Distributed Chat – Part 2: Design

Anuraag Srivastava (as4378)

Md Ashiqul Amin (ma3359)

Madhav Poudel (mp2525)

Introductory Overview

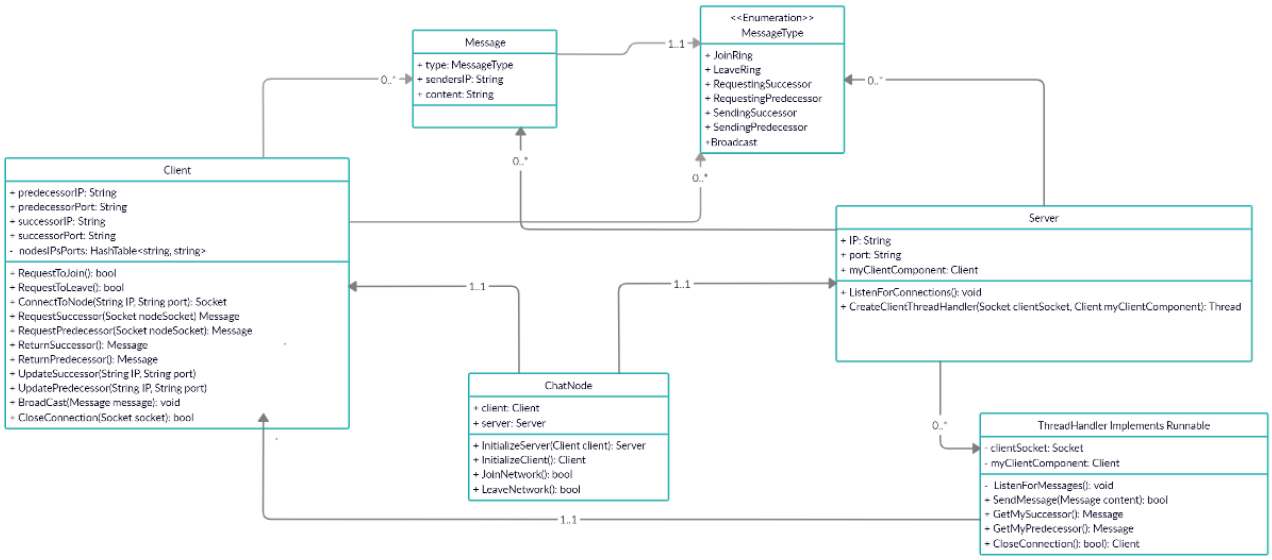
In the previous assignment of topology design, we came up with two approaches to implement the distributed chat:

1. Mesh
2. Ring

We have decided to move forward with implementing ring topology for this application. Token Ring networks have been used from a long time in LANs and WANs and are still being used which implement ring topology internally. Thus, learning and implementing ring topology would be a good learning experience for us.

In our topology we will be using volatile connections i.e. the connection between nodes is only establish when they want to communicate and is only active for one request/response cycle.

Class Diagram



Client:

This class contains attributes and methods for handling client component of the node.

Attributes:

1. **predecessorIP**: for storing IP address of predecessor
2. **predecessorPort**: for storing port number of predecessor
3. **successorIP**: for storing IP address of successor
4. **successorPort**: for storing port number of successor

Methods:

1. **RequestToJoin()**: This method will handle the process of getting into the ring as explained in the activity diagrams.
2. **RequestToLeave()**: This method will handle the process of leaving the ring as explained in the activity diagrams.
3. **ConnectToNode(String IP, String port)**: This method will be used to establish a connection to a node with given IP address and port number.
4. **RequestSuccessor(Socket nodeSocket):** This method is used to request successor from a node. It will take socket connection to the node as argument and thus can be used after the connection the node has been established.
5. **RequestPredecessor(Socket nodeSocket):** It is similar to RequestSuccessor method and gets the predecessor info.
6. **ReturnSuccessor():** This method returns the current node's successor IP address and port in a Message object
7. **ReturnPredecessor():** It is similar to ReturnSuccessor method and returns the current node's predecessor IP address and port in a Message object
8. **UpdateSuccessor(String IP, String port):** It updates the node's current successor IP address and port number with given IP address and port.
9. **UpdatePredecessor(String IP, String port)**: It is similar to UpdateSuccessor method and updates the predecessor info instead
10. **BroadCast(Message message)**: This method is used to forward the message to the node's successor in the ring.
11. **CloseConnection(Socket socket):** It closes the socket connection to a node and takes socket connection object as argument.

Server:

This class contains attributes and methods to handle server component of the node.

Attributes:

1. **IP:** to store the IP address of the node
2. **port:** to store the port number where the server will listen

3. **myClientComponent:** stores reference of Client object pointing to the client component of the node

Methods:

1. **ListenForConnections():** This method runs the server on the specified port and listens for incoming connections in an infinite loop.
2. **CreateClientThreadHandler(Socket clientSocket, Client myClientComponent):** This method will create a new thread of ThreadHandler class to handle the current client and pass the clientSocket for the connection. It also passes a reference to the client component which will help to send information stored in the client component to the current client.

ThreadHandler:

This class implements Runnable interface and contains attributes and methods for handling requests from client.

Attributes:

1. **clientSocket:** to store the reference of the socket connection to the client
2. **myClientComponent:** to store reference to the client component of the current node

Methods:

1. **ListenForMessages():** This method will create stream reader using the client's socket connection and will read messages from the client's input stream in an infinite loop for duration of the connection and take appropriate steps for each message type.
2. **SendMessage(Message content):** This method will send the given Message object to the client's output stream.
3. **GetMySuccessor():** This method will be useful to return the current node's successor info to the client. It will call the ReturnSuccessor method of the client component of the node.
4. **GetMyPredecessor():** This method is similar to GetMySuccessor but instead will return predecessor info.
5. **CloseConnection():** This method will close the input and output streams and terminate the connection with the client.

ChatNode:

This class is a representation of the chat node in the network and contains attributes and methods for initializing the client and server components of the node.

Attributes:

1. **client:** this holds reference to the Client object representing the client component of the node
2. **server:** this holds reference to the Sever object representing the server component of the node

Methods:

1. **InitializeClient():** This method creates an object of class Client which serves as the client component of the node.
2. **InitializeServer(Client client):** This method initializes the server component of the node by creating an instance of the Server class.
3. **JoinNetwork():** This method will call the RequestToJoin method of the client component to join the ring network
4. **LeaveNetwork():** This method will call the RequestToLeave method of the client component to leave the network.

Message:

This class contains the attributes which are required to send message in the ring.

Attributes:

1. **type:** This represents the message type and is of type MessageType enum.
2. **sendersIP:** This is used to record the sender's IP address and is useful for a node to identify itself as a sender of the message.
3. **content:** This contains the actual content of the message in string format.

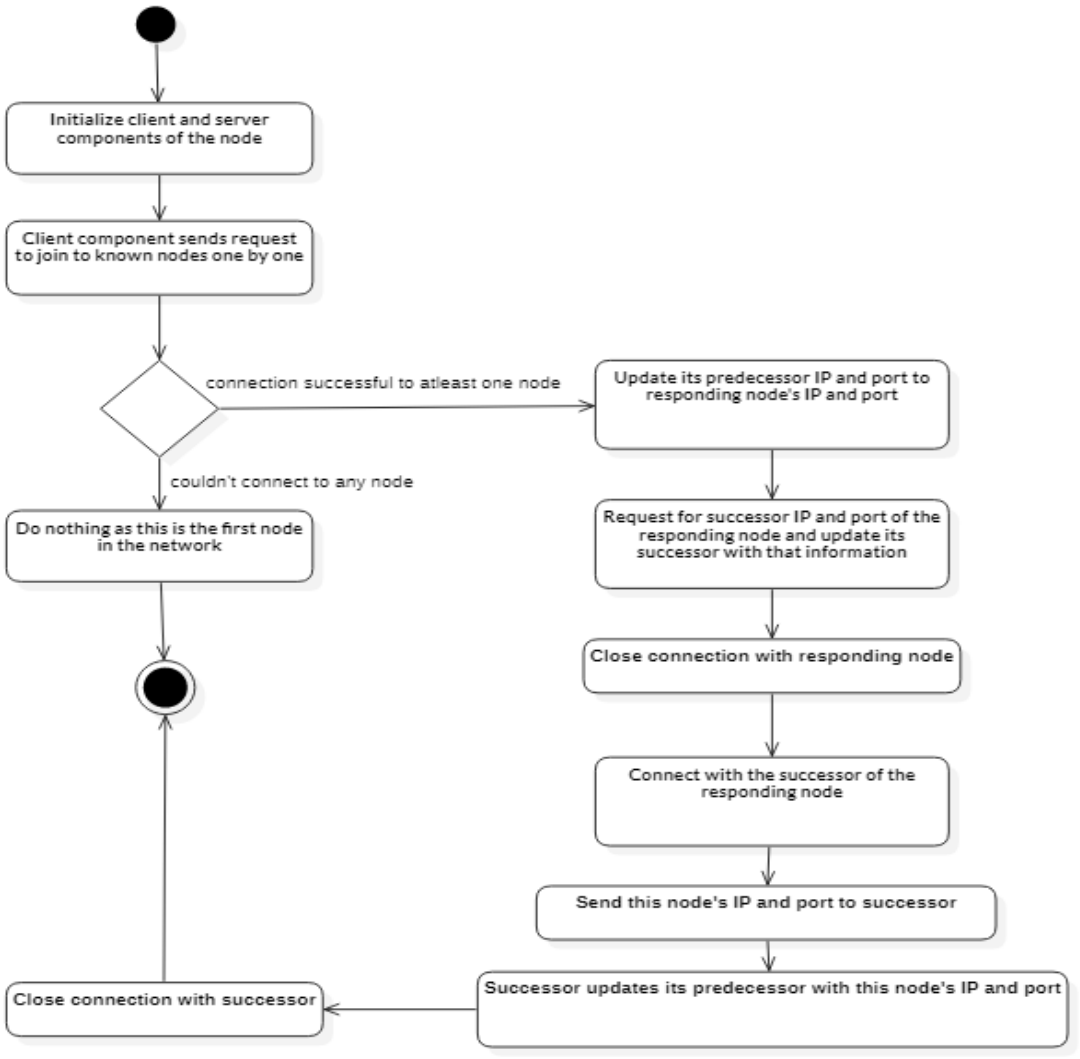
MessageType:

This is an enum for representing different types of messages that might get exchanged in the ring network

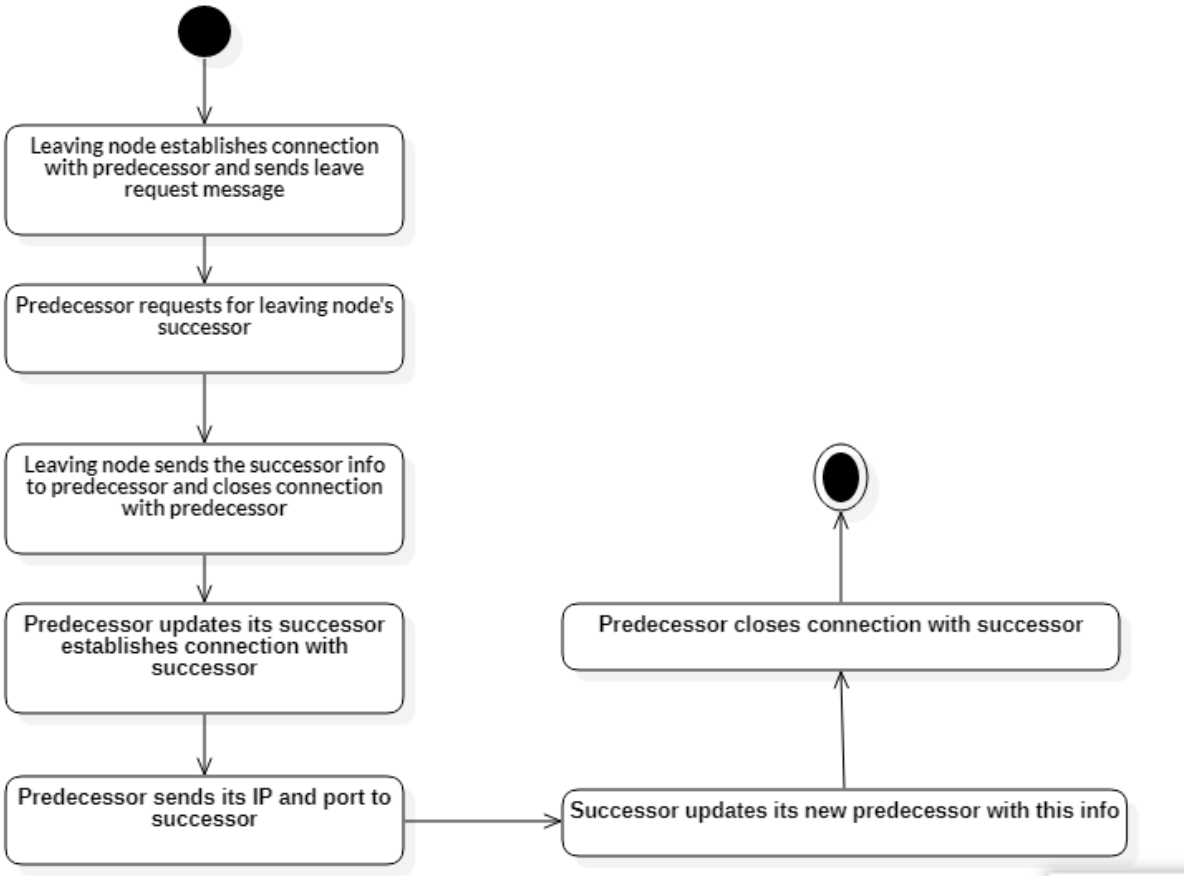
1. **JoinRing:** This type of message will be used when a new node wants to join the ring
2. **LeaveRing:** This type of message will be used when a node wants to leave the ring
3. **RequestingSuccessor:** This type of message will be used when a node is requesting successor information from a node
4. **RequestingPredecessor:** This type of message will be used when a node is requesting predecessor information from a node
5. **SendingSuccessor:** This type of message will be used when a node is sending its successor information to a node.
6. **SendingPredecessor:** This type of message will be used when a node is sending its predecessor information to a node.
7. **Broadcast:** This type of message will be used when a node wants to broadcast the message to all nodes in the ring

Activity Diagrams

1. Joining the ring

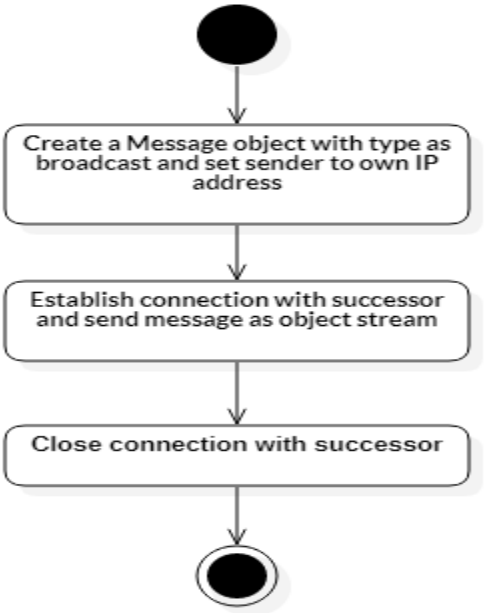


2. Leaving the ring

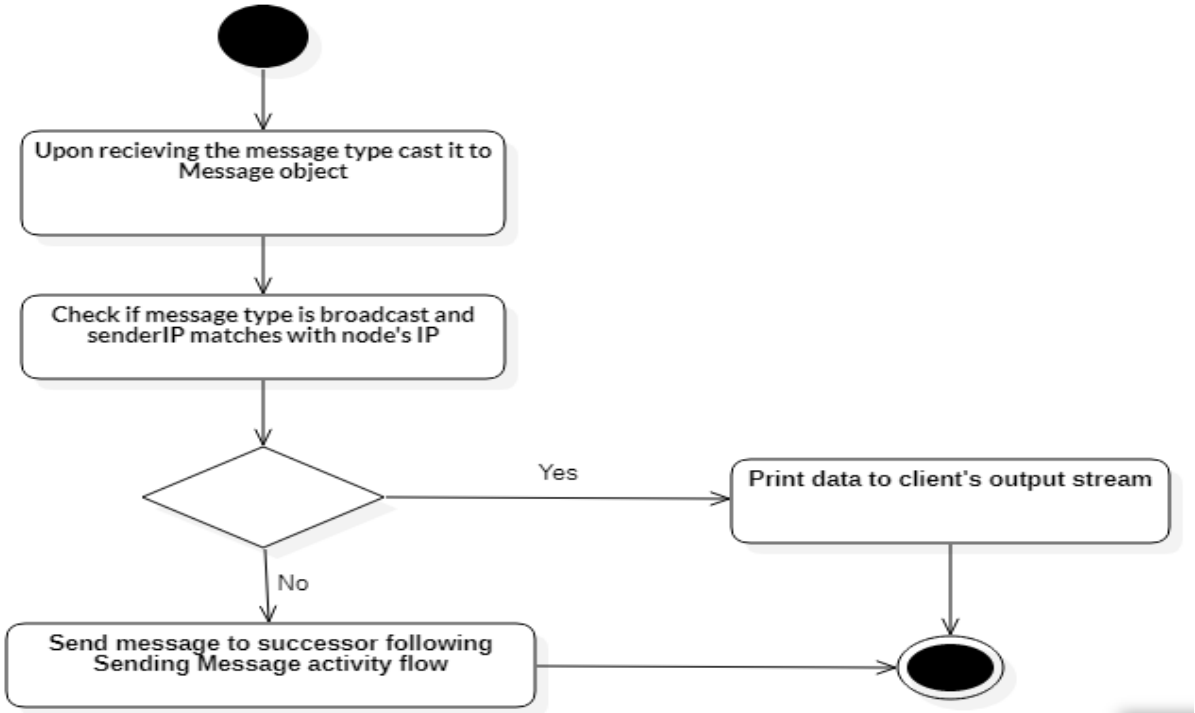


3. **Broadcasting message in a ring**

Sending:



Receiving:



Messaging Protocol

We have already described the broadcasting of messages in the ring in the activity diagram. For other messages the following protocol will be used:

1. If node **A** wants to communicate with node **B** then the client component of node **A** will establish a connection with server component of node **B**.
2. After the connection is established a Message object should be created with appropriate type and content.
3. For different message types we will have different message content formats. For instance, if message type is SendingSuccessor or SendingPredecessor then format of content string will be like "IP#Port". The receiving node will then have to split the string by '#' character and interpret first part as IP address and second part as the port number.
4. After creating the Message object, the sending node must serialize the object to ObjectOutputStream object in Java so that it can be transmitted to the output stream of the receiving node.
5. After the sending the message the client component of the node will close the connection with the server component of the other node.
6. If the server component is responding to a client's request, then it will follow steps 3 and 4 and then terminate the connection with the client component. Since we are using volatile connections for each request/response cycle a new connection would need to establish.