Plex Share

Networking Module - Design Specifications

CS5617: Software Engineering
September 2022

Bhagavatula Anish - 111901016 (Team Lead)

Table of Contents

Team

111901016	Anish Bhagavatula (Team Lead)
111901031	Mohammad Umar Sultan (Team Member)

Overview

Communication between various clients and the server is ensured by the networking module. The module uses two priority queues for communication, one for high-priority messages and the other for low-priority messages. 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. All the other modules in the system depend on the networking module for data-communication over the network, but the networking module remains completely independent of any other module for its functioning.

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

Design Architecture

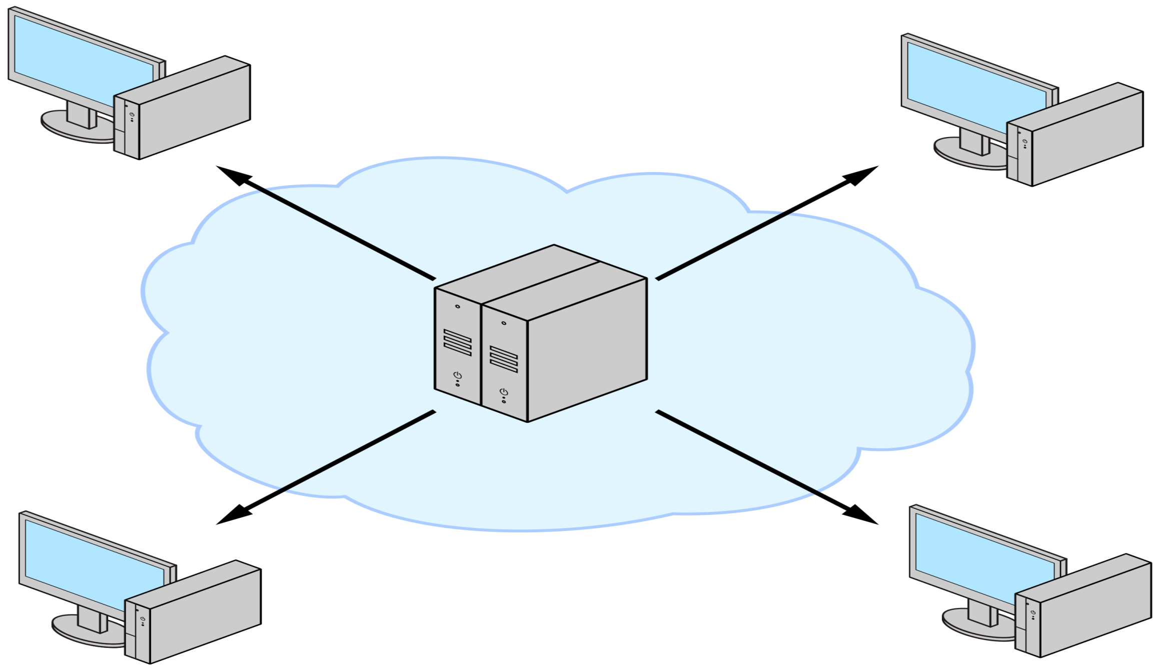
Network Architecture

We have 2 choices for the network architecture:

1. Client-Server Architecture
2. Peer-To-Peer Architecture

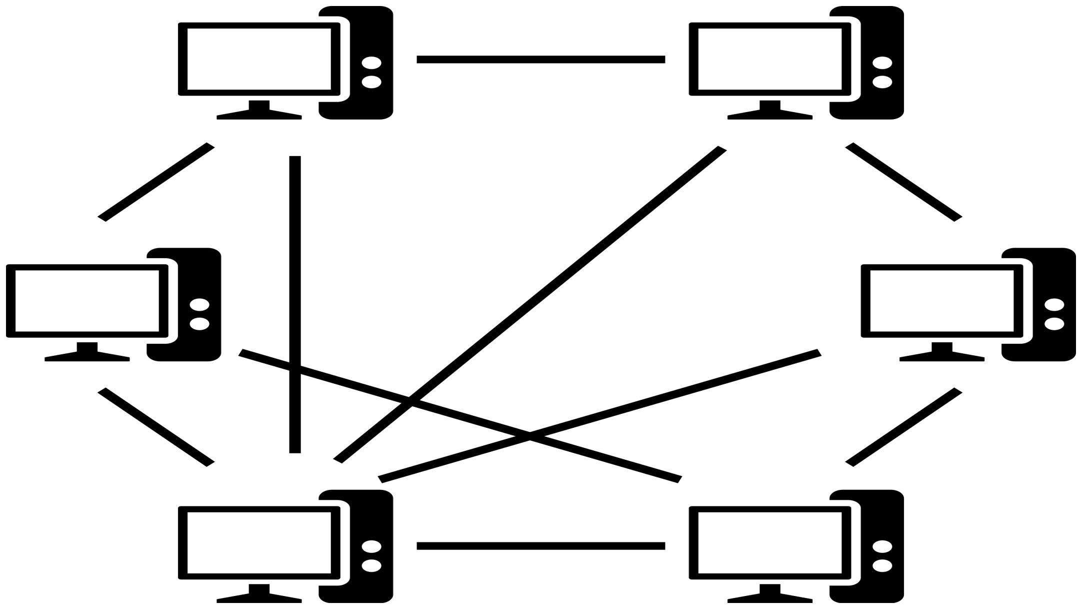
Client-Server Architecture

The client-server design model involves the server hosting and delivering all the data requested by clients. This model employs a design in which all data is constituted in the server. Exchange of information between various clients takes place through the server and not directly from one client to another.



Peer-To-Peer Architecture

In the peer-to-peer architectural model, every client is connected to other clients directly without a centralized server, with each client acting as a server. Even if one client is down, information can be requested from other clients.



Choice: Client-Server Architecture

Implementing the peer-to-peer architectural model is very cumbersome to implement and maintain, especially when multitudes of clients are interacting with each other. In any case, for simplicity, we have chosen to go for the client-server architectural model for the networking module.

Submodules

Serialization

Clients communicate with other clients and the server through information contained in different formats. For example, communication might be via files in XML format, JSON format etc.

But the networking module which facilitates the transmission of information is indifferent to the format of information and is only involved in transmission of packets.

So, the module provides a serialization interface, which is used by other modules, to encode information into strings and also to decode from strings to the original format.

Queues

The purpose of this submodule is to provide a queue interface as a buffer for the networking module. Every module uses the networking module as an intermediary to facilitate communication.

The networking module letting through many packets of information from various modules ensues a bottle-neck situation.

For this reason, the queue interface is provided, ensuring buffers while transmitting and receiving information from one client to another.

Sockets and Communication

Present in Umar's specification.

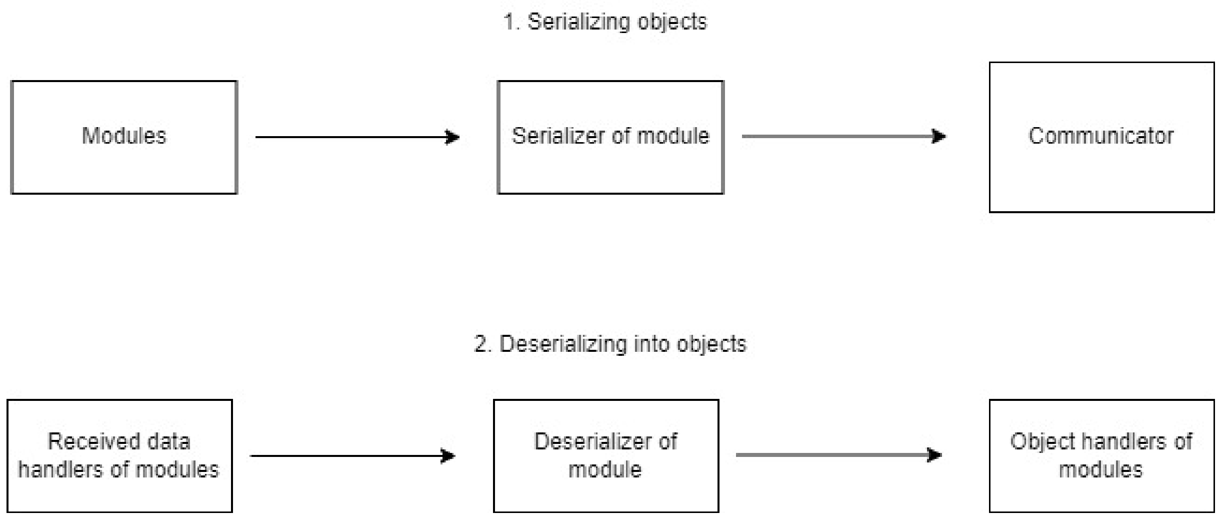
Serialization

About

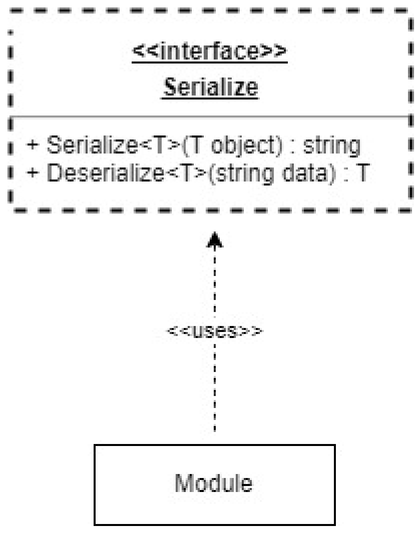
Serialization is the process of converting a data object into a byte stream for ease of storage and transmission. This encoded information can always be re-constructed back to its original form.

.NET offers three kinds of serialization, namely Binary serialization (converting .NET objects into byte stream), SOAP serialization (converting .NET objects into encoded SOAP messages) and XML serialization (converting .NET objects into XML).

Owing to the widespread use and popularity of XML serialization, we have chosen it as our serialization technique for this project.


1. Serializing objects
Modules
Serializer of module
Communicator
2. Deserializing into objects
Received data handlers of modules
Deserializer of module
Object handlers of modules

Class Diagram


<<interface>>
Serialize
+ Serialize<T>(T object) : string
+ Deserialize<T>(string data) : T
<<uses>>
Module

Queues

About Transmission

As many modules might be willing to communicate at the same time, there arises a bottle-neck situation at the networking module, where data packets of multiple modules are in contention to be transmitted across.

To tackle this situation, coming up with a buffer to store packets of all modules and transmitting each of them one after the other is imperative.

Design Choice

Single queue

The submodule may implement a single global queue which is accessible to every other module for the purpose of queueing. In this case, every module uses the same queue for buffering the messages which need to be transmitted.

But adopting this design would ignore priorities of messages from the modules. An important message might have to wait until all messages queued ahead are processed and transmitted.

Introducing as many queues as the number of modules

To solve the problem of priority of messages which was ignored in the single-queue design, we might assign a separate queue for every single module. So every module which needs to transmit a message would have to use a separate queue specific to the module.

This way, depending on the priority of messages of each module, a particular module's messages might be processed before catering to other modules.

This approach has a few issues -

1. **Exhaustion -**

   Incessantly serving prioritized modules might leave other modules starving for long periods, and even forever in some cases.

2. **Independence -**

   If this design is adopted, then the networking module would have to be well aware of the number of modules present in the system, which is not a reliable design.

A good software module must be robust and independent of other modules. Adding a new module to the system in the future would demand the networking module to be tweaked, which is inconvenient and not durable in the long run.

Multiple (pre-decided number of) queues

Introducing a few pre-decided number of queues, where each one caters to buffer messages belonging to a certain priority level is another design which can be employed.

Choice : Multiple (pre-decided number of) queues

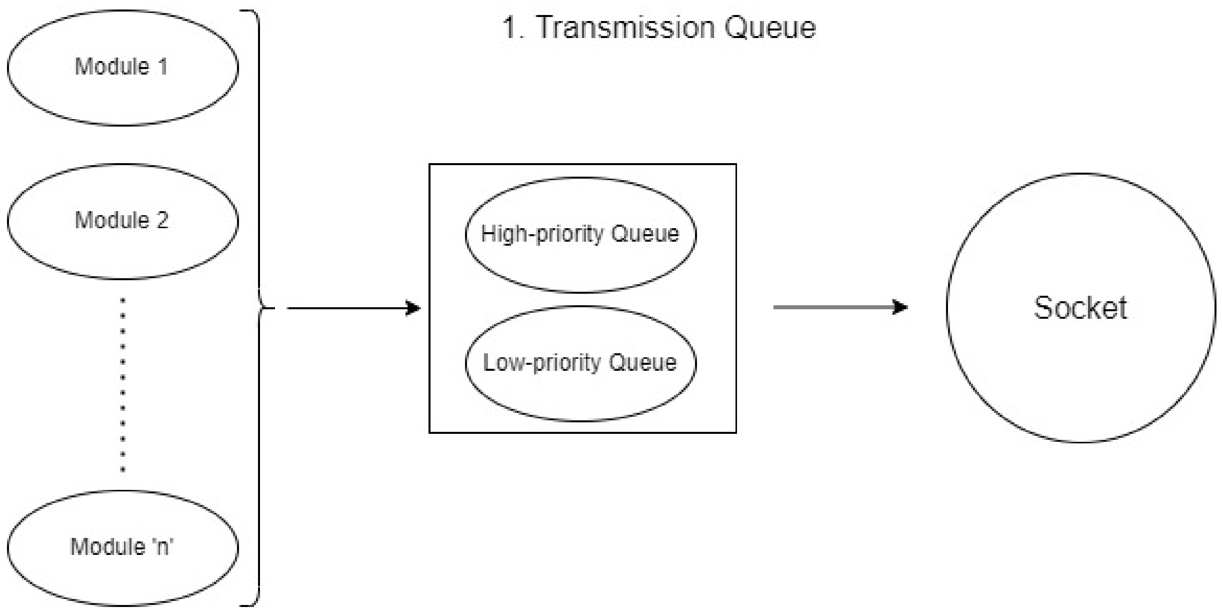
We have gone with implementing two queues, one dedicated specifically for high-priority messages and the other for low-priority messages.

We could have chosen many queues to entertain different priority levels, but we have decided to go with only a couple of them, for simplicity.

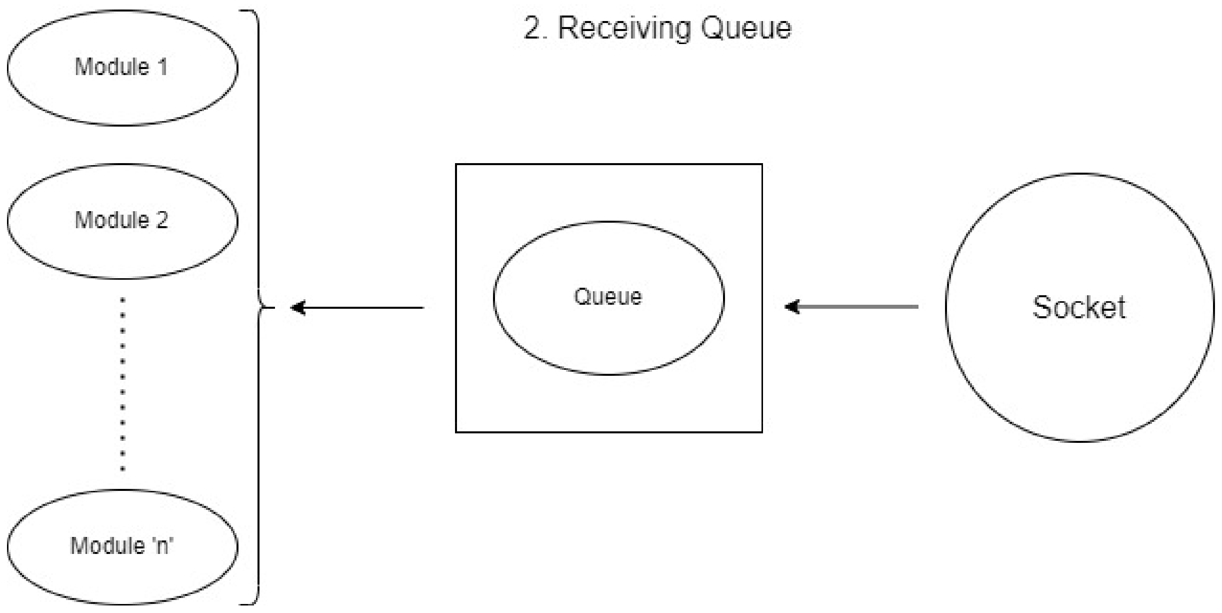
To evade the problem of exhaustion, we are going to transmit messages in a 2:1 ratio in favor of the high-priority messages.

This also ensures that the networking module is oblivious to the number of modules in the system, and introduction of newer modules in the future would be smooth and comfortable.

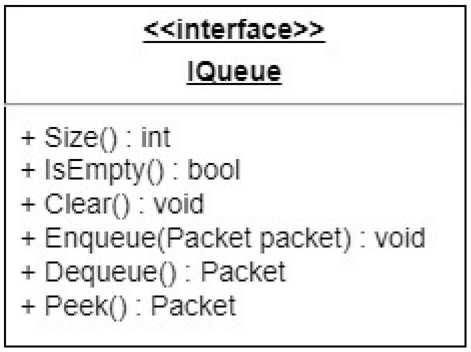
Sending Queue



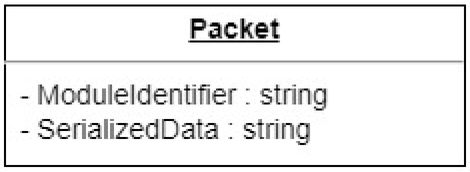
Receiving Queue



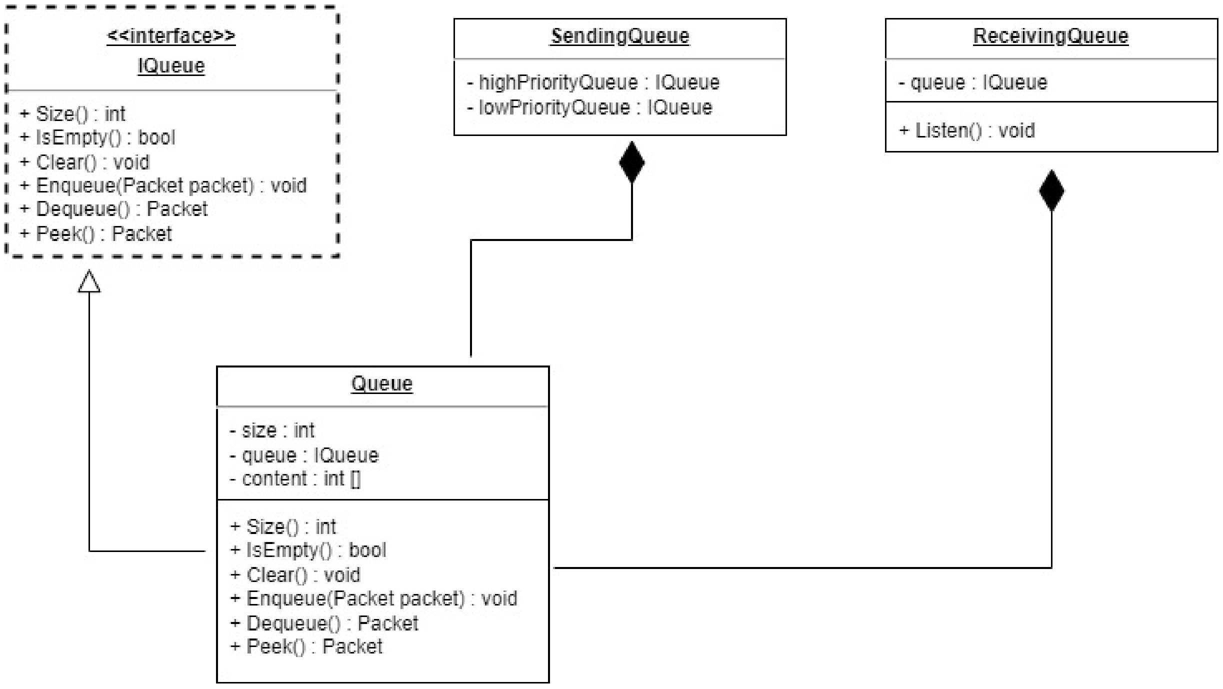
Specification



The packet class stores the module which the packet belong to, and the data after serializing

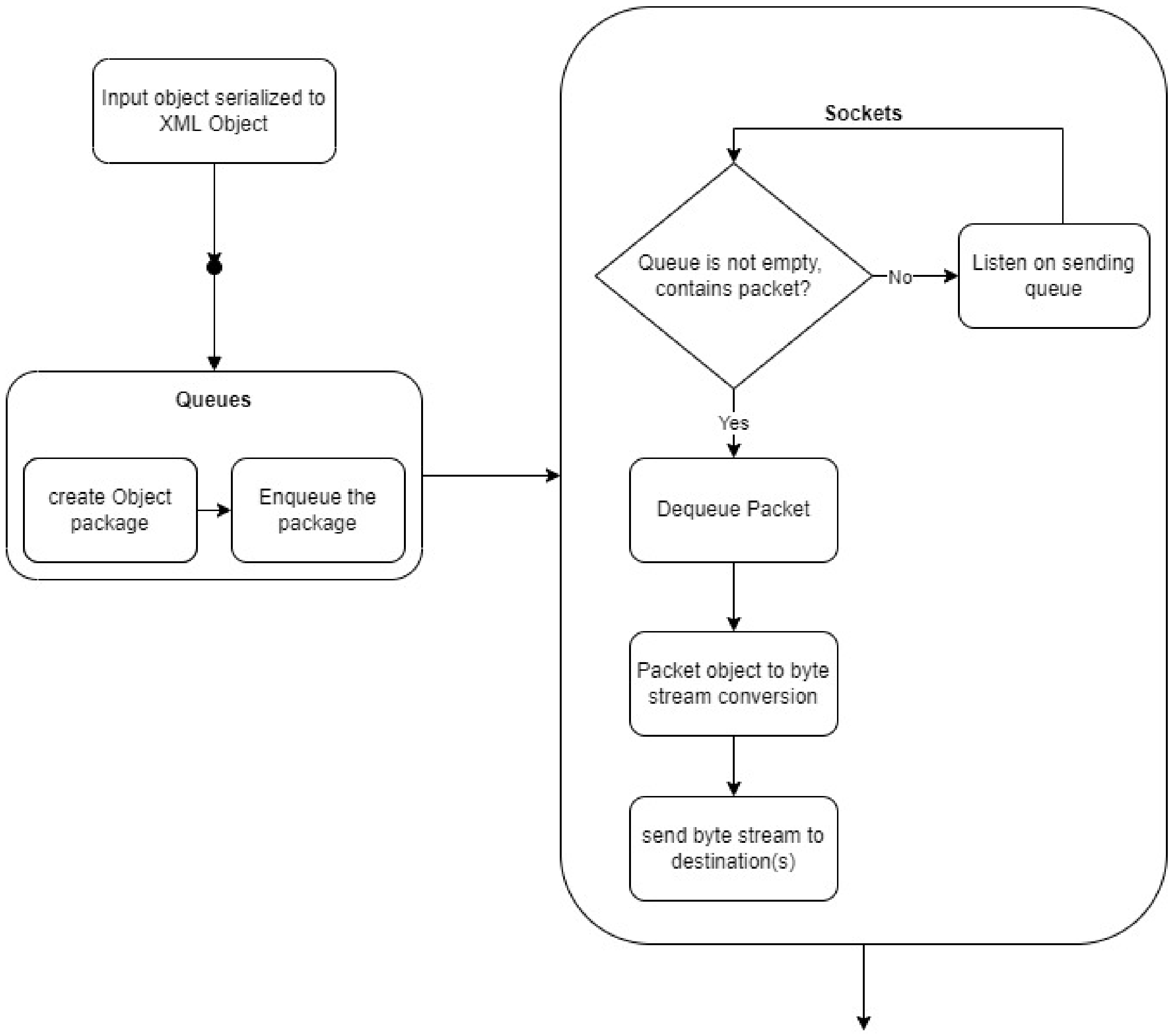


Dependency Diagram

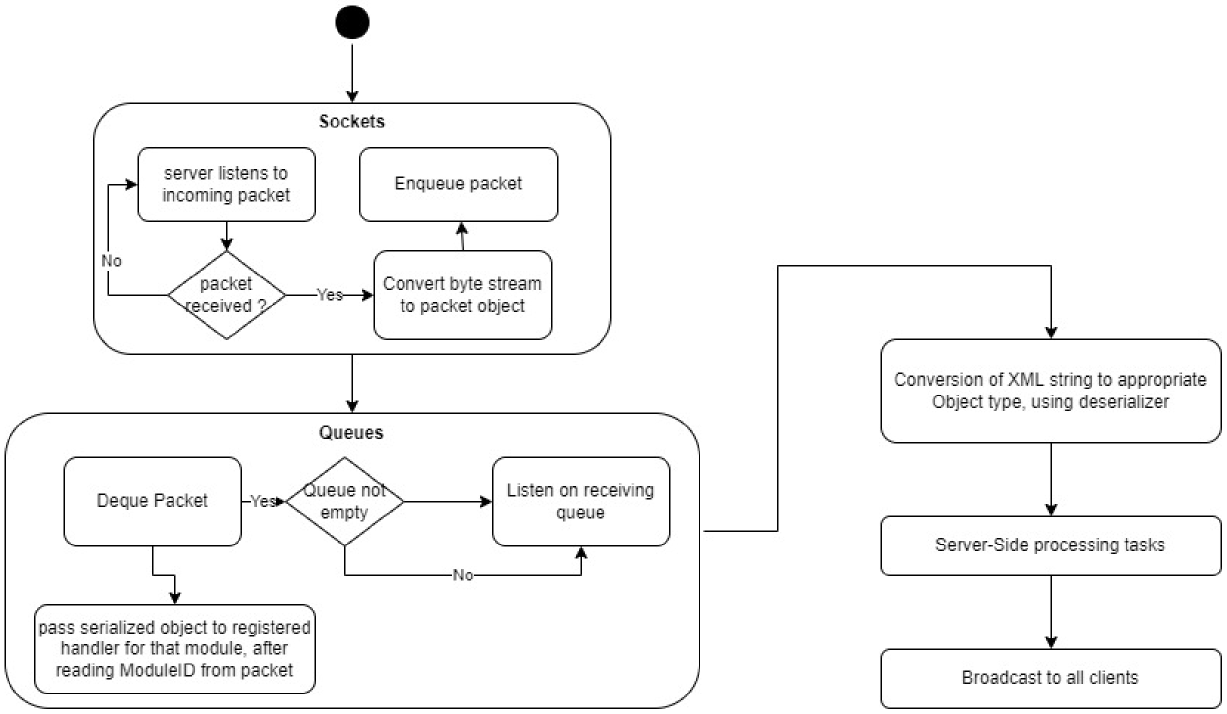


Activity Diagrams

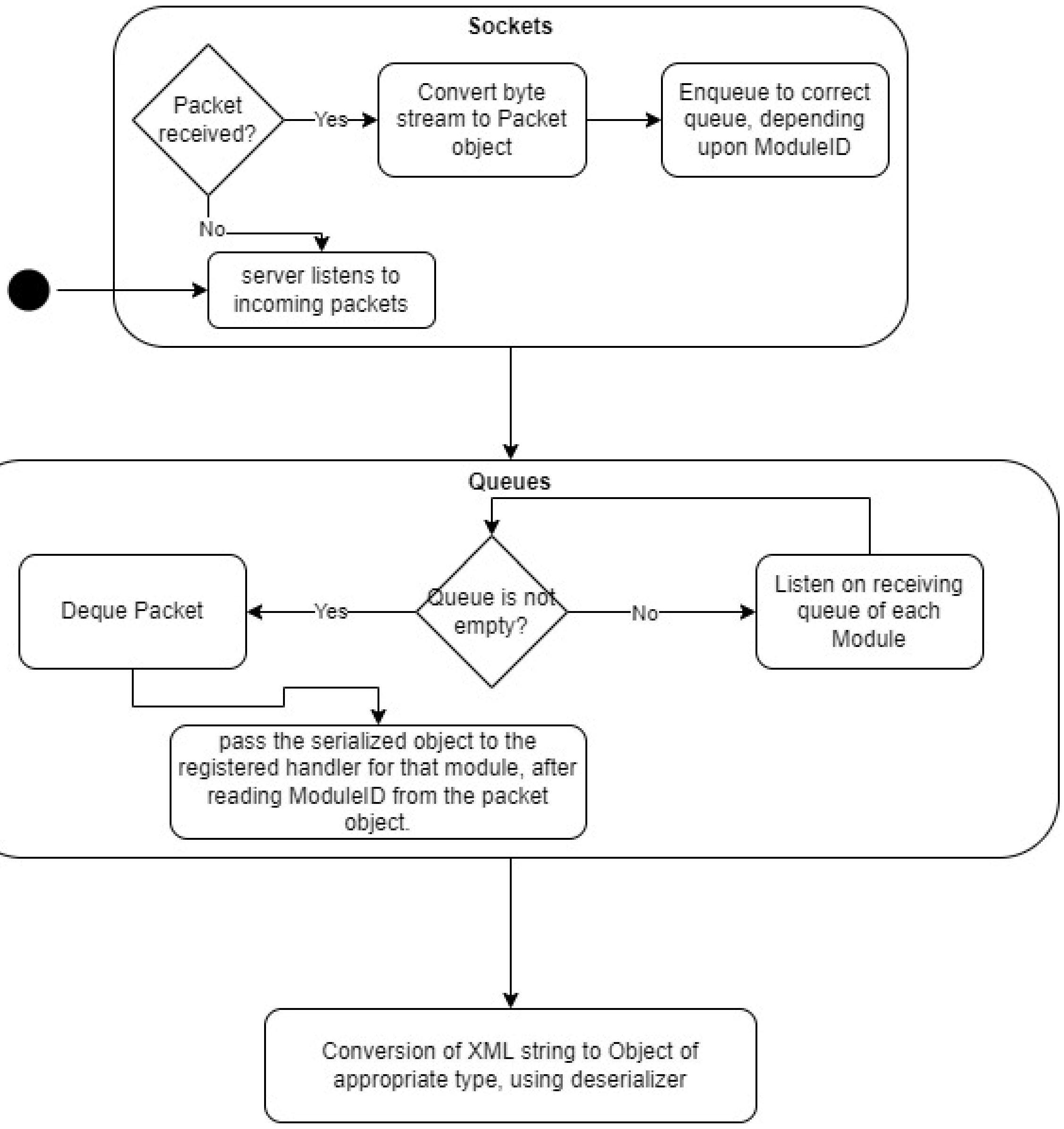
General Flow of Control


Input object serialized to XML Object
Queues
create Object package
Enqueue the package
Sockets
Queue is not empty, contains packet?
No
Listen on sending queue
Yes
Dequeue Packet
Packet object to byte stream conversion
send byte stream to destination(s)

Broadcasting packets to modules


Sockets
server listens to incoming packet
packet received ?
No
Yes
Convert byte stream to packet object
Enqueue packet
Queues
Deque Packet
Yes
Queue not empty
No
Listen on receiving queue
pass serialized object to registered handler for that module, after reading ModuleID from packet
Conversion of XML string to appropriate Object type, using deserializer
Server-Side processing tasks
Broadcast to all clients

Sending packets to a specific module


Sockets
Packet received?
Yes
Convert byte stream to Packet object
Enqueue to correct queue, depending upon ModuleID
No
server listens to incoming packets
Queues
Queue is not empty?
Yes
Deque Packet
No
Listen on receiving queue of each Module
pass the serialized object to the registered handler for that module, after reading ModuleID from the packet object.
Conversion of XML string to Object of appropriate type, using deserializer

Interfaces

Sending Data

For modules to send data, the object which is to be sent must be converted into a serialized string. For this, the modules must call the Serialize method implemented by the Serializer.

After the object is serialized, the module calls the Send method to send the data to the server.

Modules can also broadcast data, which ensures that the message is transmitted to every client (including the sender).

Receiving Data

Every module which is willing to engage in communication has to subscribe to the networking module by calling the Subscribe function.

When data is received in the received queue, the onDataReceived handler of the module is called with a serialized string.

The module then has to deserialize this message.