Master 1 MoSIG

Introduction to Distributed Systems

# Overlay

April 24, 2021

*Authors :*
William Templier
Arthur Van Schendel

*Supervisors :*
Ana Khorguani
Vania Marangozova-Martin

## Contents

# 1 Organization

## 1.1 The directory

Extracting the file `templier_vanschendel_overlay.tar.gz`, you shall find the following file tree:

```
Overlay/
├── lib/
├── manifest/
│   └── overlay.txt
├── out/
├── src/
│   └── overlay/
│       ├── exception/
│       │   ├── PacketException.java
│       │   └── RouteException.java
│       ├── frames/
│       │   ├── Message.java
│       │   ├── Packet.java
│       │   ├── RoutingTable.java
│       │   └── Sendable.java
│       ├── utils/
│       │   ├── MessageBuffer.java
│       │   ├── Neighbour.java
│       │   ├── Route.java
│       │   └── ShellColour.java
│       ├── Launcher.java
│       ├── PhysicalNode.java
│       └── VirtualNode.java
├── topologies/
│   ├── physical1.txt
│   ├── physical2.txt
│   ├── physical3.txt
│   └── physical4.txt
├── Makefile
└── node.jar
```

The `manifest/`, `lib/`, and `out/` folders are related to dependencies and compilation. The `topologies/` folder contains four different physical topologies that can be used. One must be chosen when executing the program (see below). These four topologies are visually presented in the appendix.
The `src/` folder contains the project, with the `overlay` package being the root. The `exception` package contains custom exceptions to simply give more semantic to the exceptions thrown and make the code more readable.
The `frames` package contains data structures that are sent through the network. **Message** is used at the *Virtual layer* to send, well messages. **RoutingTable** is a more complex, thread-safe through a read-write lock, data structure. It manages a list a **Route** (in the `utils` package) that is used by the *Physical Layer*. Both data structures are in fact children of the class **Sendable**, which is only used for the `data` field of the **Packet** class. The latter is in fact the "only" object transmitted over the network (others are encapsulated, but still *serialized*).
Finally, the `utils` package provides some useful data structures to the layers. **MessageBuffer** provides the functionalities of a thread-safe buffer (a typical *producer-consumer* one). It is either used between the layers for outgoing messages and incoming ones (thus two different buffers).
**Neighbour** is another thread-safe structure primarily for *pinging* the *Physical layer*'s neighbours and check whether they are dead or alive. **Route** is an important data structure for the routing algorithm and **ShellColour** is simply some utility class for the pretty printing used in the verbose mode of the *Physical layer*.

Finally, `Launcher.java` is the *main* of the program: it parses the shell command, creates a *Virtual layer* with the given identifier and its neighbour(s) according to the selected topology. Then an interacting shell starts, where the user can send messages to its virtual left and right neighbours. The `PhysicalNode.java` and `VirtualNode.java` are of course the core of this project, and are showcased throughout this report.

## 1.2 How to compile

Even though the directory contains the `node.jar` executable, you can always `make` your own:

```
 ~/overlay$ make
```

It uses `manifest` file from the to appropriately link the dependencies. It also populates `out/` with `.class` files but you need not bother about this. The Makefile is intended for Linux-based machines.

## 1.3 How to run

First, better switch on `rabbitmq-server`:

```
        ~/$ service rabbitmq-server start
```

and then, depending on the chosen topology:

```
        ~/overlay$ java -jar node.jar physical2 0
        ~/overlay$ java -jar node.jar physical2 1
        ~/overlay$ java -jar node.jar physical2 2 -v
```

In the example above, you need to run each of the command in a different shell. The first parameter is the topology (without the `.txt` extension), and the second is the virtual and physical layers' identifier. For the third node, you will have the verbose mode enable. You will have many messages about the *physical layer*'s activity, but the shell is a bit harder to use...

## 2 The Architecture

### 2.1 Neighbours & ID conventions

*Overlay* implements a virtual ring topology over a given physical network of hosts. For the ring topology, we chose the convention depicted in figure 1. The **right** neighbour is the next one going clockwise - or the subsequent identifier when incrementing. The **left** neighbour is the one next in counterclockwise order. In figure 1, node 0 has node 1 as its right neighbour, and node 4 and its left neighbour. Another convention that we agreed upon, is thenext matching between the *virtual layer* identifier and the *physical layer* one, to make things simpler.

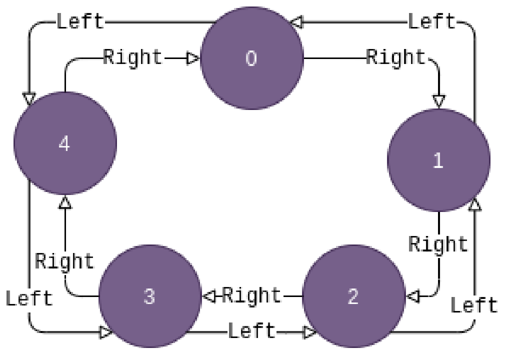


Figure 1: Ring convention

### 2.2 Inside a node

From the user's point of view, *Overlay* is just a basic, rather dull, shell that enables him or her to send messages to somehow mysterious left or right neighbours. Underneath the simplicity, a fairly elaborated structure is deployed and many workers synchronize themselves to be able to effectively send messages to their destination.

Note that throughout this report, we refer to **node** as a process that brings *Overlay* to life. The actual process is the *main* **Launcher**, that creates first a thread *Virtual layer*, which itself creates another thread, the famous *Physical layer*.

*Overlay* is a fully distributed application: all processes are equals and information about the network status is constantly exchanged. For network communication, we used RabbitMQ as its communication philosophy is more suited for a decentralized architecture. The *physical layer* connects to an *exchange* declared as **topic**, so we can use routing keys to address our neighbours.

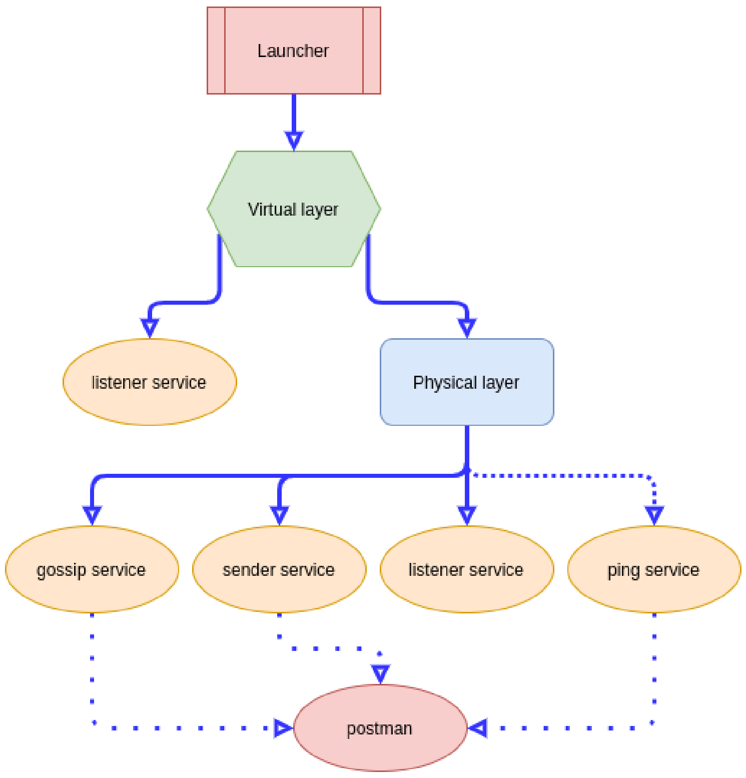


Figure 2: Decomposition of a node

Figure 2 above shows the structural decomposition of a node. The blue lines indicates that the component pointed to is handled by a thread. The short-dotted arrow to *ping service* indicates it is a scheduled thread that runs every $x$ seconds. The long-dotted arrows pointing to *sender* indicates that a specific thread is created for the service when they need to transit a packet (one for each packet). Figure 2 highlights that the *physical layer* is like a conductor that orchestrates many workers (not the don't play music unfortunately) - some of them which interact with the *virtual layer*, and others with the *physical layer* via packet transitioning on the network.

## 2.3 The Virtual layer

Once the *virtual layer* is instantiated, it fulfills two main functionalities: send and receive messages. As you can see in figure 3, the *virtual layer* is interacting with the *physical layer* via two buffers (the **Message-Buffer**). As the user enters a message to send in the shell, the *virtual layer* creates a **Message**, with its identifer as *sender*, summons the *physical layer* to obtain its neighbour's address. The shell input message is written into the data structure too, which is finally put into the `send buffer`.

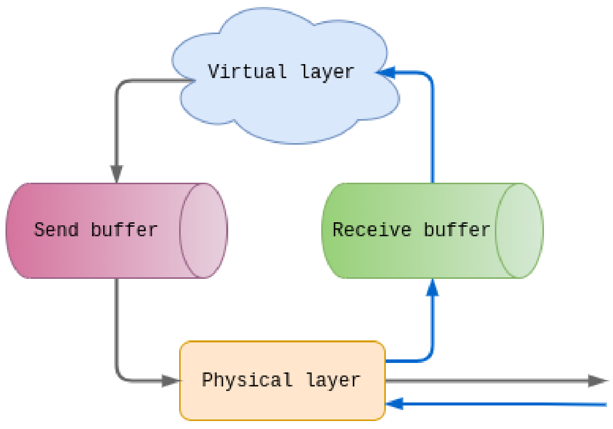


Figure 3: Decomposition of a node

On the *physical layer* side, the *sender service* takes the message, find the routing gate in the routing table and creates a **Packet** with the routing information (*i.e.* the `gate`) and the message in its `data` field. Finally the *sender service* calls the *postman*, which creates a thread dedicated to sending the packet over the RabbitMQ infrastructure. The *virtual layer*'s *listener service* is quite simple: it simply sleeps on the `receive buffer` and when a message arrives from below, it prints its content to standard output.

For these functionalities to work, we instantiate a pool of two threads: one of them is simply the *physical layer*, and the other thread is charged to listen for new incoming messages (listener service).

### 2.4 The Physical layer

The *physical layer* is the place where the magic happens: it contains the necessary RabbitMQ's components (connection factory, channel, queue etc..) to communicate with other nodes.

Via Java's *ExecutorService* API, the *physical layer* manages a pool of five threads: three more or less permanent: (*gossip*, *sender*, and *listener*), a scheduled service (*ping*), and *postman* which is mostly a functionality that is used by the other services and creates its own thread to perform its task.

The *gossip* works as follows: whenever a node's routing table is modified, it sends it to its neighbours. The *gossip* service is woken up by a post to a semaphore, it creates `TABLE` packets with the neighbours' address as destination and its routing table - then hands it down to the *postman*.

The *sender* is a rather simple one, it waits on the other side of the *virtual layer*'s `send buffer` and performs multiplexing: takes the **Message**, fills the `to` field, creates a `MSG`-type packet, and then hands it down to the *postman* service.

The *listener* service is somewhat the core of the *physical layer*: it performs the packet demultiplexing task for incoming packets. According to the packet type, it performs some functions. `HELLO` packet triggers the *gossip* service, like `TABLE` packet (if some modifications happened only) and `BYE` packet, after killing all routes associated to the dead gate.

`MSG` packets are unpacked to a **Message** and put into the *virtual layer*'s `receive buffer` if the node is the destination. Otherwise it simply hands over the packet to the *postman* who finds the appropriate gate to route the packet.

There is also a **PING** message type, that we explain within the *Dynamic Ring* section.

## 3 The distributed routing algorithm

As each node begins with only local information (*i.e.*, how to get to its neighbour(s)), it needs to exchange routing information with other nodes. Our routing algorithm is loosely based on our understanding/interpretation of the Bellman-Ford algorithm seen in our *Computer networks* class. We retain two properties from it, that we call **flooding** and **gossiping**. The former refers to the fact that one consequence of the algorithm is that the network is (almost) constantly flooded with routing tables information, that can often be redundant. The latter refers to the fact that whenever a node learns new information about the network, it gossips it to its neighbour(s):

Example 1

- Node1: *hey! do you know I can go to Node4 in two hops?*
- Node2: *hmm, that's three hops for me... I have better already, sorry*

Example 1 shows an instance when unnecessary information is sent to a neighbour. Node1 may have just receive from one of its neighbours that it can go to Node4. Happy to have new information it then gossips to Node2 about this new route... Nevertheless, Node2 already has a better route to Node4. It might even be the case that it was Node2 that sent this information to Node1. This situation is not precluded by our algorithm and this useless redundancy is one of its flaws.

Nonetheless, the example also shows that a strength of this algorithm is to ensure optimal and non-circular routes. When a node receives a known route, it compares it in terms of *number of hops*. Routing algorithms have different metrics to compute optimality: ours is to minimize the number of hops to destination. We see that Node2 ignores the route announced by Node1 since it would imply increasing its number of hops.

**Example 2**

- Node1: *Hey! do you know I can get to Node4 in two hops?*
- Node2: *No! thanks for the info, I'll add to my routing table that, to go to Node4, I just give you my packet!*
- Node1: *Sure!*

Example 2 simply shows an instance where through gossiping, a Node2 extends its routing table beyond local information via its neighbour.
Let us go a little be more in the details. Figure 4 below shows a rather simple example based on the topology `physical2`. At initialization, Node0 and Node2 only know about Node1 - which in returns also has information about both. Let us imagine Node1 is online first, it sends a `hello`-type packet that will be unanswered. Then Node0 comes online, and sends its `hello` packet. When Node1 receives it, its *gossip* service is triggered and will send Node1's routing table (`table`-type packet) to Node0. The latter *listener* service will see its a `table` packet and proceed to the routing table update. The route to 0 is ignored and the route to 2 is added. The routing table being updated, Node0's *gossip* service is triggered. In this case, its only neighbour is Node1 so completely useless information is resent - but this stage would be essential if Node0 had another neighbour, which would learn about Node1 and Node2.

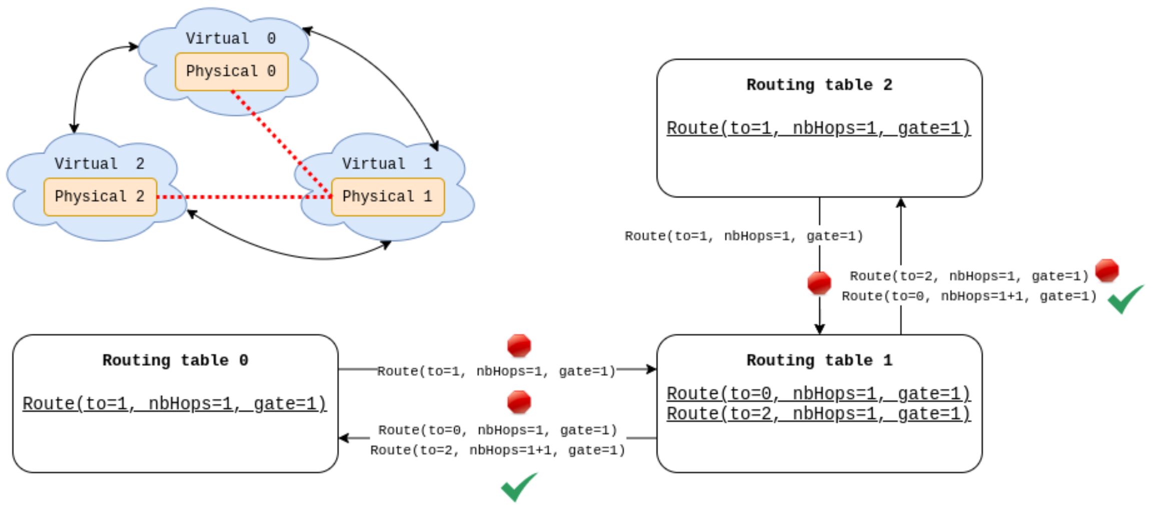
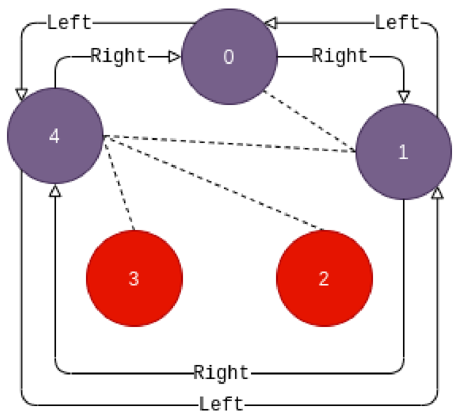

Virtual 0
Physical 0
Virtual 2
Physical 2
Virtual 1
Physical 1
Routing table 2
Route(to=1, nbHops=1, gate=1)
Route(to=1, nbHops=1, gate=1)
Route(to=2, nbHops=1, gate=1)
Route(to=0, nbHops=1+1, gate=1)
Routing table 0
Route(to=1, nbHops=1, gate=1)
Route(to=1, nbHops=1, gate=1)
Route(to=0, nbHops=1, gate=1)
Route(to=2, nbHops=1+1, gate=1)
Routing table 1
Route(to=0, nbHops=1, gate=1)
Route(to=2, nbHops=1, gate=1)

Figure 4: A simple gossip example

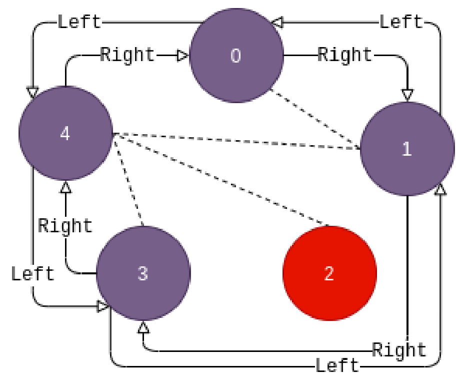
The functionality to update the routing table returns special **RouteStatus** and the *listener* service triggers the gossip service (by posting on a semaphore) whenever the status is `ADDED` or `UPDATED`. You might wonder why we need two statuses if the consequence seems to be the same. This has to do with the dynamic properties of the virtual ring explained in the following section, but basically `ADDED` is needed to increment a counter that is used to compute a node's left or right neighbour. Another possible status is `IGNORED`, in which case the *gossip* service is left sleeping beside its semaphore.

## 4 A dynamic ring

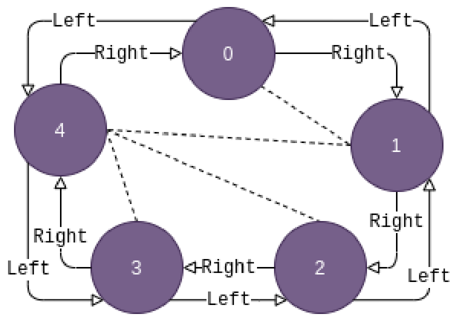
The dynamic property of our virtual ring is an extra feature that we implemented. It basically allows the ring to grow from one node, to all the nodes included in the given topology. If a node is alone, it cannot communicate. If there is two nodes, they communicate with each other (left and right being the same). Let us look at a more interesting situation.



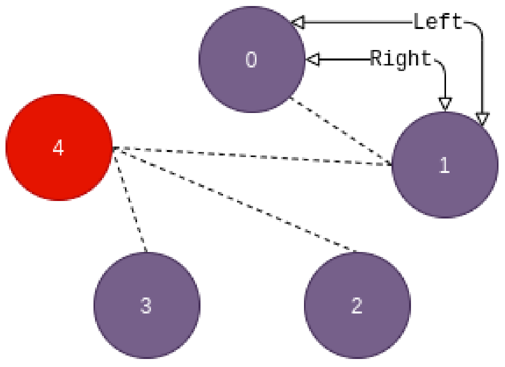
(a) 3 nodes with physical connection



(b) Node3 comes online, reconfiguration



(c) Node2 comes online, the ring is complete



(d) Node4 leaves, breaking the ring

Figure 5: Dynamic reconfiguration as nodes comes on or off

Figure 5 shows the dynamic configuration of the ring as the number of online nodes increases. In (a), Nodes 0, 1, and 4 can form a ring as Node1 can handle the routing. Then Node3 comes online and some of the neighbours are updated. When Node2 arrives, the ring is complete. In (d), we have the scenario where Node4 would go offline (either willingly or by failure). The ring is broken as Node4 was a sort of routing "hub". We see that Node0 and 1 can still communicate, but 2 and 3 are shut off. If Node4 reconnects, then automatically (after a certain delay) the configuration would go back to the one in (c).

The crucial element here is the ability to detect if one's neighbour(s) is/are alive or not. For this we need to add an extra information the the **Route**: an `alive` boolean that tells if the gate for the route is alive. If we come back to our example (d), Node3 is alive, but cannot go to Node1 (still alive), because its gate to it (Node4) is dead. With the `hello`-type packet sent upon connection, it is easy to know when our neighbour is alive. Same for the opposite, with the `BYE`-type packet. But for the latter, what happens is the Node4 is shutdown inadvertently, network connection is lost or whichever reason that makes it impossible for the `BYE` message to be sent or reach its destination(s)? This is where the *ping service* comes into play.

The *ping service* is a scheduled service that routinely awake to perform its duty. The service works with the **Neighbour** data structure mentioned earlier. It associates a node's neighbour(s) to a status boolean (alive or dead) and a time-to-live counter. When a `PING`-type packet is received, the *listener* service sets the boolean to true for the according neighbour and the TTL to $n$. When the *ping service* wakes up, it looks at the **Neighbour** table and decrements the TTLs by one. For all entries with a $TTL \leq 0$, the service pings them and then sleeps for $y$ seconds. Then it looks at its neighbours with a false boolean and if not already dead, kill them. Killing a route implies altering the routing table - which as a careful readers you are have already guessed - implies that we need to trigger the *gossip service*. Notice also that a node can be dead, but the service cannot know until the TTL is down to zero. This delay is a trade-off to avoid constantly pinging nodes that we know are alive.

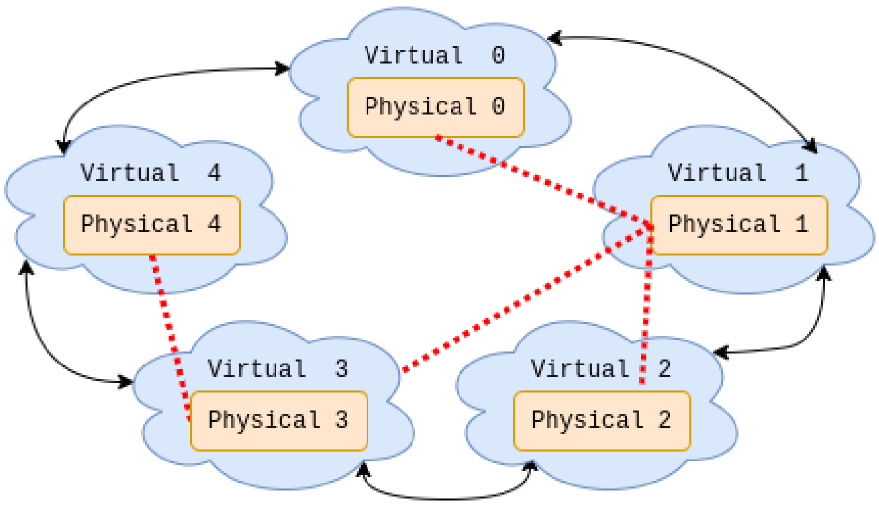
## 5 Conclusion

*Overlay* is in essence a fairly simply application. It is supported by a distributed algorithm that can be summarized as follows: *when you learn something about the network, spread the info until everyone is aware*. Then based on this, the *physical layer* can handle all the necessary routing for its *virtual layer* above to communicate with its neighbours.

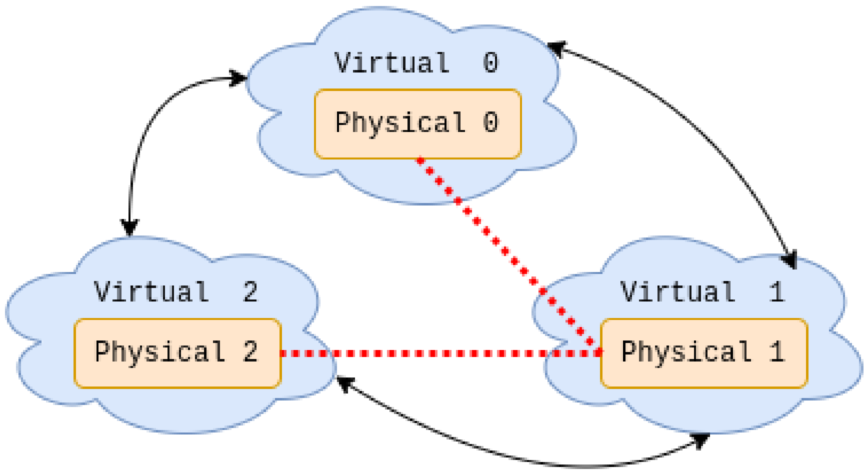
Things get more complex when we add a dynamic property. Finding about its neighbours is one thing, detecting who is dead or alive is another. We firstly added a shutdown hook to handle `CTRL+C` exit, but doing so made the *ping* service useless. In reality, we could keep it but since on a local machine, simulating network crash or failure is impossible - `CTRL+C` is our only way to test the feature.

As we explained, we chose RabbitMQ over Java RMI since we wanted a fully distributed architecture. Having a centralized node (like a routing server) to query for routing information would have made the routing task easier, but this creates a single failure point. What if this node crashes? Well you could cache some addresses and query it only once... But that's another project...

## A Available topologies



physical1

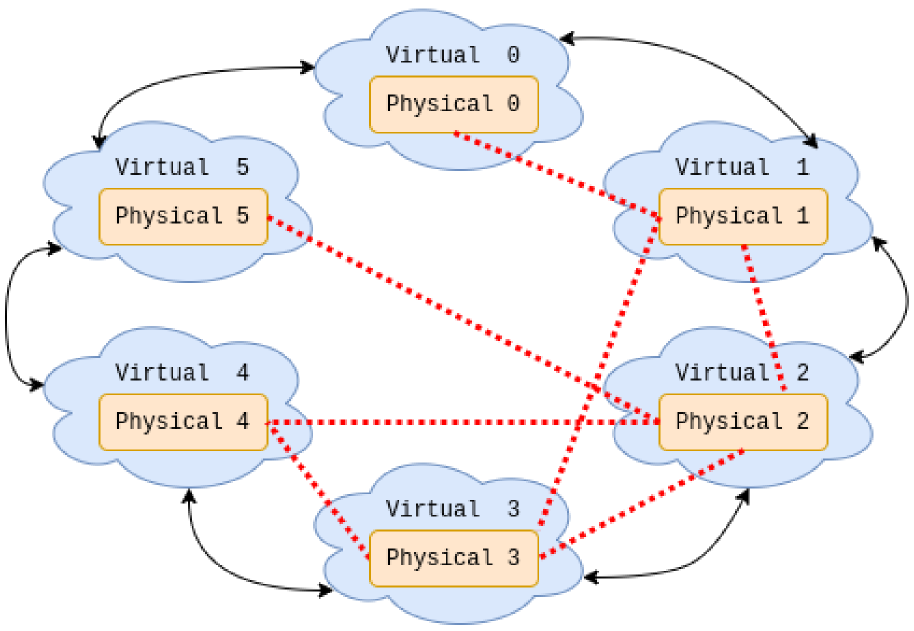


physical2

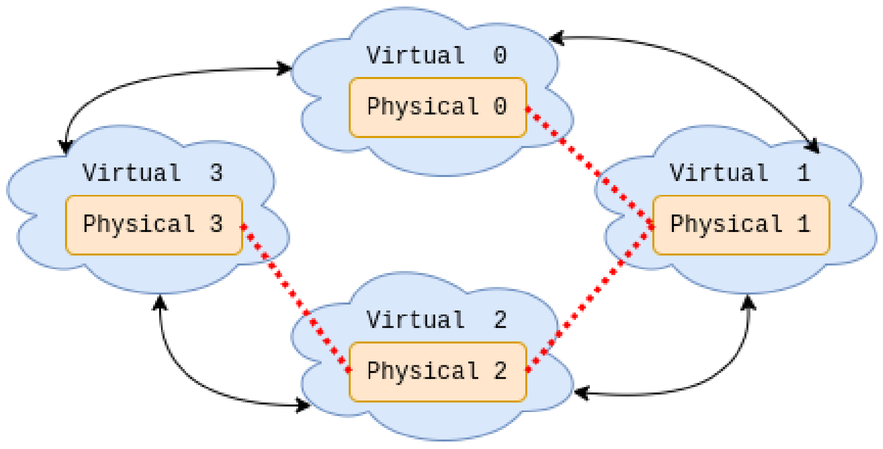
You can always device your own topology by creating a `topologies/physicalX.txt` file as follows:

```
0:1:0
1:0:1
0:1:0
```

for something like `physical2`



physical3



physical4