INTRODUCTION TO DISTRIBUTED SYSTEMS FINAL PROJECT: OVERLAY

MOSIG 2019-2020

FERNÁNDEZ ARRIZABALAGA, IÑIGO
BALNAVES, CHRISTOPHER

INDEX

ARCHITECTURE

Design

When designing an overlay, the key principle of the system architecture is abstraction. The user should be able to work with the system interface without knowledge of the underlying node or code structure; the interface should be simple and intuitive to use. This interface is the Overlay class. Overlay primarily allows the user to control the virtual topology and send messages through that topology. To ensure the Overlay class is abstracted from the underlying nodes, there is no way for Overlay to communicate directly with the Nodes; instead, Overlay can only communicate with class Node_Registry.

Node_Registry acts as the communication layer between the interface and the underlying node structure; commands from the user are passed to Node_Registry from Overlay and then Node_Registry interprets the commands and either executes them itself or instructs the corresponding nodes to perform the task. Node_Registry can communicate with all members of the system and holds the full knowledge of the physical and virtual topologies, which the nodes do not and Overlay only learns if requested by the user.

Although it is a slight breach in the concept of abstraction for Overlay to learn the physical topology, it is necessary for generating a visual representation of the topology on a distributed system where Overlay and Node_Registry may be run on different machines. In other words, Node_Registry cannot generate the image as the user may be running Overlay on a different machine and cannot see the Node_Registry outputs.

The Node class implements a node in the physical topology. A Node knows by default how to communicate with Node_Registry and can be informed by Node_Registry on how to communicate with another Node. On initialisation into the topology, the Node learns its connections from Node_Registry; however, it does not know the full topology.

Figure 1 shows an example of the system architecture with three nodes. Each object has a send and receive channel, with the receive channel being bound to a single queue in each case, as indicated in the Figure. The send channel connections are indicated by the arrows, with the solid lines indicating the connection is permanent (made by default) and the dashed lines indicating the connection is non-permanent (made only when a message needs to be passed to that node).

The specific design choice was made to not use exchanges, despite this feature being common in RabbitMQ designs. An earlier version of this project implemented a routing exchange at the Node_Registry level; however, it was felt that this blurred the lines of communication separating Overlay from the Nodes and created doubt over the routing of messages via the physical topological connections. Therefore, the decision was made to use solely queues, to ensure the separation of Overlay from the Nodes and clearly display the routing of messages through the physical Node network.

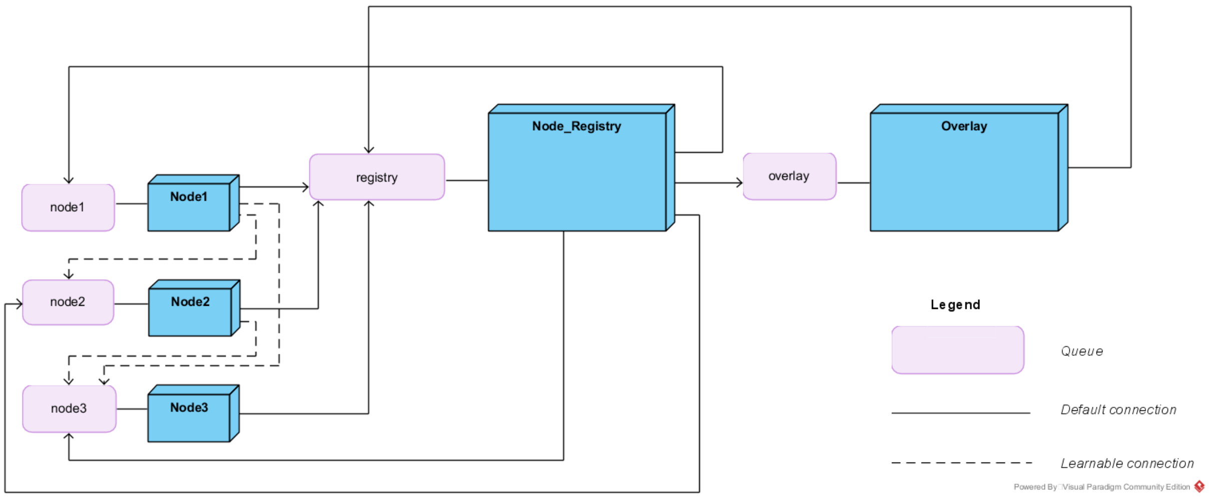


Figure 1: System Architecture

Assumptions and Limitations

It is assumed that the overlay design is a bi-directional ring. The system will not allow a node to have more than two virtual connections; however, there is no checking that the virtual topology forms the assumed ring.

It is assumed that only one Overlay is being used. The effect of two Overlays running concurrently is unknown; however, as only erroneous commands are reported back to the Overlay (eg. trying to make three connections to a node), there would have to be out-of-system communication between the two Overlay users to prevent confusion and possible errors.

It is also assumed that physical topology will not be changed. The physical topology is set in a separate file from the rest of the system and loaded into Node_Registry through a hard-coded command. As such, it is not possible for an Overlay user to specify the physical topology. To change the physical topology requires specifying the connection matrix in PhysicalTopology (if not already present) and changing the call in Node_Registry to match that topology. This is a limitation of the system but is also to some extent by design as the Overlay user should not be able to control the physical topology, only the virtual one.

The use of a single Node_Registry was chosen for simplicity, although there is the understanding that such a set-up creates weaknesses in the design. The weaknesses have been considered, though, and where possible, have been mitigated or reduced, for example:

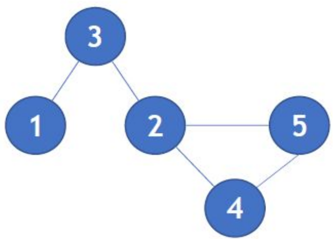
- The single recv queue acts as a bottle-neck point, as messages are received and implemented sequentially; however, after all the nodes have been initialised, only the single Overlay is sending messages to that queue so this should not pose a major problem.
- A failure in Node_Registry will affect the entire system. While the queues are durable and will survive a system restart, the virtual topology will be lost and will have to be re-initialised.

IMPLEMENTATION

Topological Representation

The topologies, both physical and virtual, are represented in the system by 2 dimensional square arrays, where the length is equal to the number of nodes. The connections in the topology are represented in the array by 1's. For example, the following 2D array corresponds to the adjacent topology:

```
topology ={{0, 0, 1, 0, 0},
           {0, 0, 1, 1, 1},
           {1, 1, 0, 0, 0},
           {0, 1, 0, 0, 1},
           {0, 1, 0, 1, 0}};
```



Node ID

All the nodes registered have the following structure:

nodeX

where X is a number from 1 to number of nodes the physical topology has.

Connections/Route Table

A NodeX after being registered is going to request to Node_Registry its connections with the rest of the nodes or its route table. The Node_Registry calculates the connections or route table by calculating the shortest path from the NodeX to the rest of the nodes using the Dijkstra algorithm.

The Dijkstra algorithm is a greedy algorithm that calculates the shortest path from source NodeX to the rest of the nodes. A shortest-path tree is generated with the given source as the root. Two sets are maintained; one set contains vertices included in the shortest-path tree, while the other set includes vertices not yet included in the shortest-path tree. At every step of the algorithm, a vertex is found which is in the other set (set of not yet included) and has a minimum distance from the source.

This route table is an array whose length is equal to the number of nodes of the physical topology. Each element of the array can be either:

- -1; If the position of the array is equal to X.
- [1,number of nodes]; in the remaining cases.

The array is interpreted such that each index of the array corresponds to the id of the next node that a message destined for Node of id equal to the index must be sent to. In other words, for a message being sent to node1, the node with the message will look at the 1st index and send the message to the node specified.

AMQP Properties

In the project it has been necessary to send different pieces of information or attributes in addition to the message. In order to achieve this, the AMPQ BasicProperties call has been utilised, specifically the Application ID field (AppID) and the Headers field. The AppID contains the command that needs to be executed, such as “send” (see the example immediately below), and the headers is composed of a String to String HashMap and stores nodeID information, such as “srcNode”.

For example, if Overlay instructs Node_Registry to send a message from nodeX to nodeY, it is necessary for the message between them to have four components: the source Node, the destination Node, the send instruction and the message itself. The message is sent in the body of the basicPublish() function, the source and destination nodes are stored in the headers and the send command in the AppID field of the AMQP properties.

Image Generation

A feature of the system is that it can generate a visual representation of the physical and virtual topologies. This is implemented through the CreateImage and DrawImage classes, which are used by Overlay. When passed the 2D topology array, these classes generate a simple diagram displaying the nodes and connections, then display it in a separate window. In the case of the virtual topology, each call to *show_topology_overlay* (see below) generates a new file; hence it is possible to track the changes made to the virtual topology by viewing these images. The files are deleted on conclusion of the program.

Commands

The system has the following commands in Overlay:

A. *list* : To obtain a list of the nodes initialized/registered.
B. *connect [nodeX] [nodeY]* : To connect nodeX to nodeY in the logical layer or virtual ring.
C. *disconnect [nodeX] [nodeY]* : To disconnect nodeX and nodeY in the logical layer or virtual ring.
D. *show_topology* : To get an image of the physical topology.
E. *show_topology_overlay* : To get an image of the logical layer or virtual ring.
F. *send [nodeX] [nodeY] [message]* : To send a message from nodeX to nodeY.
G. *send_left [nodeX] [message]* : To send a message to the left side of nodeX.
H. *send_right [nodeX] [message]* : To send a message to the right side of nodeX.
I. *help* : To see all the commands available and its description.
J. *exit* : To exit the program.

Examples

Presented below are two examples demonstrating how the implementation is used in key parts of the system function: the connection of a node to the system and the sending of a message.

The connection of a node to the system executes as follows:

1. Node is run on a machine. It creates a generic receive queue and a send channel connected to Node_Registry's receive queue.
2. Node sends an empty message to Node_Registry requesting to be registered.
3. Node_Registry receives the message and assigns an id. It creates a send channel connection to the node via that id and replies to node with that id via the specified replyTo queue.
4. Node receives the id and creates a receive queue based on that id.
5. Node contacts Node_Registry and requests its connections in the virtual topology.
6. Node_Registry calculates the connections using the Dijkstra algorithm and returns an array of connections to Node.
7. Node is now connected and ready to receive and transmit messages.

The sending of a message, either sendLeft or sendRight, executes as follows:

1. The user specifies the source node and inputs the message.
2. Overlay builds an AMPQ property with the source node and send command and sends that and the message to Node_Registry
3. Node_Registry receives the message, decodes the send command and source node then determines the physical destination node based on those factors.
4. Node_Registry builds a new AMPQ property with the destination node and sends that and the message to the source node.
5. The source Node will receive the message and check if it is the destination node. If it is, it prints the message. If not, it uses the destination node id as the index for its connections array and finds the Node to which the message must be sent to continue its journey.
6. Step 5 repeats with each subsequent Node until the destination Node is reached.

DEPLOYMENT

To deploy the project there are two possible options; one is to compile all the necessary JAVA classes and run the system (Compile&Run) or execute the JAR files.

Compile&Run

To Compile&Run, complete the following steps:

1) Set the classpath to make it easier for the Compilation and Execution.
    a) For Linux:
    ```
    export CP=.:amqp-client-5.7.1.jar:slf4j-api-1.7.26.jar:slf4j-simple-1.7.26.jar
    ```
    b) For Windows:
    ```
    set CP=.;amqp-client-5.7.1.jar;slf4j-api-1.7.26.jar;slf4j-simple-1.7.26.jar
    ```

The Linux commands have not been tested, with both test computers running Windows; however, it is assumed they do as they are copied from the RabbitMQ Tutorial.
It is important to correctly set the classpath as the packages are necessary libraries to use RabbitMQ; without them, there will be errors in the compilation or execution.

2) To compile the project:
    a) Compile the packages Dijkstra and Image:
        i) Enter the folder Dijkstra and Image and in both of them execute:
        For Linux and Windows:
        ```
        javac *.java
        ```
    b) Compile the rest of the project, executing the following commands in the root directory of the project.
        i) For Linux:
        ```
        javac -cp $CP *.java
        ```
        ii) For Windows:
        ```
        javac -cp %CP% *.java
        ```

3) To execute the project:
    a) Start RabbitMQ
    b) Run the Node Registry:
        i) For Linux:
        ```
        java -cp $CP Node_Registry
        ```
        ii) For Windows:
        ```
        java -cp %CP% Node_Registry
        ```
    c) Run a Node:
        i) For Linux:
        ```
        java -cp $CP Node
        ```
        ii) For Windows:
        ```
        java -cp %CP% Node
        ```

Note that the command mentioned only runs one Node, to execute more Nodes, the same commands must be executed again in another terminal.

d) Run the Overlay:
   i) For Linux:
```
java -cp $CP Overlay
```
   ii) For Windows:
```
java -cp %CP% Overlay
```

Once everything is running, the focus is on the terminal running the Overlay instance because it is where the functionality of the system can be used, eg. seeing the physical or virtual topology, connecting and disconnecting nodes, sending messages, etc...

JAR Files

To execute the project using the JAR files, where all the classes are already compiled, type the following commands:

1) Run the Node_Registry:
   a) java -jar node_registry.jar
2) Run Nodes:
   a) java -jar node.jar

   Execute this command for as many times as nodes are desired.
3) Run Overlay:
   a) java -jar overlay.jar

The commands are the same for Linux and Windows according to what it was found on the Internet. As both test computers are Windows, the Linux commands have not been tested.

CONCLUSION

We believe our project offers good functionality, with some bonus features (such as the graphical display), although there are some areas of error handling, stability and performance that could be improved upon. We faced some issues handling the multiple queue, single consumer situation; without a single converging exchange to manage the queues, the DeliverCallback expression proved somewhat complicated with which to work.

We found RabbitMQ a useful and effective technology to code with, despite certain aspects being challenging to work with. We would be interested in trying to expand this project, either functionality and performance, or in increasing the range of the application space. For example, it could be an interesting exercise to try and expand the system to accommodate multiple topological layers, such as an overarching topology composed of a number of cluster-like sub-topologies.

The project was very enjoyable and offered a delightful insight into the nature of connecting physical and virtual topologies in a real-world sense. We feel that this understanding has the potential to prove valuable in the future, as opposed to the amusing but less applicable project of a mini-game.