a. ServerNode Class

i)

import java.util.ArrayList;

import java.util.List;

public class ServerNode {

// List of connected client nodes

private List<ClientNode> connectedClients;

// Data values stored on the server (can be expanded as needed)

private String data1;

private String data2;

// Constructor

public ServerNode() {

this.connectedClients = new ArrayList<>();

this.data1 = null;

this.data2 = null;

}

// Method to add a client node

public void addClient(ClientNode client) {

this.connectedClients.add(client);

}

// Method to broker messages between clients

public void sendMessage(String message, ClientNode sender) {

// Don't send message back to sender

for (ClientNode client : connectedClients) {

if (client != sender) {

client.receiveMessage(message);

}

}

}

ii)

* This method takes a message string and the sending ClientNode as arguments.
* It iterates through the connectedClients list.
* For each client except the sender, it calls the receiveMessage method on the client, passing the message. This simulates sending the message to all connected clients except the sender.

b) ClientNode Class

i)

import java.util.ArrayList;

import java.util.List;

public class ClientNode {

private String id;

private ServerNode server;

public ClientNode(String id, ServerNode server) {

this.id = id;

this.server = server;

}

public void send(String message) {

// Delegate message sending to the server

this.server.sendMessage(message, this);

}

public void receiveMessage(String message) {

System.out.println("Client " + id + " received message: " + message);

}

}

ii)

* This method takes a message string as input.
* It delegates the message sending to the server's sendMessage method.
* It passes the message and itself (this) as arguments to the server method. This allows the server to identify the sender.

iii)

* This method takes a message string as input.
* It simply prints the received message along with the client's id.

c. Star Class

i)

import java.util.ArrayList;

import java.util.List;

public class Star {

private ServerNode serverNode;

public Star() {

this.serverNode = new ServerNode();

}

public void insertNode(String clientId) {

ClientNode clientNode = new ClientNode(clientId, serverNode);

serverNode.addClient(clientNode);

}

public void deleteNode(String clientId) {

// Find the client node to be deleted

ClientNode clientToDelete = null;

for (ClientNode client : serverNode.connectedClients) {

if (client.id.equals(clientId)) {

clientToDelete = client;

break;

}

}

// If client found, remove it from the server's list

if (clientToDelete != null) {

serverNode.connectedClients.remove(clientToDelete);

} else {

System.out.println("Client with ID " + clientId + " not found.");

}

}

}

ii)

* Takes a client ID as input.
* Creates a new ClientNode object with the given ID and a reference to the serverNode.
* Adds this new client to the server's connectedClients list.
* Takes a client ID as input.
* Iterates through the connectedClients list to find the client with the matching ID.
* If the client is found, it removes it from the list.
* If the client is not found, it prints a message indicating that the client with the given ID does not exist.

i)

**Suitable Compression Algorithm(s):**

In a star topology network with a central server mediating communication, a dictionary-based compression algorithm like LZ77 (Lempel-Ziv 77) could be well-suited for compressing messages sent between clients.

Because:

**Repetitive Data:** If messages between clients contain duplicates, such as common words or data patterns, LZ77 can use these for compression.

**Centralized Management:** The server can function as a central repository for the LZ77 dictionary. This dictionary can hold previously encountered substrings and their references. Clients can then refer to these references in their messages rather than transmitting the whole substring multiple times.

**Decompression:**

The server maintains the dictionary throughout the communication. When it receives a message with references, it uses the dictionary to replace those references with the actual substrings, reconstructing the original message.

**Worst-Case Time Complexity (d(i))**

The worst-case time complexity of LZ77 compression and decompression is O(n^2), where n is the length of the message.

* **Compression:** In the worst case, the entire message might consist of unique characters, requiring iterating through the entire message (n times) for each character to find matches in the dictionary (potentially another n comparisons).
* **Decompression:** Decompression involves looking up references in the dictionary. In the worst case, each character in the compressed message might be a reference, leading to n lookups in the dictionary, resulting in O(n) complexity.