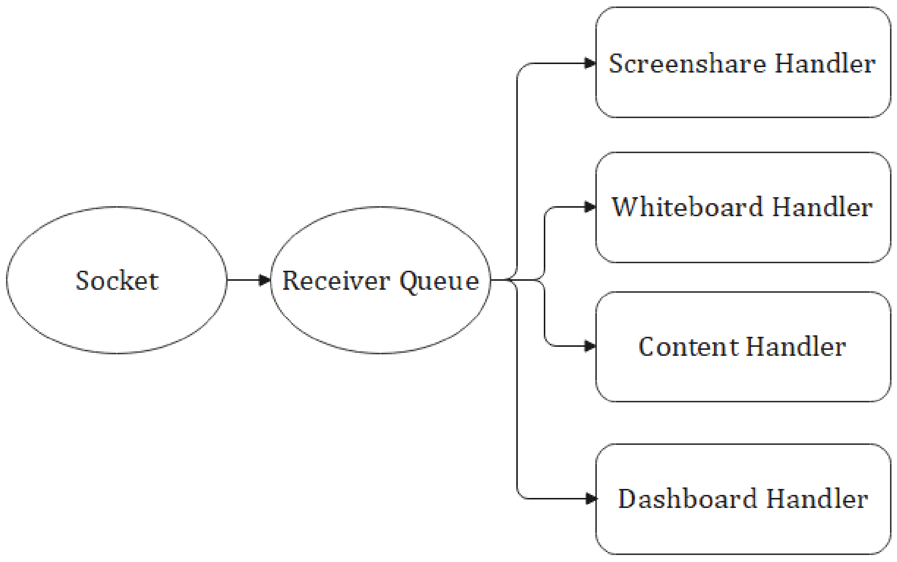
Figure 5: Sending Queue



Sending queue is a part of the socket listener which will be covered in detail in the Sockets section.

Receiver Queue

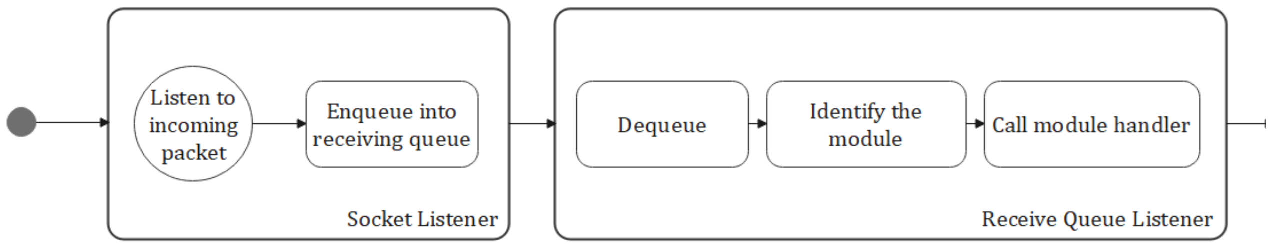
Figure 6: Receiver Queue



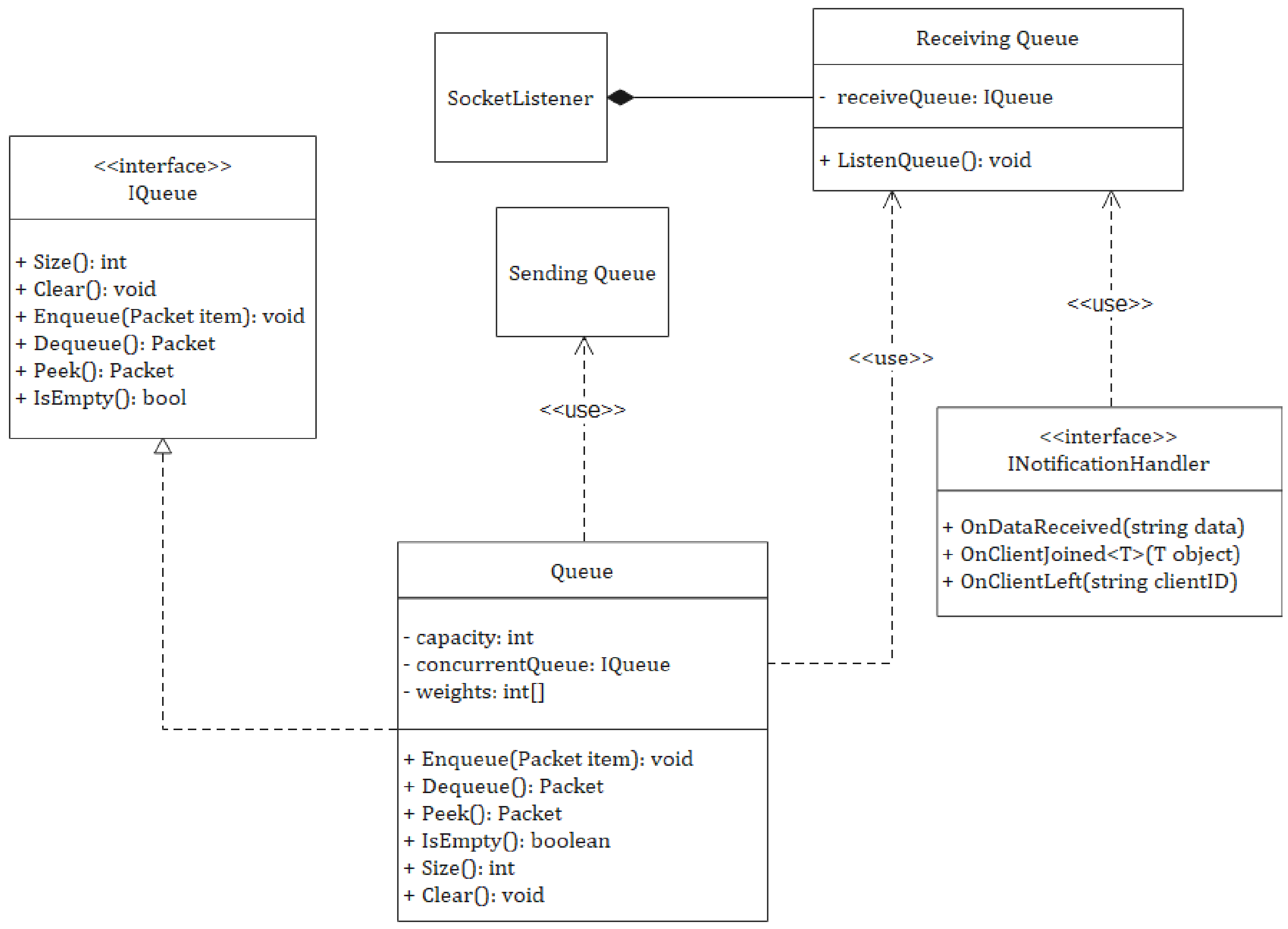
The receiver queue system contains a single receiving queue that is enqueued by the socket listener module. The receiver queue module listens on this queue. If this queue is not empty, then the queue starts to dequeue packets from the queue. These packets are then scanned to identify the module they belong to and the corresponding module handler is called.

Activity Diagram

Figure 7: Activity Diagram



Dependency Diagram



Sockets

Overview

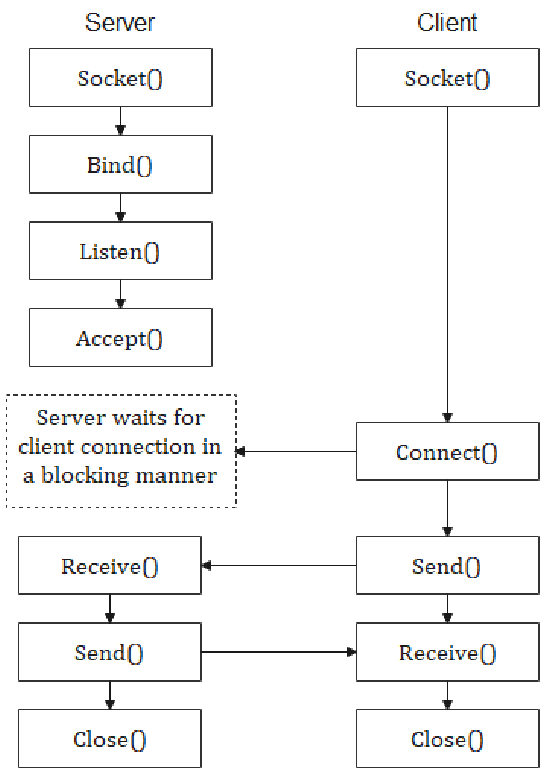
The Networking Module is for communication. To transfer data from one host to another host we need a network module. When the Content module, ScreenShare module, Whiteboard module, and Dashboard module need to send data, they give it to the networking module with required details, networking module, in turn, sends the data to the destination.

Responsibility

1. Scan for port free port and start the server on that port.
2. On the client-side, take a packet from the queue and send it to the server.
3. On the receiving end, receive the byte-stream, convert it into a *Packet* object and enqueue the object to the queue.

About Sockets

The socket is an endpoint of communication between 2 devices. Sockets use nodes' IP addresses and a network protocol to create a secure channel of communication and use this channel to transfer data. In socket communication, one node acts as a listener and other node acts as a client. The listener node opens itself upon a pre-established IP address and on a predefined protocol and starts listening. Clients who want to send messages to the server start broadcasting messages on the same IP address and same protocol. A typical socket connection uses the Transmission Control Protocol (TCP) to communicate.



Design Choices

Protocol: TCP

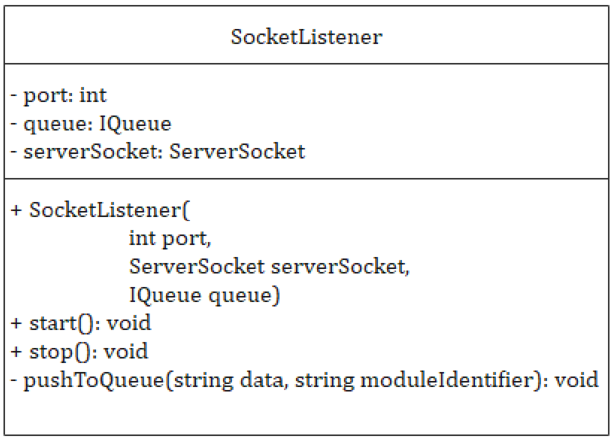
It is a communication protocol. Connection-orientation means that the communicating devices should establish a connection before transmitting data and should close the connection after transmitting the data. TCP provides extensive error-checking mechanisms. It is because it provides flow control and acknowledgement of data. It is slower than that of UDP but more reliable.

Fragmentation

In TCP we can only send 65,535 bytes in a single packet. So, if the information that needs to be sent is more than the threshold, we divide the message into chunks that can be sent over the network using TCP.

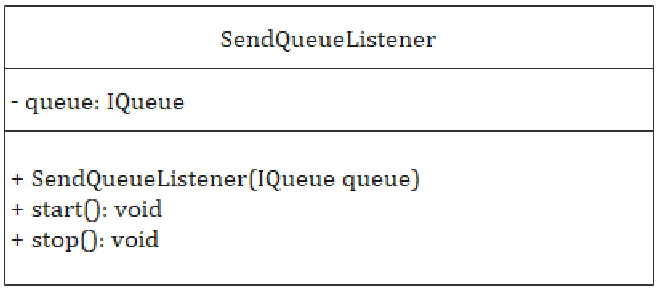
Socket Listener

This thread continuously listens for incoming data and after receiving data from the network it pushes into the 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

This thread continuously listens for incoming data and after receiving data from the network it pushes into the queue.

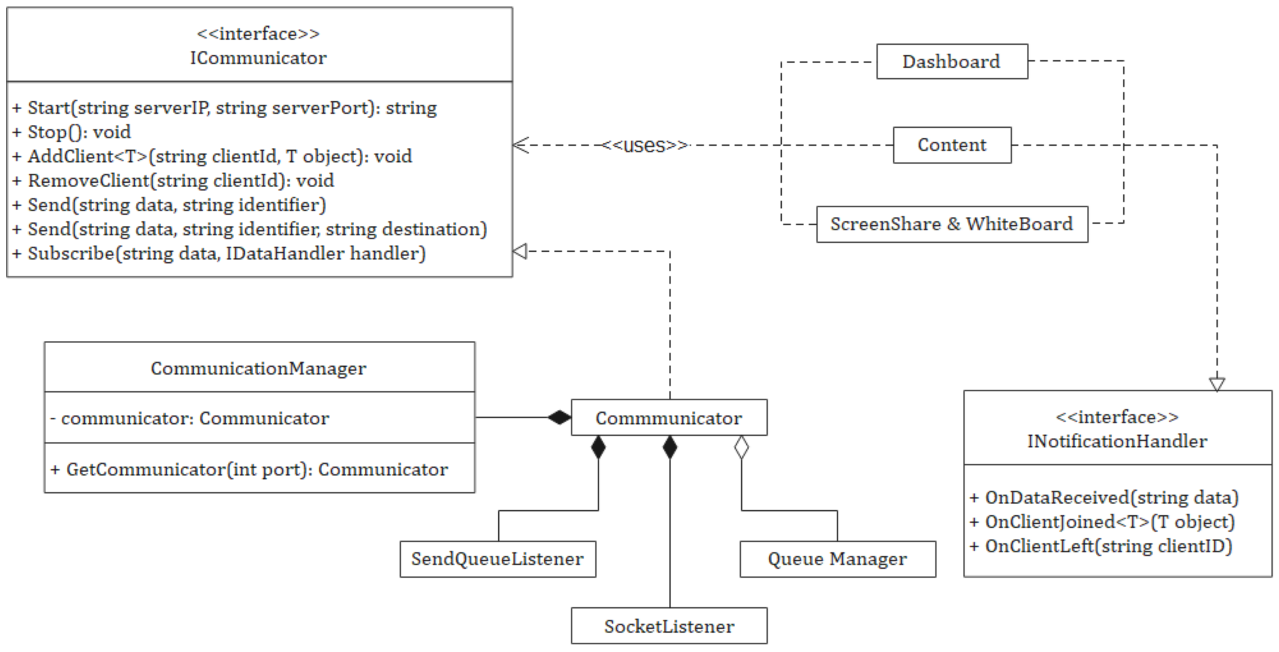
Send Queue Listener



This thread continuously listens for data from the receiving queue, when there is some data in receiving queue, it takes the data and sends it over the network.

Class Diagram

Figure 8: Class Diagram for Socket Manager

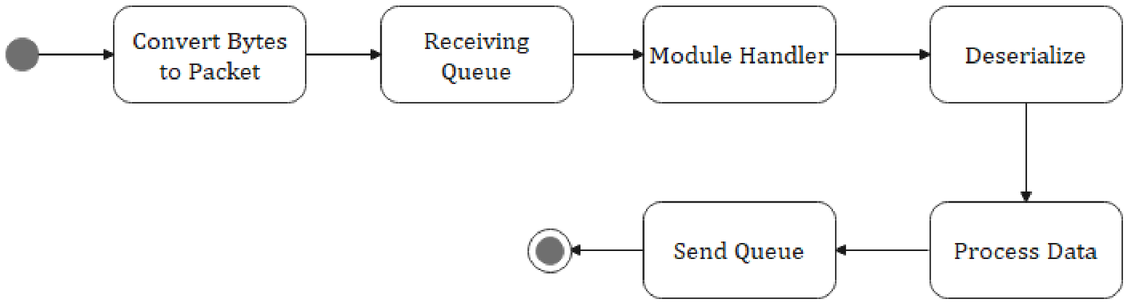


<<interface>>
ICommunicator
+ Start(string serverIP, string serverPort): string
+ Stop(): void
+ AddClient<T>(string clientId, T object): void
+ RemoveClient(string clientId): void
+ Send(string data, string identifier)
+ Send(string data, string identifier, string destination)
+ Subscribe(string data, IDataHandler handler)
<<uses>>
Dashboard
Content
ScreenShare & WhiteBoard
CommunicationManager
- communicator: Communicator
+ GetCommunicator(int port): Communicator
Commmunicator
SendQueueListener
SocketListener
Queue Manager
<<interface>>
INotificationHandler
+ OnDataReceived(string data)
+ OnClientJoined<T>(T object)
+ OnClientLeft(string clientID)

Program Flow

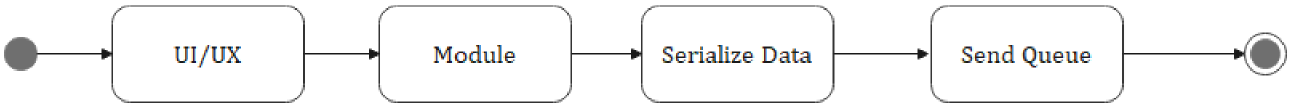
Server Side

The server socket(socket-listener) will continuously listen for data when some data comes, socket-Listener will capture it and after converting byte into packet object it will push into the receive queue. The Receive queue identifies the message and calls the corresponding handler(content, dashboard, screenshare&whiteboard), Handler will deserialize the message and give it to the module for processing and when the module is done with processing it will push into sending queue, at the end of the queue socket (send-queue-listener) will broadcast the data.

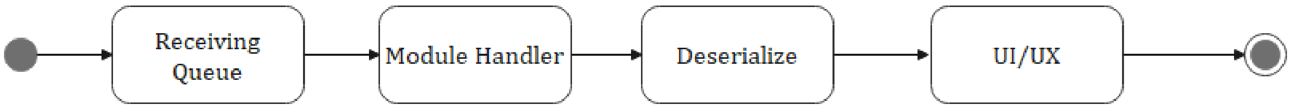


Client-Side

Data from the client will first go to a module (content, dashboard, screen share and whiteboard), which after processing calls serializer, from there it will go to the send queue, and at the end of the send queue, a socket will send the message.



In the receiving end of the client, a socket will listen for data, when data is captured by the socket then it converts the byte into a packet and pushes it into receiving queue, at the end of the queue handler will call the corresponding module (content, dashboard, screen share &whiteboard) and after deserializing the data module will process it and from the module, data comes to UI.



Communication Manager

Overview

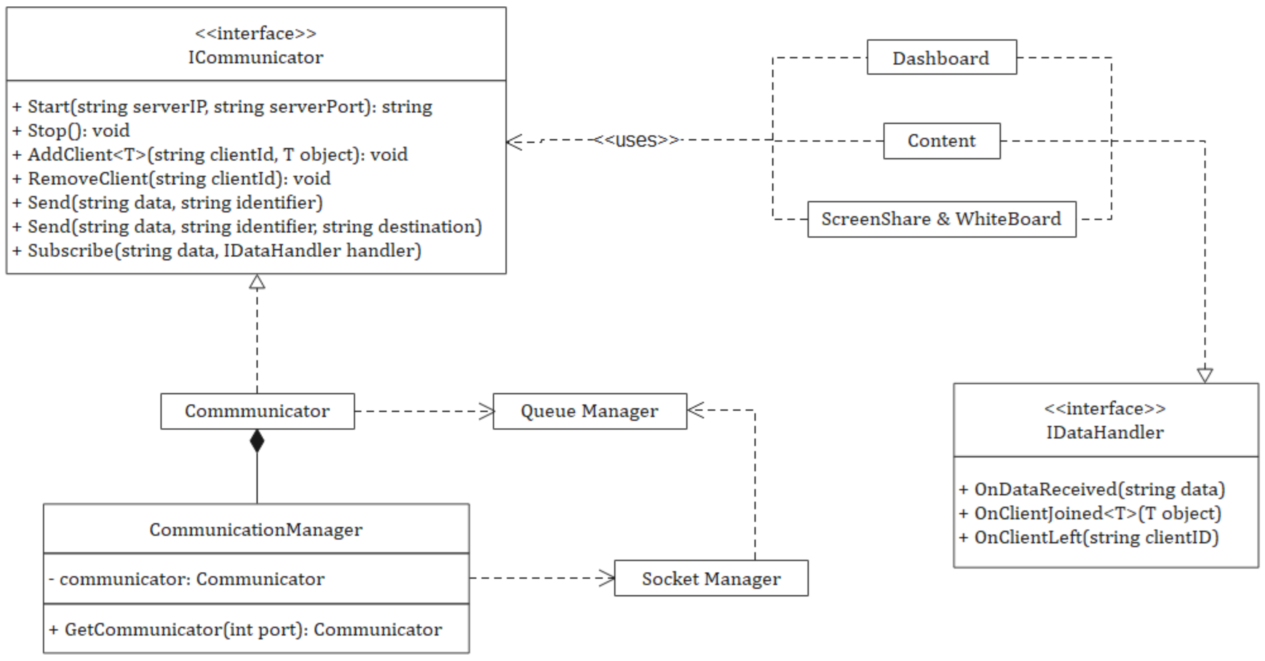
The communication Manager is responsible for managing the internal structure of the networking class and implementing the communicator in a singleton factory pattern. The Communication Manager gives a communicator to the user which is the primary source of communication to and from the server. The communicator class is responsible for sending the message and calling the appropriate message handler upon receiving a message.

Design Choices

Communication Manager is implemented using a singleton factory pattern, which invokes only a single instance of the communicator class. A single networking instance is required as all the packets going outside this machine will be identified as coming from the same source. This reduces confusion and decreases the complexity of the solution. Singleton pattern is often used when using the factory method pattern; which abstracts the internal workings and instantiations of its subclasses behind a layer of public interface.

Class Diagram

Figure 9: Class diagram for the CommunicationManager



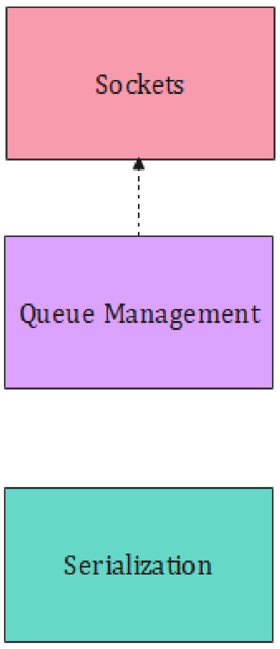
<<interface>>
ICommunicator
+ Start(string serverIP, string serverPort): string
+ Stop(): void
+ AddClient<T>(string clientId, T object): void
+ RemoveClient(string clientId): void
+ Send(string data, string identifier)
+ Send(string data, string identifier, string destination)
+ Subscribe(string data, IDataHandler handler)
<<uses>>
Dashboard
Content
ScreenShare & WhiteBoard
Commmunicator
Queue Manager
<<interface>>
IDataHandler
+ OnDataReceived(string data)
+ OnClientJoined<T>(T object)
+ OnClientLeft(string clientID)
CommunicationManager
- communicator: Communicator
+ GetCommunicator(int port): Communicator
Socket Manager

The Communication Factory create a single instance of the communicator object which is dependent upon the Queue Manager class for enqueuing the message. This object also notifies the subscribed module upon receiving a message by using the Message Listener object implemented by each subscribed module.

Depending upon the type of instance {client or server} appropriate interface will be instantiated. The client communicator doesn't have access to broadcast and only uses send method which send the serialized XML string message to the server. The server receives the message and appropriate server-side handler is called by the private notify method. These handlers can either broadcast the message or send to a single client. The *CommunicationManager* gives a communicator object using the *GetCommunicator* method.

Dependency Diagram

Figure 8: Dependency of each submodule on the other



The queue management submodule does not depend on any other modules. It provides interfaces like *enqueue, deque, size* and other standard queue methods and properties. This submodule implements a multi-level priority queue and abstracts its implementation from other modules. The receive queue listener listens to the queue on the receiving end, either on the server or the client, it pops a packet from the queue if available and calls the handler of the appropriate module.

The serialization submodule is responsible for converting objects of other modules into an XML string, which will be used by the socket submodule to send across the network. This submodule is also responsible for deserializing the XML string to the correct object type and on the client's receiving end.

The sockets submodule is responsible for sending *Packet* objects across the network, it depends on the serialization submodule for serializing objects from other modules into the *Packet* object and it depends on the queue management to buffer packets before sending and it adds the packet to the appropriate queue after it has received a packet on client's receiving end.

Activity diagrams

Figure 9: General program flow to send any object

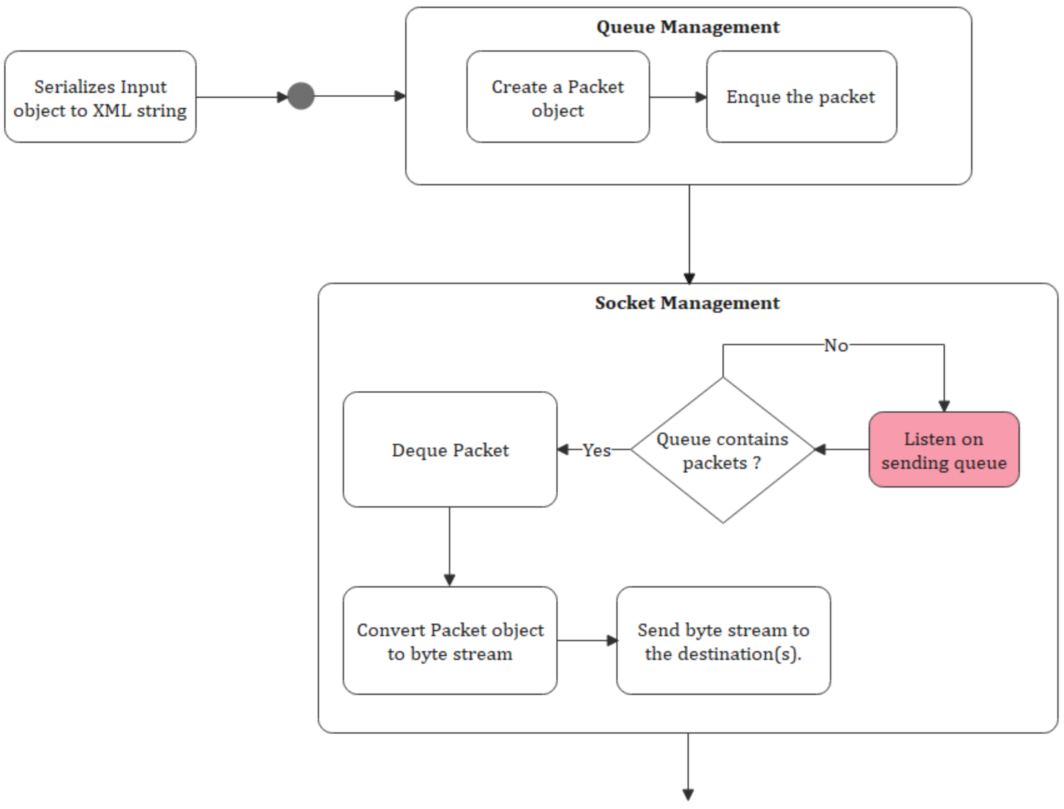


Figure 10: Sending objects from a client to server, the "Send to Server" activity follows the diagram as shown in Figure 9

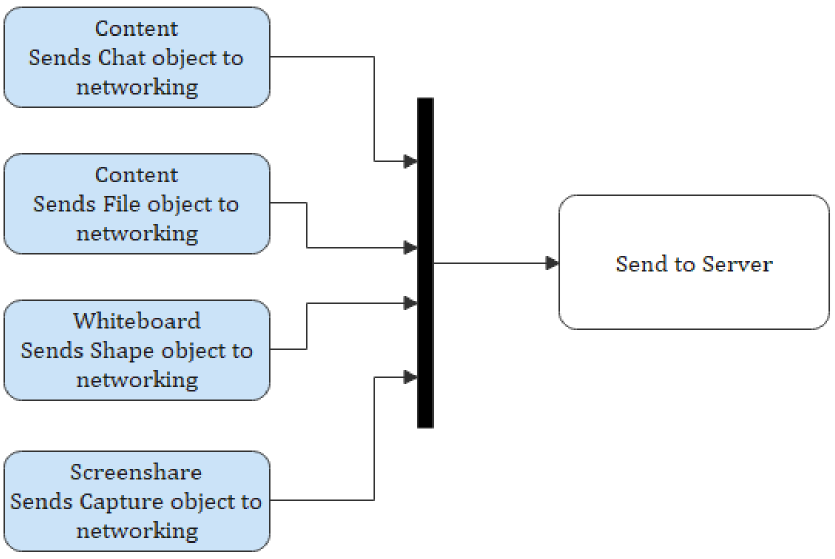
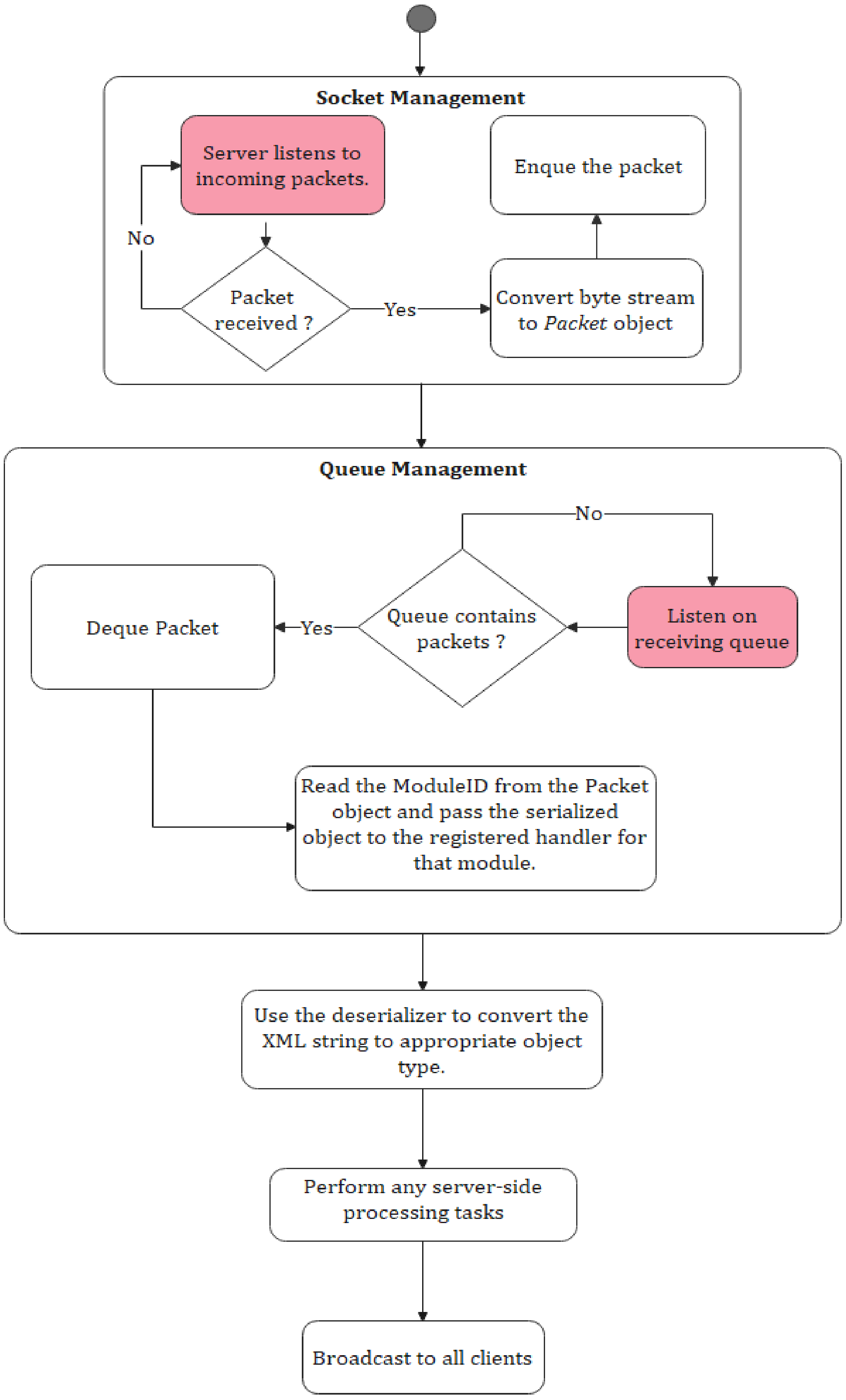
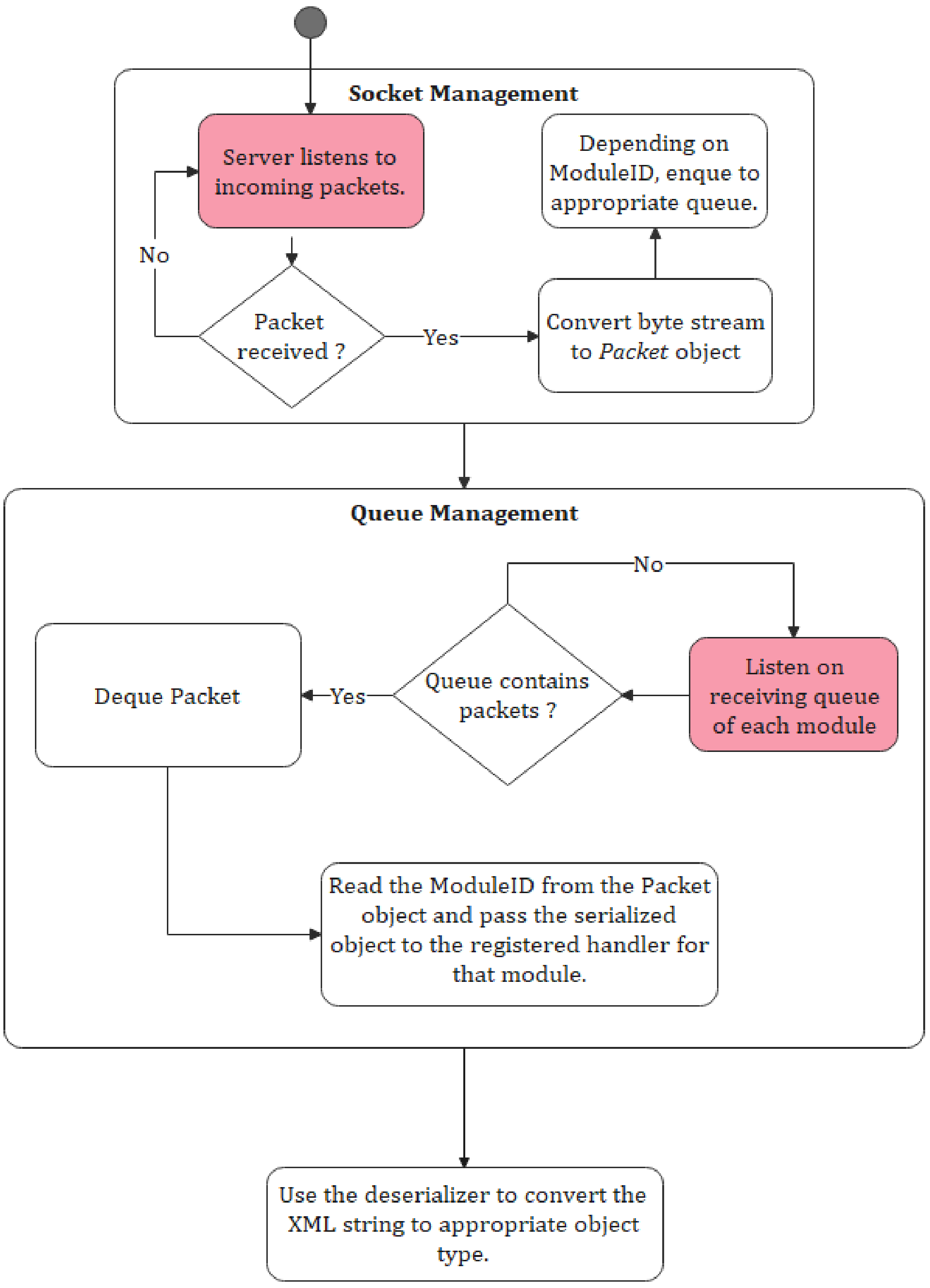

Content
Sends Chat object to networking
Content
Sends File object to networking
Whiteboard
Sends Shape object to networking
Screenshare
Sends Capture object to networking
Send to Server

Figure 11: Receive packets on server and broadcast to all clients, "Broadcast to all clients" activity follows the diagram as shown in Figure 9



Socket Management
Server listens to incoming packets.
No
Packet received ?
Yes
Convert byte stream to *Packet* object
Enque the packet
Queue Management
No
Listen on receiving queue
Queue contains packets ?
Yes
Deque Packet
Read the ModuleID from the Packet object and pass the serialized object to the registered handler for that module.
Use the deserializer to convert the XML string to appropriate object type.
Perform any server-side processing tasks
Broadcast to all clients

Figure 12: Receive packets on client and call the handlers of concerned modules


Socket Management
Server listens to incoming packets.
No
Packet received ?
Yes
Convert byte stream to *Packet* object
Depending on ModuleID, enque to appropriate queue.
Queue Management
No
Listen on receiving queue of each module
Queue contains packets ?
Yes
Deque Packet
Read the ModuleID from the Packet object and pass the serialized object to the registered handler for that module.
Use the deserializer to convert the XML string to appropriate object type.

Interfacing with other modules

Sending an object

To send the object of a module over the network, the serialization submodule implements the *Serialize* and *Deserialize* methods for that module. Before calling the send or broadcast function, each module must serialize its object to an XML string.

The module must extend the *INotificationHandler* interface and implement the *onDataReceived* method, once this is implemented, the module has to register a handler using the object of its custom handler class, which the queue management submodule uses to pass the XML string. The module can then deserialize the object using the deserializer implemented by the serialization submodule.

On the server, each module that needs to send objects over the network has to use its custom object handler to receive objects using which it can perform any necessary processing steps such as removing abusive words in chat, storing the data for future usage etc. and after it finishes the intermediate steps, the module has to call the *Send* function so that the networking module can send the updated object to the client(s).

Note: If a module requests to send an object over the network, the networking module, during the broadcast, sends the object to all clients (including the sender).

Let us say the content module wants to send a chat message to all the other clients.

Initialisation steps:

1. The content module subscribes to the networking module (both on client and server) by calling the *subscribe* method and passing the *moduleID* and *INotificationHandler* object.

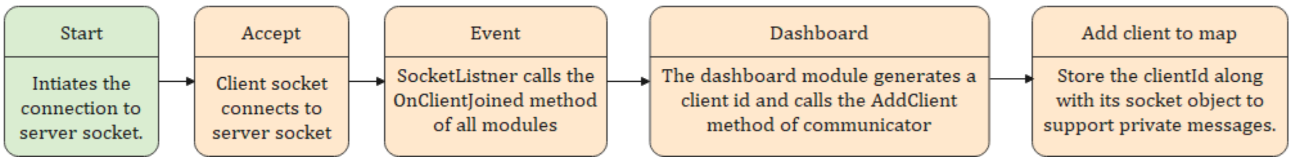
Sending the message:

1. The content module builds the *Chat* object using the data from the UX module. Let us name this object as *chat.*
2. The chat module has to serialize the chat object to an XML string (named *serializedChat*) using the serializer implemented by the serialization submodule.
3. Now the chat module calls the *Send* function on the client-side with parameters as *serializedChat* and *moduleID.*
4. The networking module sends this message to the server and on the server-side calls the *onDataReceived* method of the handler provided by the content module with the XML serialized string of the *chat* object.
5. The content module deserializes the *chat* object using the deserializer implemented by the serialization submodule.
6. The content module performs any necessary steps on the server-side and performs steps 22 and 22 to broadcast the message to all clients.

Receiving the message on client-side

1. On the client-side, the networking module calls the *onDataReceived* of the handlers provided by the content module with the XML serialized string of the *chat* object.
2. The chat module has to deserialize the message using the deserializer implemented by the serialization submodule.

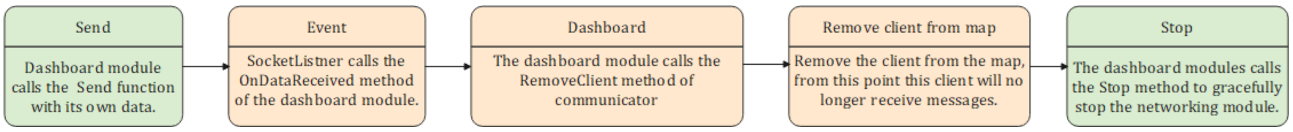
Client Joins the room


Start
Intiates the connection to server socket.
Accept
Client socket connects to server socket
Event
SocketListner calls the OnClientJoined method of all modules
Dashboard
The dashboard module generates a client id and calls the AddClient method of communicator
Add client to map
Store the clientId along with its socket object to support private messages.

When a client joins the room, the dashboard module must call the *Start* method of the networking module which takes in the IP address and port of the server as input. It sets up other sub-modules in networking and tries to connect to the server, on a successful connection, the server receives the *socket* object of the client through which it can send and receive messages.

Now, to be able to send private messages we need to be able to identify each client's socket object uniquely. To do this, we use the client id generated by the dashboard module so as soon as a client connects to the socket, we call the OnClientJoined event of the handler of all modules with the socket object as the parameter. The dashboard module receives this event and it generates a new user id and user object and calls the AddClient method of the communicator object by passing the client id and the socket object that passed previously. The networking module stores the tuple (client id, socket object) into a map and uses the client id to identify the client uniquely to sent private messages in the future.

Client Leaves the room


Send
Dashboard module calls the Send function with its own data.
Event
SocketListner calls the OnDataReceived method of the dashboard module.
Dashboard
The dashboard module calls the RemoveClient method of communicator
Remove client from map
Remove the client from the map, from this point this client will no longer receive messages.
Stop
The dashboard modules calls the Stop method to gracefully stop the networking module.

When the client decides to leave the room, the dashboard module will send an object to the server through the networking module, once it receives the message and realizes that it is a client departure event, the dashboard module calls the *RemoveClient* method of the *communicator* object. This ensures that the client is removed from the map of the networking module and hence does not receive further messages until the client re-joins the room.