Overlay Project

Distributed System

20th of April 2024

—

Submitted by Groupe K

PHAN Manh Tung

Vadym Vyhovskyi

MOSIG M1

INTRODUCTION

The project aims to create a simulated distributed network where nodes can communicate using RabbitMQ messaging protocol. Each node is assigned a unique identifier and positioned within a virtual ring topology. However, the routing decision within this virtual ring is influenced by the underlying physical network topology. This means that messages may traverse through intermediary nodes before reaching their intended destination in the virtual ring. The primary goal of the project is to develop an efficient routing algorithm that takes into account both the virtual ring structure and the physical network topology, ensuring optimal message delivery between nodes.

Github: https://github.com/phanmanhtung/Distributed-System-MoSIG-M1

RUN THE PROGRAM

Compile the code:

```
javac -cp .:amqp-client-5.16.0.jar:slf4j-api-1.7.36.jar:slf4j-simple-1.7.36.jar Sender.java OverlayNode.java Start.java
```

Run RabbitMQ server:

```
sudo rabbitmq-server
```

Then, open 5 terminals (5 nodes from 1 to 5):

```
java -cp .:amqp-client-5.16.0.jar:slf4j-api-1.7.36.jar:slf4j-simple-1.7.36.jar Start 1
```

Initiate a message with the format: Sender nodeId R/L “Message”. For example we want to send from node 1 in the right direction message “1R” as follows:

```
java -cp .:amqp-client-5.16.0.jar:slf4j-api-1.7.36.jar:slf4j-simple-1.7.36.jar Sender 1 R "1R"
```

STRUCTURE

• **Topology**: The project relies on two fundamental data structures to establish communication within the Overlay network. Firstly, the *Virtual Ring*, represented as an array of node IDs arranged in a circular order, dictates the path a message should take. Each node only interacts with its immediate left and right neighbors. Secondly, the *Physical Topology*, depicted by a two-dimensional adjacency matrix, defines the actual route that a message traverses from one node to another.

In the code, the Physical Topology is stored in a boolean matrix (*globalPhysicalTopology*), while the Virtual Topology is represented by arrays. Additionally, the nextHop array holds the next-hop routing decisions, enabling correct message forwarding based on the physical topology.

• **Nodes**: In the network, nodes operate based on the Physical Topology, which is represented by RabbitMQ queues serving as the connections between them. Each node is identified uniquely and keeps track of both the virtual and physical topologies. When activated by senders, nodes receive messages and pass them along to the next node using the appropriate RabbitMQ queue, following predetermined guidelines (left or right).

• **Sender**: The sender serves as the initiator of communication within the network. By triggering a specific node, the sender prompts it to send a message either to its "left" or "right" neighbor. The content of the message is provided by the user through the sender interface.

DIAGRAMS

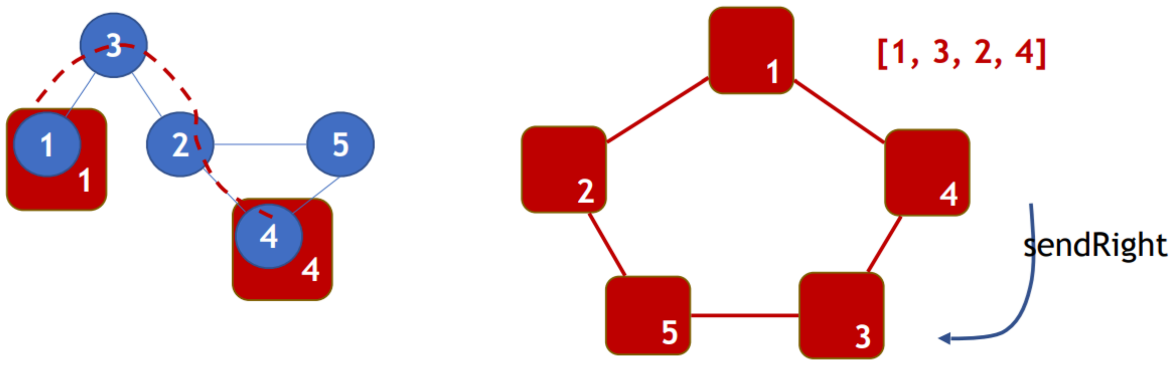
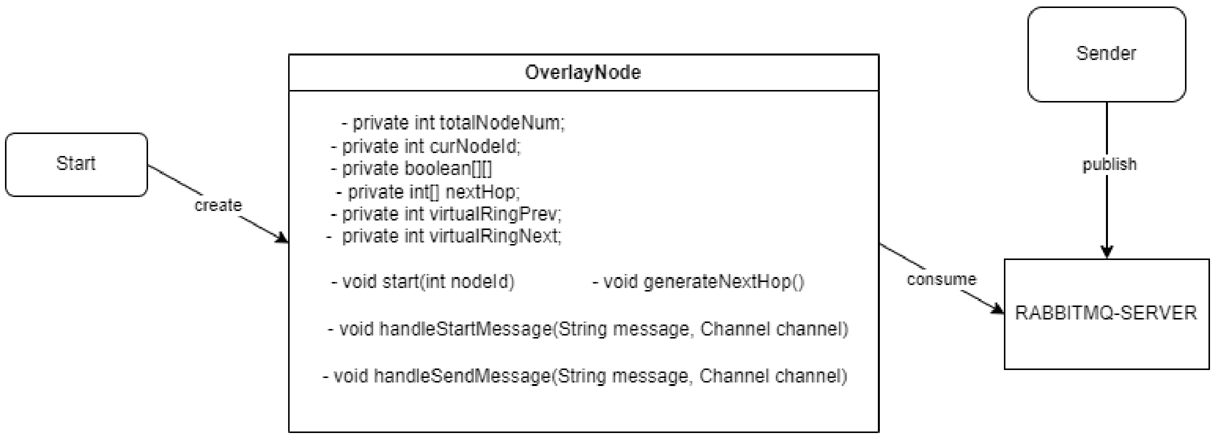


Figure 1: Physical Topology and Virtual Ring



OverlayNode
- private int totalNodeNum;
- private int curNodeId;
- private boolean[][]
- private int[] nextHop;
- private int virtualRingPrev;
- private int virtualRingNext;
- void start(int nodeId)
- void generateNextHop()
- void handleStartMessage(String message, Channel channel)
- void handleSendMessage(String message, Channel channel)
Start
create
Sender
publish
consume
RABBITMQ-SERVER

Figure 2: Overlay Object Diagram

ALGORITHM (Breadth-First Search)

The *generateNextHop* method employs a breadth-first search algorithm to calculate the shortest paths between nodes in the Overlay network. Initially, it initializes arrays *prev* and *distance* to track the previous nodes and distances from the current node, respectively. These arrays are implemented using the integer data type. Utilizing a queue data structure, represented by *LinkedList<Integer> queue*, the algorithm explores the network starting from the current node. At each iteration, it dequeues the current node from the queue and examines its neighboring nodes. If a link exists to a neighbor and a shorter path is discovered, the algorithm updates the distance and previous node information. This process continues until all reachable nodes are explored, ensuring an exhaustive search across the network. Finally, as the output of the process, the algorithm updates the *nextHop* array based on the computed shortest paths.

For example, the nextHop array of node 4:

nextHop = [1, 1, 1, 3, 4]

- nextHop[0] = 1 => To reach node 0, the next-hop node is node 1.
- nextHop[1] = 1 => To reach node 1, the next-hop node is node 1.
- nextHop[2] = 1 => To reach node 2, the next-hop node is node 1.
- nextHop[3] = 3 => To reach node 3, the next-hop node is node 3.
- nextHop[4] = 4 => To reach node 4, the next-hop node is node 4.

MESSAGE HANDLING

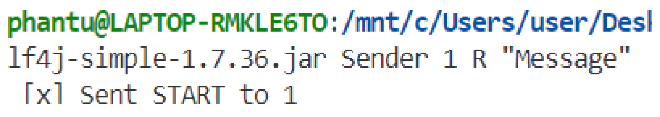
When a message is initiated, it first arrives at the initial node in the network. If the message is a "START" type (from the sender, one time at the start of the process), the node examines the request to determine whether it should be sent left or right within the virtual ring. Subsequently, the message is changed into the “SEND” type and forwarded to the next node in the specified direction.

If the message is of type "SEND," the node verifies if it represents the final destination. If not, the node calculates the next node in the physical ring and forwards the newly constructed SEND message to that next node. If yes, the system prints out the reaching destination message and terminates the process.

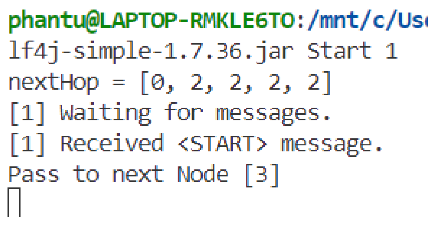
One note to point out is that any printing regarding the nodeId will plus 1 for the human-readable numbering purpose, starting from 1 instead of 0. This adjustment is made for clarity in output messages using 1-based indexing.

EXECUTION SCENARIO

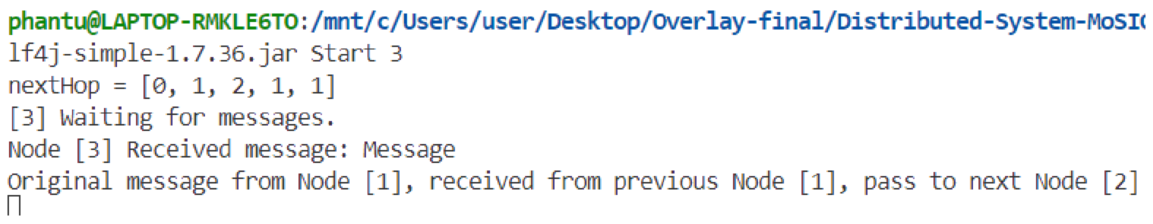
To illustrate our code in action, we send a message from node 1 to the right. First node 1 should receive a “START” notification and the message will traverse through node 3, then node 2 before reaching the destination at node 4.



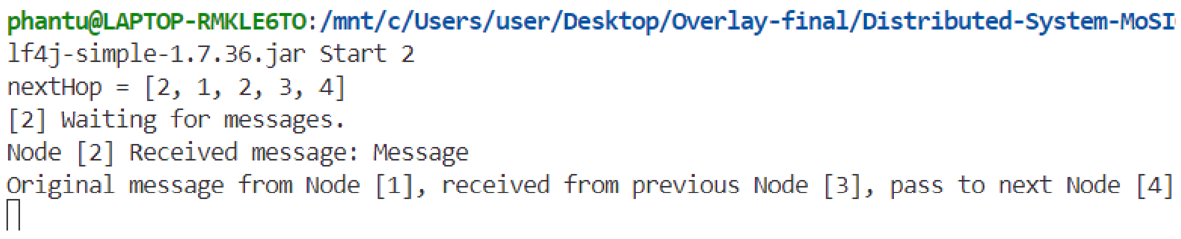
```
phantu@LAPTOP-RMKLE6TO:/mnt/c/Users/user/Desl
lf4j-simple-1.7.36.jar Sender 1 R "Message"
 [x] Sent START to 1
```



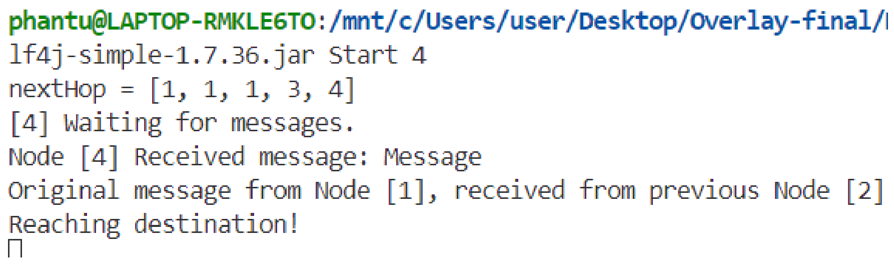
```
phantu@LAPTOP-RMKLE6TO:/mnt/c/Us
lf4j-simple-1.7.36.jar Start 1
nextHop = [0, 2, 2, 2, 2]
[1] Waiting for messages.
[1] Received <START> message.
Pass to next Node [3]
```



```
phantu@LAPTOP-RMKLE6TO:/mnt/c/Users/user/Desktop/Overlay-final/Distributed-System-MoSI
lf4j-simple-1.7.36.jar Start 3
nextHop = [0, 1, 2, 1, 1]
[3] Waiting for messages.
Node [3] Received message: Message
Original message from Node [1], received from previous Node [1], pass to next Node [2]
```



```
phantu@LAPTOP-RMKLE6TO:/mnt/c/Users/user/Desktop/Overlay-final/Distributed-System-MoSI
lf4j-simple-1.7.36.jar Start 2
nextHop = [2, 1, 2, 3, 4]
[2] Waiting for messages.
Node [2] Received message: Message
Original message from Node [1], received from previous Node [3], pass to next Node [4]
```



```
phantu@LAPTOP-RMKLE6TO:/mnt/c/Users/user/Desktop/Overlay-final/
lf4j-simple-1.7.36.jar Start 4
nextHop = [1, 1, 1, 3, 4]
[4] Waiting for messages.
Node [4] Received message: Message
Original message from Node [1], received from previous Node [2]
Reaching destination!
```

Figure 3: Terminal captures