IDS Project: Overlay

Iskander Gaba
iskander.gaba@etu.univ-grenoble-alpes.fr

Kseniia Masalygina
kseniia.masalygina@grenoble-inp.org

May 4, 2020

1 Introduction

The goal of the project is to construct a virtual network (overlay) over a physical one (underlay). We implemented the following features:

- Parsing an adjacency matrix and checking the input graph for connectivity.
- Creation of RabbitMQ nodes.
- Finding the shortest paths between all nodes and the creation of a routing table for each node.
- Creation of ring topology.
- Sending and receiving messages with RabbitMQ directly over the physical topology.
- Message exchange between virtual nodes.

2 Architecture

The overall structure of the project is presented in Figure 1. We built an overlay network on top of a physical network, where nodes of the original network are connected in a ring.

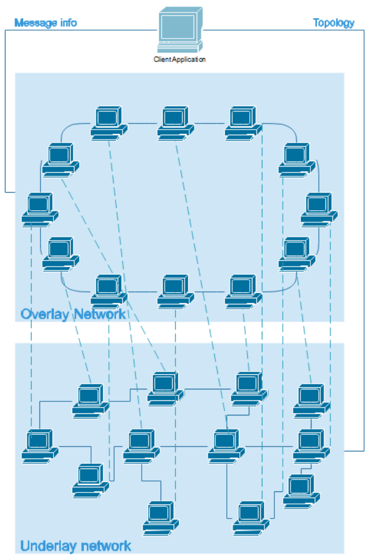

Figure 1: Architecture of an overlay

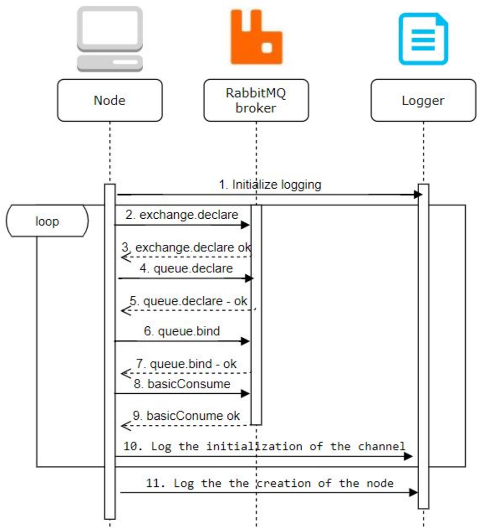

Figure 2: Creation of physical nodes

We divided the explanation of the structure architecture into two parts: creation of nodes and message exchange.

Creation of underlay nodes is shown in Figure 2. Here we have three entities: physical node, RabbitMQ docker instance and logger.

- Physical nodes get routing tables and exchange map from processed input graph. To get routing tables with shortest paths we used Breadth First Search as it always finds optimal solutions. We worked under assumption that all nodes are reachable. Exchange channel names are obtained from the initial graph.

- RabbitMQ was used as it is more robust and flexible. Working with it means that we don't have to worry about centering node being down. Nodes communicate with RabbitMQ broker on the underlay level.

- We use a logger within each node to track the movement of messages. This mimics real life. Administrator of the network should be able to see the data flow in the network.

Message exchange between sender and receiver is shown in Figure 3.

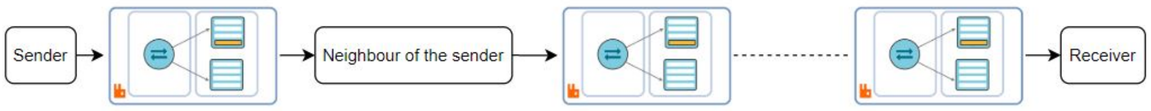

Figure 3: Message exchange between two nodes

Direct message exchange between two underlay neighbours is shown in Figure 4.

- Each physical node works with RabbitMQ entities: queues, bindings and exchanges. The message gets published to an exchange, which distributes it to a queue using bindings. For each edge in the graph, there is an exchange channel to which only the linked nodes by that edge are connected to. For example, if we have nodes 1 and 4 that are connected by an edge in the input graph, we would have a unique channel exchange name `1-4` such that only 1 and 4 are connected to that channel. The naming convention followed here is starting with the id of the node that has minimum value among both nodes (i.e. `1-4` but not `4-1`).

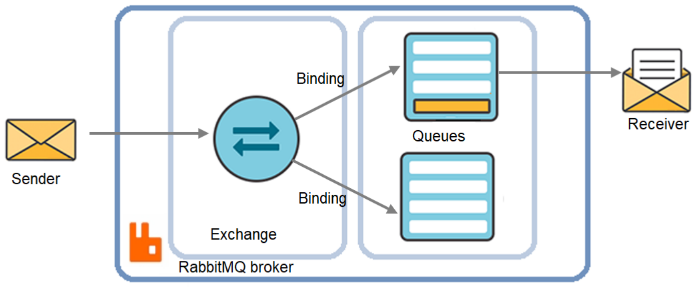

Figure 4: Message exchange between two neighbouring nodes

- We decided not to use a single queue to which all the nodes are connected. Even with some rules that restrict communication between nodes, the nodes would be connected. We wanted to have nodes without direct connection (i.e. an edge in the physical topology connecting them) to not have a link.
- The virtual layer is built on top of the physical level. We get the ring topology by applying nearest neighbour algorithm. We decided to use this algorithm because it is simple to implement, the time complexity is $O(n^2)$ in the worst-case scenario, and the resulting ring is, in most cases, within 25% of the optimal ring in terms of average distance between virtual neighbors measured by the number of hops. Ring topology defines the limitations of our program. Severing one node of the topology leads to malfunctions in the entire network. Sending methods on the virtual level use the sending method from the physical level.

3 Implementation

3.1 Code structure

- Physical node: `Node.java`
  A physical node represents an actual node on the underlay. Physical nodes are created from the topology description given to the main program. Each node is instantiated with an id, an exchange map and a routing table. Each physical node object has the following internal state:
  1. `id`: A unique integer representing the node.
  2. `exchangeMap`: A map in which the keys are ids of the neighboring nodes and the values are unique `DIRECT` RabbitMQ exchange names that each correspond to an edge connected to `this` node. This is used so that each physical link (i.e. edge) is represented by a RabbitMQ exchange so that only the two connected nodes by that link can use it to communicate. When sending a message to the neighboring node (i.e. the next hop), `this` node can look up the exchange name to use (i.e. the edge linking it to the specific neighboring node) in order to send or forward a message.
  3. `routingTable`: A map in which keys are target node and values are the next hop to be taken. This is to be used so that when a node is sending a message to another node, it needs not to know the full shortest path. It simply checks in its routing table what is the next hop to be taken (i.e. the next neighboring node the message should be forwarded to) in order to arrive for it to arrive to destination.

When the node is created, it creates an exchange (or gets a reference to it if it is already created) to each of its neighbors with the corresponding exchange name in the `exchangeMap`, creates a queue and binds itself to it with its id as a routing key. This way, `this` node has a one queue per each `DIRECT` connection (i.e. edge on the graph) with a physically neighboring node and is listening for messages from each queue using a callback that will either forward the message to the next hop or accept it if `this` node is the final target. Finally, `Node.java` has the following two primitives:

1. `void send(Message message)`: Takes a `Message` object, reads the target node, queries the next hop (a neighboring node) from the routing table, queries the exchange name with the next hop from the exchange map, and sends the message to the next hop using the corresponding unique exchange name they share.
2. `DeliverCallback receiveCallback`: This is the callback that gets triggered whenever a node receives a message from a neighboring node. If the target node is `this` node, the node logs the message. Otherwise, the node calls `send(Message message)` explained above to forward the message to the next hop.

- Virtual node: `VritualNode.java`
  A virtual node is a wrapper on the top of `Node` that has the following state:

  1. `node`: A reference object of the physical node (`Node` object) corresponding to `this` virtual node. Note that it is fine for a virtual node to have a reference of its respective physical node in a distributed setting (since the virtual node is basically a wrapper existing on the same machine as the physical node). It is out of the question however for a virtual node to have references to physical nodes other than the one it wraps, which is exactly what our implementation avoids.
  2. `leftNeighbor`: ID of the left neighboring virtual node on the ring.
  3. `rightNeighbor`: ID of the right neighboring virtual node on the ring.

  A virtual node comes with the following two primitives:

  1. `sendLeft()`: Calls `node.send()` passing it a message with the target node being `leftNeighbor`.
  2. `sendRight()`: Calls `node.send()` passing it a message with the target node being `rightNeighbor`.

- Message: `Message.java`
  A container class for the actual message and its meta-data. It has the following fields:

  1. `id`: Unique identifier for the message.
  2. `src`: ID of the node that sends the message.
  3. `dest`: ID of the recipient-node.
  4. `message`: String containing the message.

  Methods implemented in this class have explanatory names.

- Logger: `Logger.java`
  A class instantiated inside each physical node that logs to a shared file between all nodes - `comms.log`. Logs the creation of the node, passage of the message and final hop of the delivery.

- Main program: `Main.java`

  1. `boolean isConnected(boolean[][] graph, boolean[] marked, int node)`: The method checks if the topology `graph` is connected using Depth-first-search for the graph traversal. This recursive algorithm starts at node with index `node` as a root and exhaustively searches for unvisited neighbors of this node using backtracking.
  2. `List<Map<String, String>> getExchangeMaps(boolean[][] graph)`: The method gets the list of exchange channel maps for nodes from `graph`. For each node a `HashMap` is created. If there is an edge between two nodes in the `graph` then `exchangeId` is added to the `HashMap`-s of these nodes. The connection `exchangeId` is presented as a string created from the indices of the nodes and a hyphen in between.
  3. `List<List<Integer>> getShortestPaths(boolean[][] graph, int node)`: It returns the shortest paths between node with index `node` and all other nodes. It takes two arguments: `graph` (initial graph) and `node` (index of the starting node). We apply Breadth-First-Search to find the shortest paths. After that, we traverse all found paths and add them to the final list.
  4. `List<Map<String, String>> getRoutingTables(boolean[][] graph)`: This method uses the shortest paths from each node to every other node obtained from `getShortestPaths()` to build a routing table for each physical node to be used when instantiating it.
  5. `List<Integer> tour(boolean graph[][], int nnodes, int start)`: This method constructs ring topology from a given `graph` using nearest neighbor algorithm. In the beginning, all nodes are initialized as unvisited. Node with index `start` is marked as visited. Next step is to find the closest unvisited neighbor. The found node becomes the current node and is marked as visited. Algorithm is repeated until all nodes are visited. Method returns list `ring` with correct topology.
  6. `printRing(List<Integer> ring)`: The method is used to print ring topology.

3.2 General Implementation

The Main program reads a topology description file (i.e. a file containing the number of nodes on one line followed by the adjacency matrix that describes the topology of the underlay) and checks if the topology represents a connected graph. If it does not, the program will exit with an error stating that the underlay topology is not a connected graph and therefore no overlay ring can be formed on the top of it.
After that, the program works with the graph. It creates unique exchange channel names between each two neighboring physical nodes and a routing table for each physical node. The physical nodes are then instantiated each with its own id, exchange map, and routing table. After that, overlay topology is computed in the form of a ring. Finally, lists of nodes for both levels are created.
The part of the program that interacts with the user trough the terminal includes multiple options:

- Send message on the virtual level to the left neighbor. This option will call `sendLeft` method from `VirtualNode.java`.

- Send message on the virtual level to the right neighbor. This option will call `sendRight` method from `VirtualNode.java`.
- Send message on the physical level directly. This will call `send` method from `Node.java`.
- Print ring topology. It will call `printRing` method.
- Exit. This will terminate the program

4 Deployment

In this project, we used docker (Docker v19 or higher) RabbitMQ image. To work with our program, a container has to start in the background running an instance of RabbitMQ based on the official image. The project is written in Java so OpenJDK v11 (or higher) is required to run the program. We have grouped everything into a `Makefile` for convenience. The following commands are for compiling the project into one JAR file:

```
cd src
make
```

Listing 1: Compilation

To start the program run:

```
cd ../bin
java -cp Main.jar:../lib/* Main topology-connected
```

Listing 2: Launching

To clean the binaries and class files, run:

```
cd ../src
make clean
```

Listing 3: Cleaning

It is possible to change physical topology. The topology description `topology-connected` file is stored in `bin` directory and can be changed to represent any underlay graph you want. The format of the file is the following:

```
6
1 1 1 0 0 0
1 1 0 0 0 0
1 0 1 1 1 0
0 0 1 1 0 1
0 0 1 0 1 1
0 0 0 1 1 1
```

Listing 4: Underlay Topology Description File

The first line represents the number of nodes and the rest of lines represent an adjacency matrix describing the underlay topology.
Interaction between the user and the program happens in the terminal. The user can choose the action that he or she wants to do. All activity is recorded in `comms.log`.
We didn't test the project on `manedlbrot` server as we do not have user rights to install RabbitMQ. However, we worked on two different machines so mobility should not be a problem.

5 Conclusion

The original goal of the project was reached. We've spent some time on improving the first version of the code. It was difficult to create an architecture that is close to real life. We had no way of testing distribution on multiple machines. Nonetheless, we were able to create a distributed system.

The completion of this project has strengthened our knowledge of RabbitMQ and distributed systems in general. Creating distributed programs that are asynchronous in nature and geographically independent with little to no shared physical resources proved to be quite interesting and challenging to reason about. We believe we met all the requirements of the project and we think that this project has a solid implementation. We are satisfied with our results.