

Master degree in Computer Engineering

Distributed Systems
and
Middleware technologies



UNIVERSITÀ DI PISA

Key-value distributed storage

Project specification

December 2020

Federico Cappellini, Andrea Lelli, Alberto Lunghi, Lorenzo Susini

1 - Introduction	3
2 - Functionalities	4
2.1 - Service usage	4
2.1 - Functional requirements	4
2.2 - REST API specification	4
2.2.1 - Insertion by key	4
2.2.2 - Retrieval by key	5
2.2.3 - Deletion by key	5
3 - Non-functional requirements	7
4 - Architecture and data flow	7
4.1 - Distributed Insert-by-Key	8
4.2 - Distributed Get-by-Key	9
4.3 - Distributed Delete-by-Key	10
4.4 - Joining of a node	11
4.5 - Leaving of a node	12
5 - Client user manual	13
5.1 - Set the Rest Server URL	14
5.1 - Insert a new object	15
5.2 - Retrieve object	16
5.3 - Delete object	17
6 - Testing	18
7 - The key-value store	20
7.1 - Central Node	20
7.2 - Central Node Monitor	20
7.3 - Data Node	21
7.4 - Reader	21

1 - Introduction

The purpose of this project is to create a fast in-memory distributed key-value storage.

During large computations on distributed platforms not all the input, intermediate and output data can fit the memory of each individual computational node. This leads to the need for a shared storage system that can offer low latencies. The use of a traditional database system would lead, indeed, to the inevitable increase in latencies in data access and, when access operations are many and frequent, the result will be an increase in overall computation times. Furthermore, the complex query languages offered are in most cases a useless feature for a system that basically just needs to be able to store and retrieve data.

A valid solution is offered by in-memory key-value storage systems that can offer a simple and extremely efficient system for this purpose. The only drawback of these systems is the relatively low storage capacity. To solve this problem, it is possible to deploy this system, instead of on a single computer, on a network of computers that are able to share the workload equally increasing linearly the resulting storage capacity.

The introduction of a distributed algorithm also leads to other advantages such as the possibility of having a dynamic number of computers and therefore the ability to dynamically scale (horizontally) as the overall workload increases.

2 - Functionalities

2.1 - Service usage

Since the application includes an entire set of REST APIs, the storage service can be exploited by any third-party automated REST client.

Anyway a minimal graphic user interface will be created to let any human user interact with the system.

2.1 - Functional requirements

The following functional requirements have to be implemented:

1. **Insertion by key:** a client can request the insertion of a new value in the store related to a given key.
2. **Retrieval by key:** a client can retrieve an already stored value by specifying its key.
3. **Deletion by key:** a client can delete an already stored value specified by a key.
4. **REST APIs:** the access node has to implement a set of REST APIs to let the client perform the previous specified actions.
5. The system has to support the **joining of a new node** without the need to turn off the system or part of it.
6. The system has to support the **leaving of a node** without losing any value and without the need to turn off the system or part of it.

2.2 - REST API specification

2.2.1 - Insertion by key



The body of the request has to respect the following JSON structure:

```
{
  "key": <the key the client wants to use>,
  "value": <the value the client wants to use>
}
```

In case of success, the structure of the reply follows this JSON structure

```
{
  "status": <operations status>,
  "data": <Erlang reply>
}
```

where "operation status" is always "ok" and "data" can be:


- "ok" if the action was performed
- an informational message that warns the user that the key they want to use is already associated with a value

In case of severe errors, the structure of the reply follows this JSON structure:

```
{
  "status": <operations status>
}
```

where "operations status" is the error message.

2.2.2 - Retrieval by key

GET `/rest/kv-entries/{key}` Retrieve the value from the key-value storage by the key 

The body of the request is empty and the key is provided in the URL.

In case of success, the structure of the reply follows this JSON structure

```
{
  "status": <operations status>,
  "data": <Erlang reply>
}
```

where "operation status" is always "ok" and "data" can be:


- the actual value associated with the specified key
- an informational message that warns the user that the key they are looking for is not associated with any value

In case of severe errors, the structure of the reply follows this JSON structure:

```
{
  "status": <operations status>
}
```

where "operations status" is the error message

2.2.3 - Deletion by key

DELETE `/rest/kv-entries/{key}` Remove the entry from the key-value storage by the key 

The body of the request is empty and the key is provided in the URL.

In case of success, the structure of the reply follows this JSON structure

```
{
  "status": <operations status>,
  "data": <Erlang reply>
}
```

where "operation status" is always "ok" and "data" can be:

- "ok" if the action was performed
- an informational message that warns the user that the key they are looking for is not associated with any value

In case of severe errors, the structure of the reply follows this JSON structure:

```
{  
  "status": <operations status>  
}
```

where "operations status" is the error message

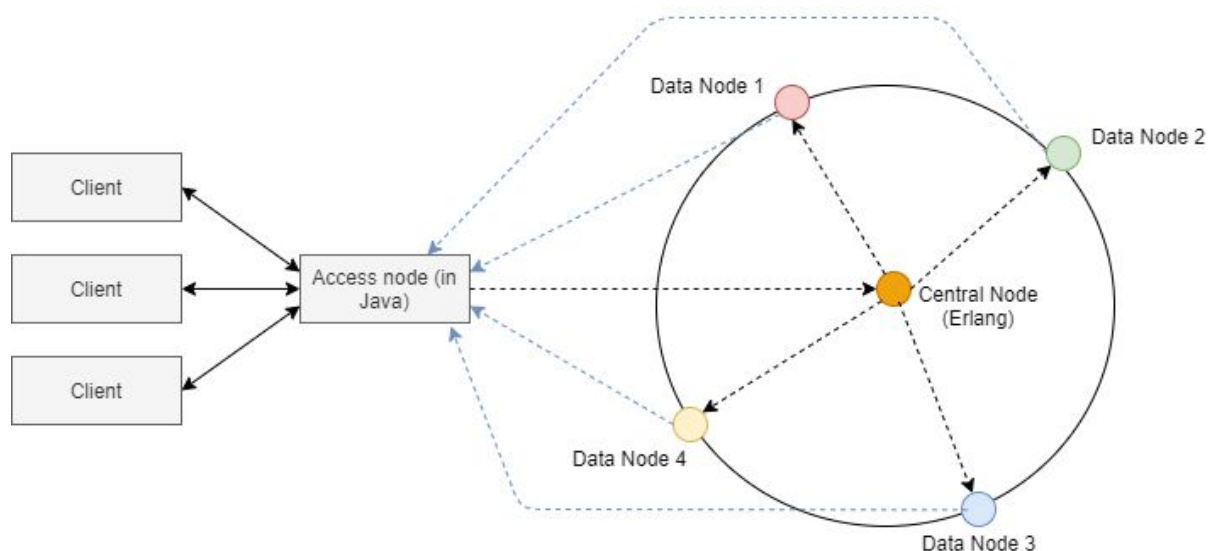
3 - Non-functional requirements

The system has to present some specific characteristics. Some of those can be expressed as non-functional requirements:

1. **Low latency:** the system has to have low latency in each operation. This result can be achieved using the in-memory paradigm.
2. **Load balancing across the nodes:** after the entrance of a new node, the system has to redistribute the minimal amount of stored values with the aim to balance the memory load between the nodes.
3. **Fast storage redistribution:** although the entrance of a new node can be considered as a “sporadic activity”, after the entrance, the system has to redistribute the minimal amount of stored values to guarantee scalability also with a large number of nodes and stored values.

Non-functional requirements 2 and 3 are satisfied by implementing consistent hashing.

4 - Architecture and data flow



Client: written in Java. It uses REST APIs to access the remote storage service.

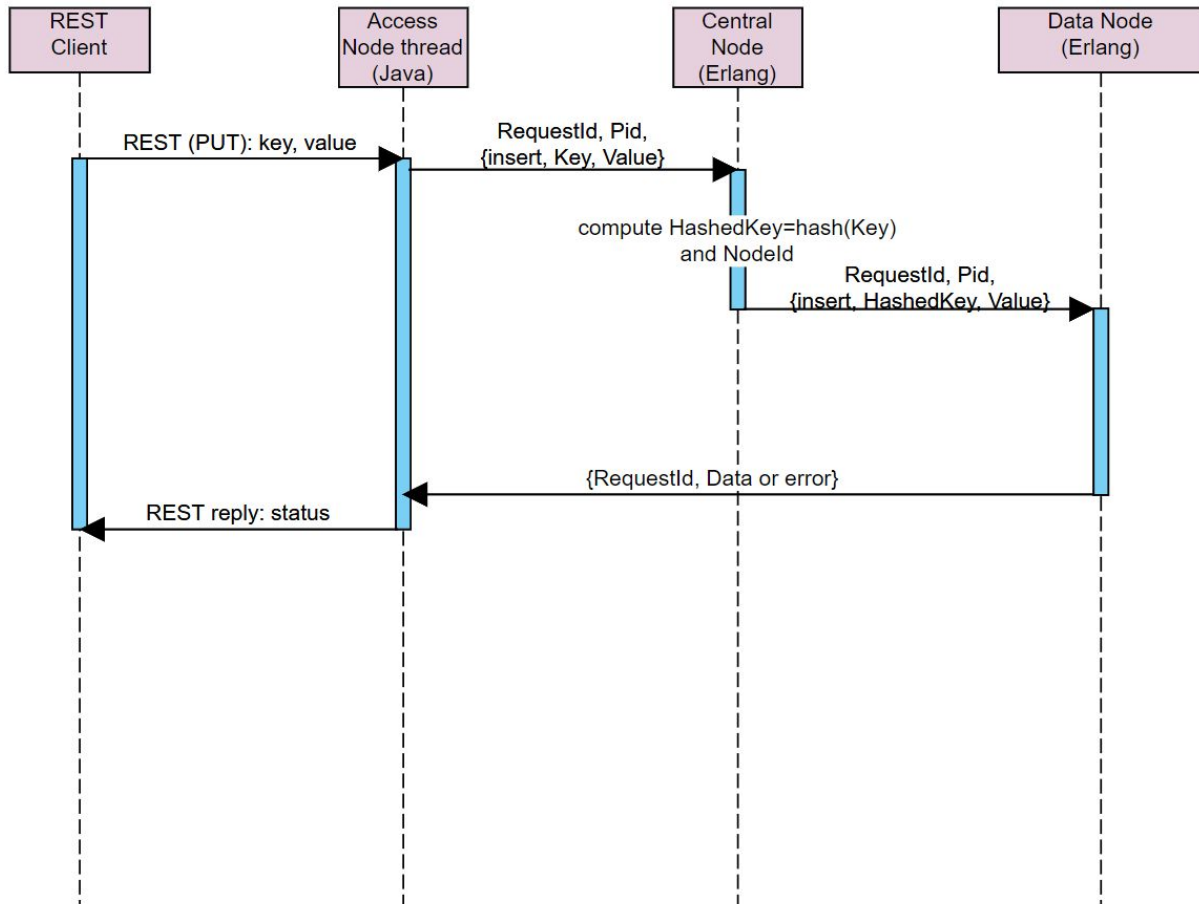
Access Node: written in Java. It exposes the REST APIs to clients and asks the Erlang central node to handle the requests coming from the clients.

Central Node: written in Erlang. It receives the requests coming from the Access Node and forwards them to exactly one Erlang Data Node, which is the owner of the key involved in the operation. It also manages the key partitioning during Erlang Data Node arrival and departure, by making use of Node Ids, hashed in the same way as the keys.

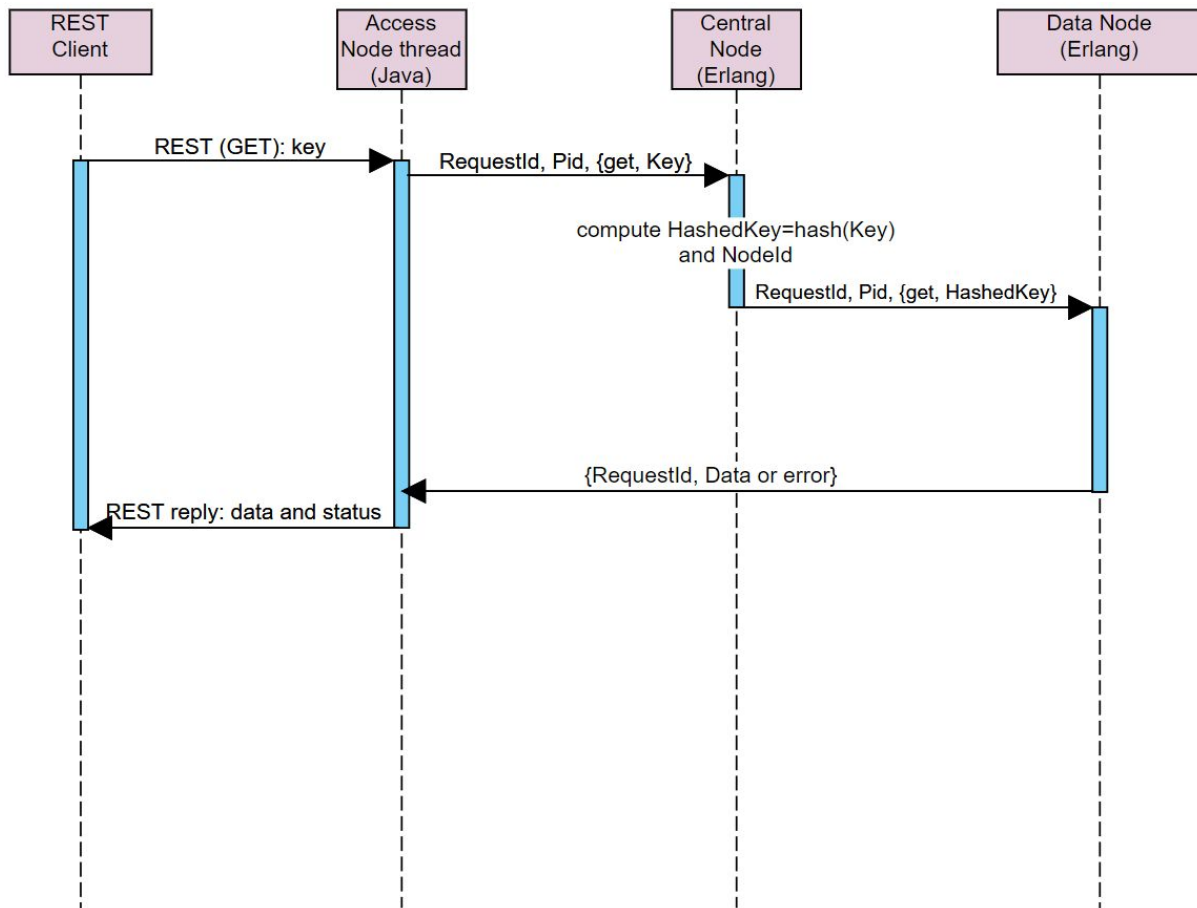
Ring of Data Nodes: written in Erlang. All the nodes are coordinated by the central node and take care of storing a portion of the key space. Both keys and Data Nodes are placed on the ring using the same hash function. Each Data Node stores all the keys for which their

corresponding hash value has as first successor that Data Node. They are also responsible for replying the result of an operation back to the access node.

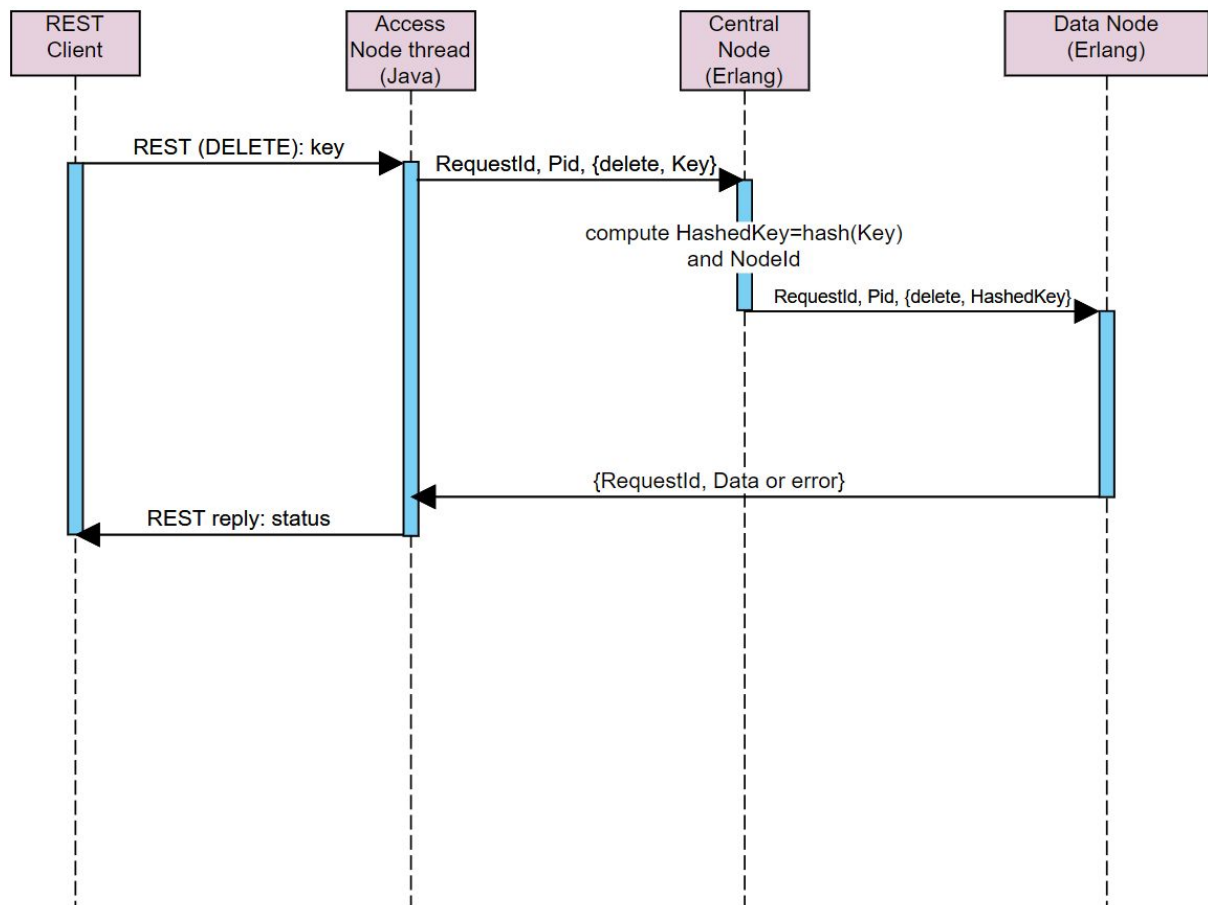
4.1 - Distributed Insert-by-Key



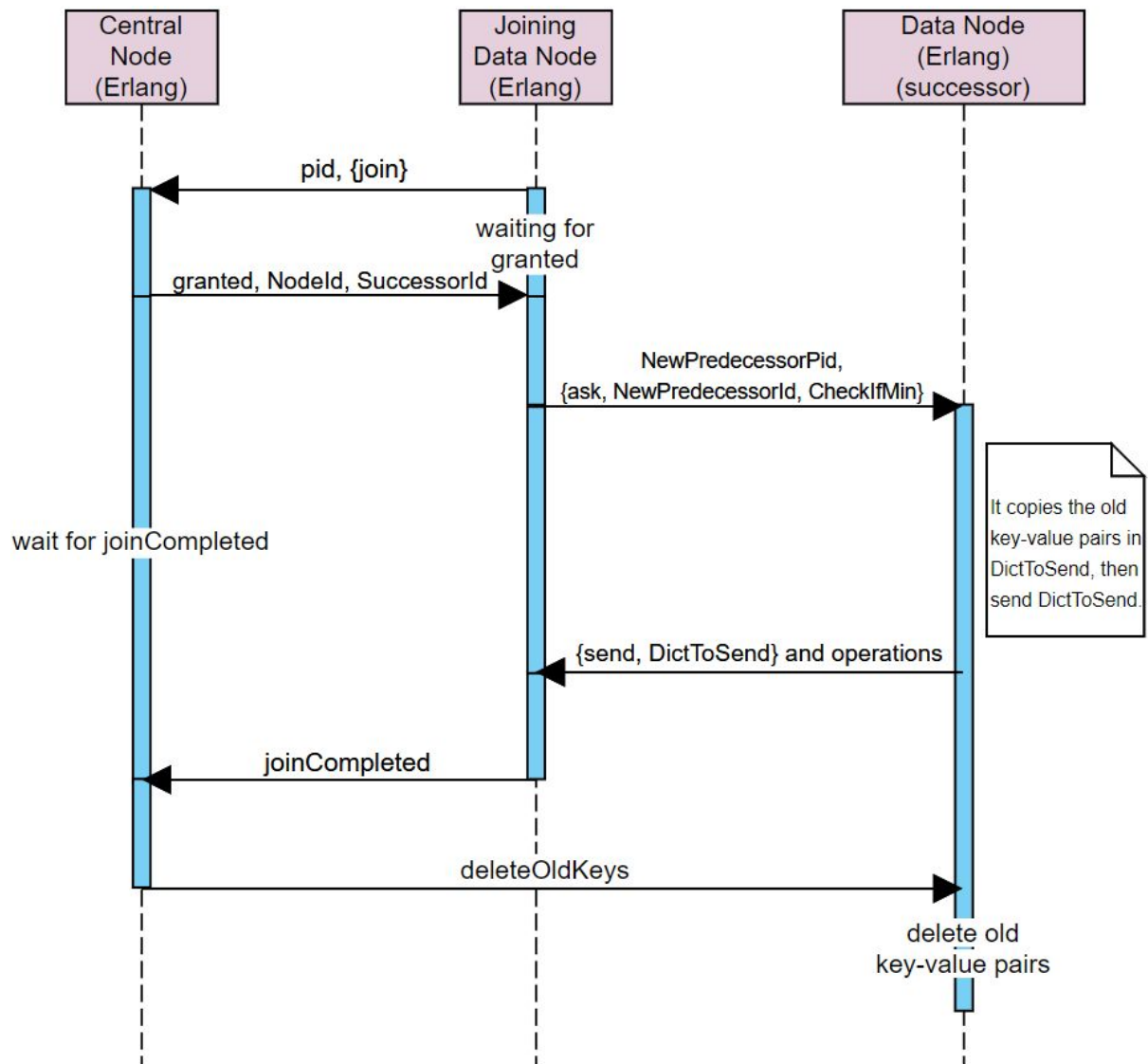
4.2 - Distributed Get-by-Key



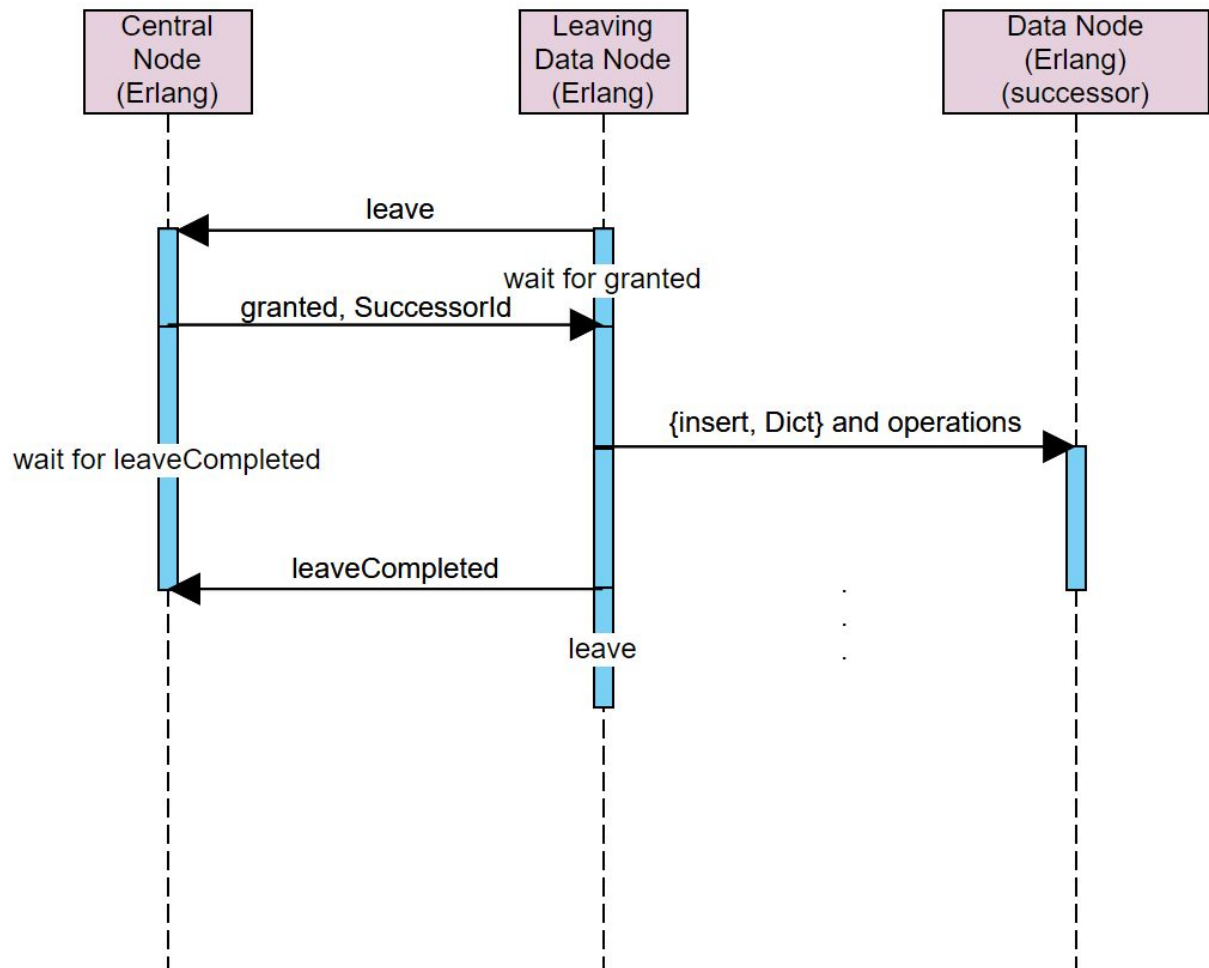
4.3 - Distributed Delete-by-Key



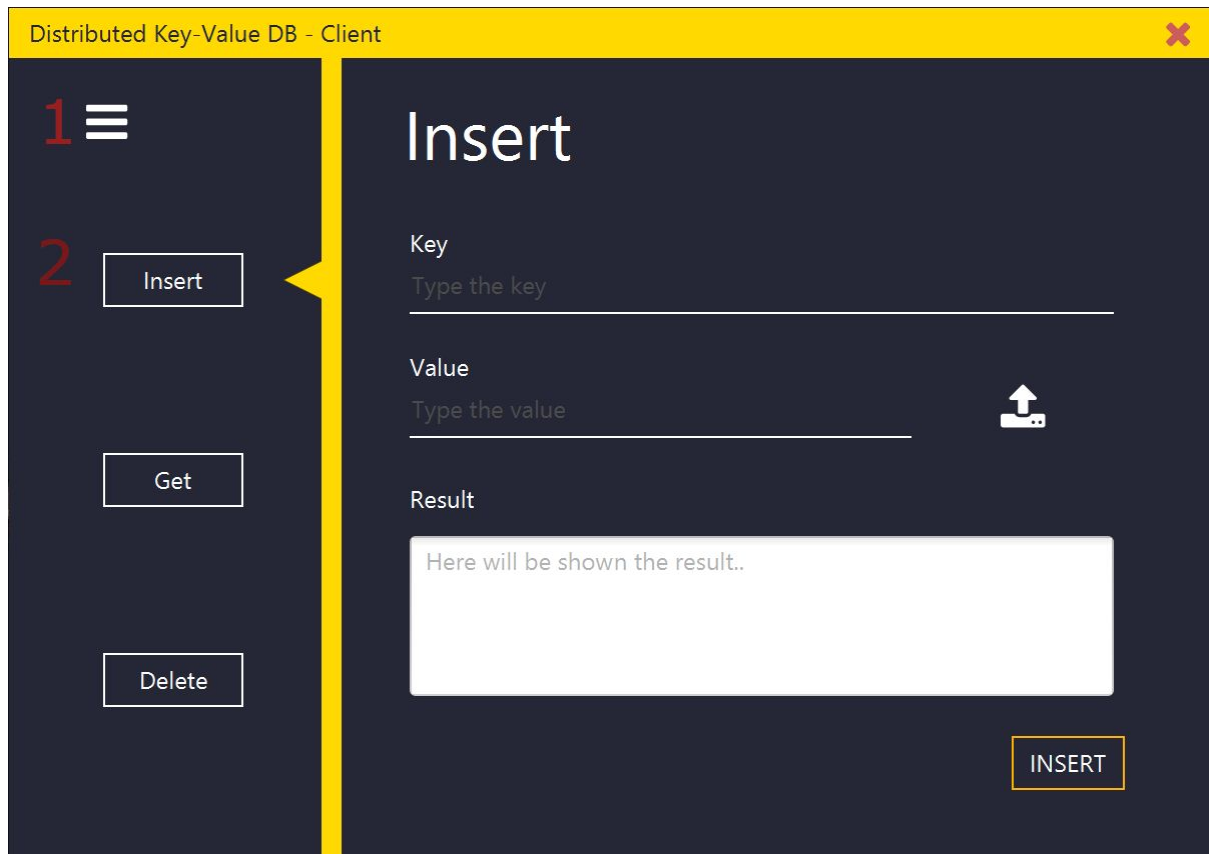
4.4 - Joining of a node



4.5 - Leaving of a node



5 - Client user manual



The main window: the initial window shown when the application is opened.

This is the initial window shown when the application is started.

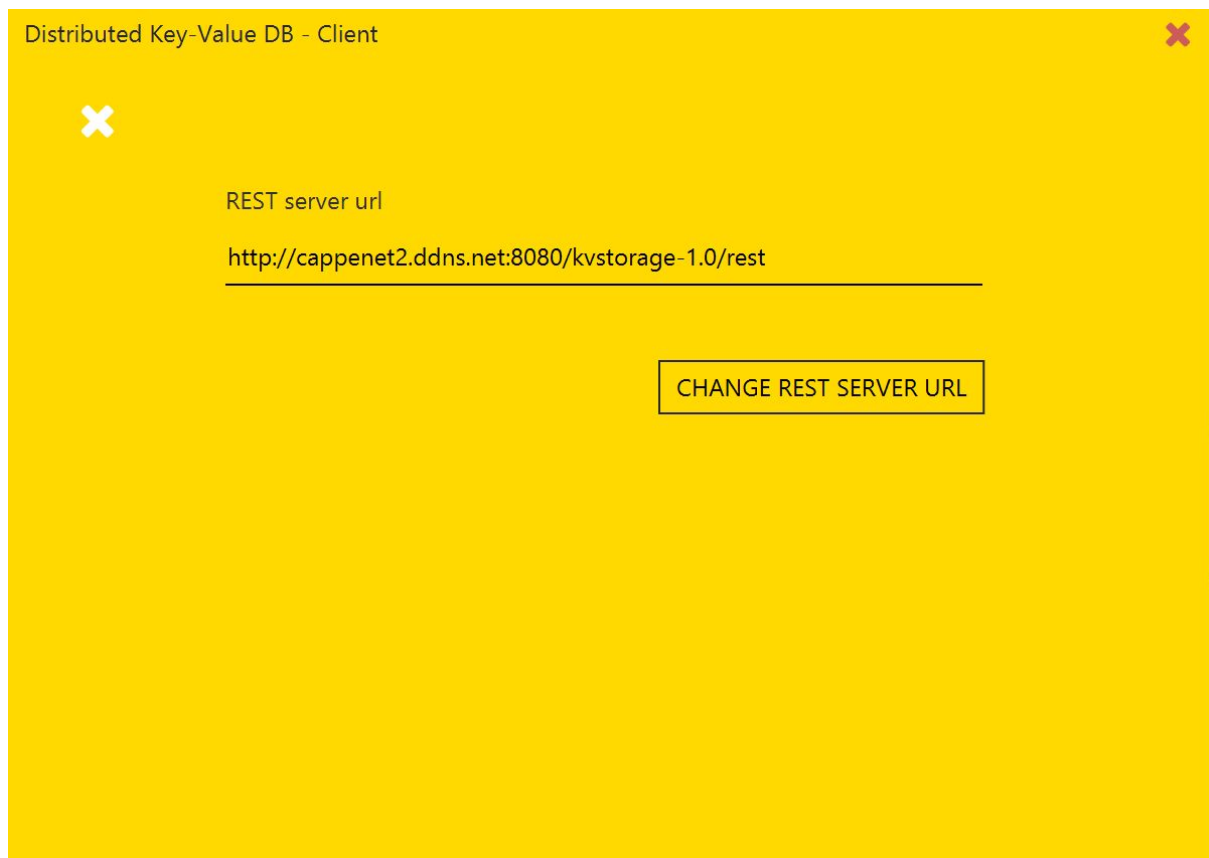
The main window shows:

1. The **open configuration button**:
 - when the user presses this button, the configuration window will be opened. This configuration window offers the possibility to change the REST server URL.
2. The **REST operation chooser**:
 - the user can choose between 3 REST operations (*insert new object*, *get object*, *delete object*). In order to do one of them the user has to press on the corresponding button and complete the operation. To the right of the yellow delimiter will be shown the chosen REST operation interface that the user has to use in order to complete the operation.

Any error such as “server unreachable” or missing a field in a request will be notified in a new alert window.

Both the main window and the alert window can be closed by pressing the red cross button located at the top-right of the window.

5.1 - Set the Rest Server URL



The configuration window the user can choose the REST server URL.

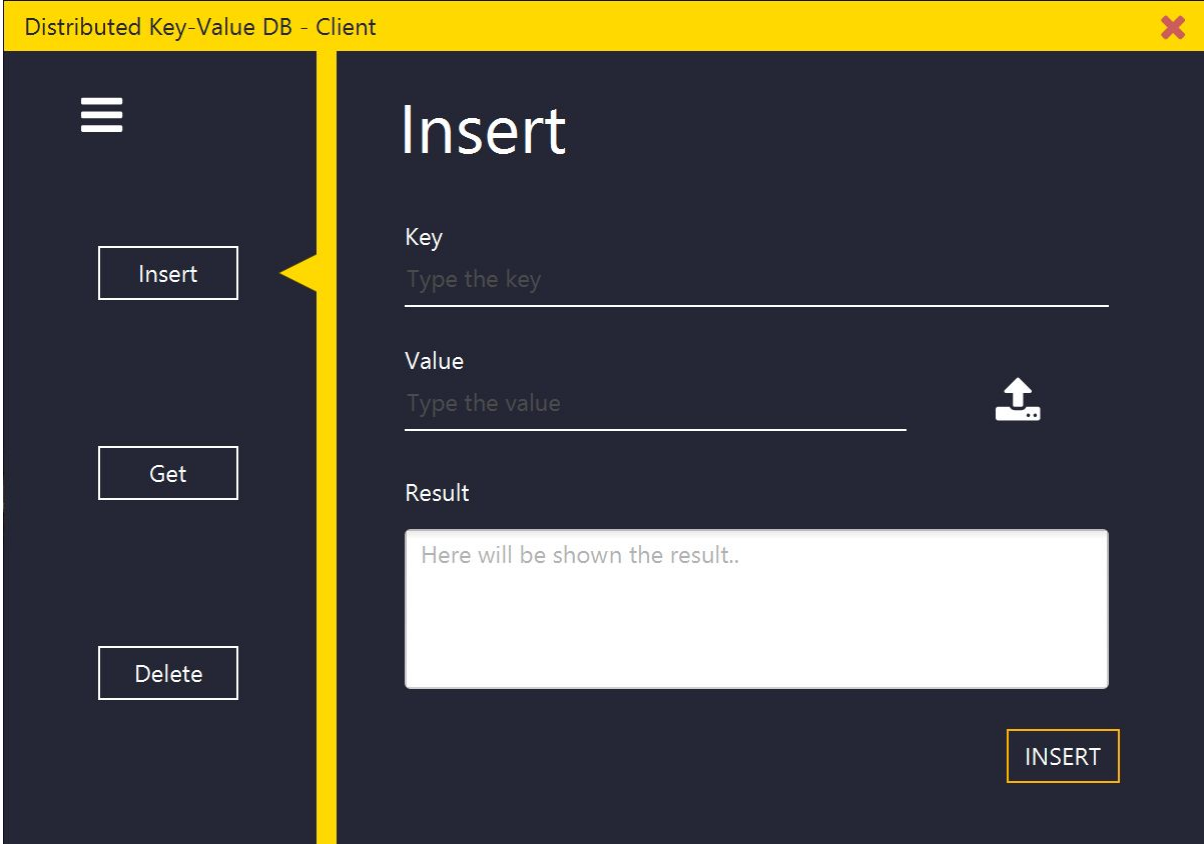
This is the configuration window that will be shown after that the user pressed on the open configuration button.

In the text field will appear the REST server URL read from the configuration file.

If the user wants to change it, he needs to write in the textfield the new URL and press the "CHANGE REST SERVER URL" button. In this way the URL in the configuration file will be updated with the new value and the Client will use it from now on.

When the user wants to return to the main window he has to press on the white cross button located at the top left of the window.

5.1 - Insert a new object



The screenshot shows a web application titled "Distributed Key-Value DB - Client". On the left is a dark sidebar with a menu icon and three buttons: "Insert", "Get", and "Delete". The "Insert" button is highlighted with a yellow arrow. The main area has a dark background and is titled "Insert". It contains three sections: "Key" with a text input field labeled "Type the key"; "Value" with a text input field labeled "Type the value" and a file upload icon to its right; and "Result" with a large white text area containing the placeholder text "Here will be shown the result..". A yellow "INSERT" button is located at the bottom right of the main area.

The insert operation window..

In the distributed key-value database the user can insert new objects but he must create a request in the form “key-value” pair.

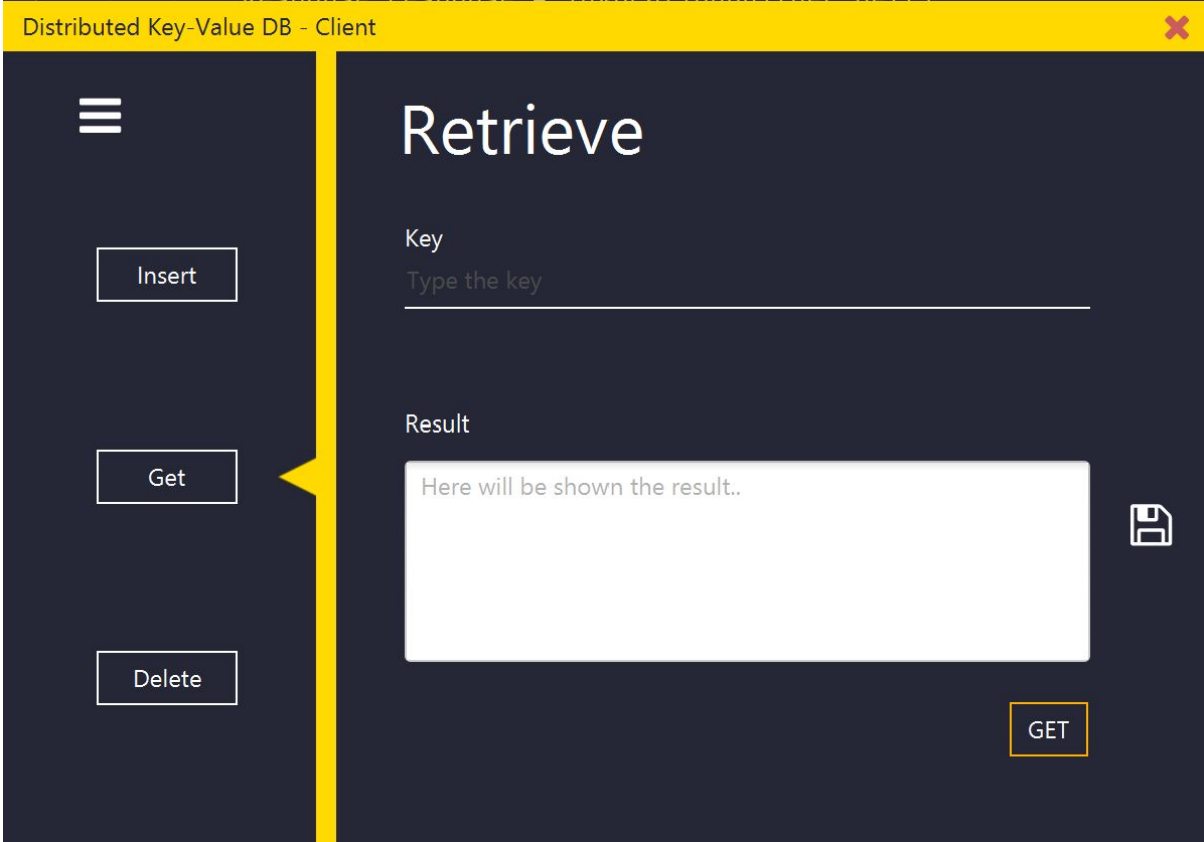
In order to create the key-value pair the user has to type both the key and the value in the corresponding text field.

The user has the possibility to load a file as the value of the key-value pair pressing on the load button located at the right of the value textfield.

In order to complete the insert operation the user has to press the *INSERT* button located at the bottom-right of the window.

The result of the operation will be shown in the corresponding textarea.

5.2 - Retrieve object



The screenshot shows a web application window titled "Distributed Key-Value DB - Client". The interface is split into a left sidebar and a main content area. The sidebar contains three buttons: "Insert", "Get", and "Delete". The "Get" button is highlighted with a yellow arrow. The main content area is titled "Retrieve" and contains a "Key" input field with the placeholder text "Type the key". Below the input field is a "Result" section with a large white text area containing the text "Here will be shown the result..". To the right of the text area is a "save as file" icon. At the bottom right of the main content area is a "GET" button.

The retrieve operation window.

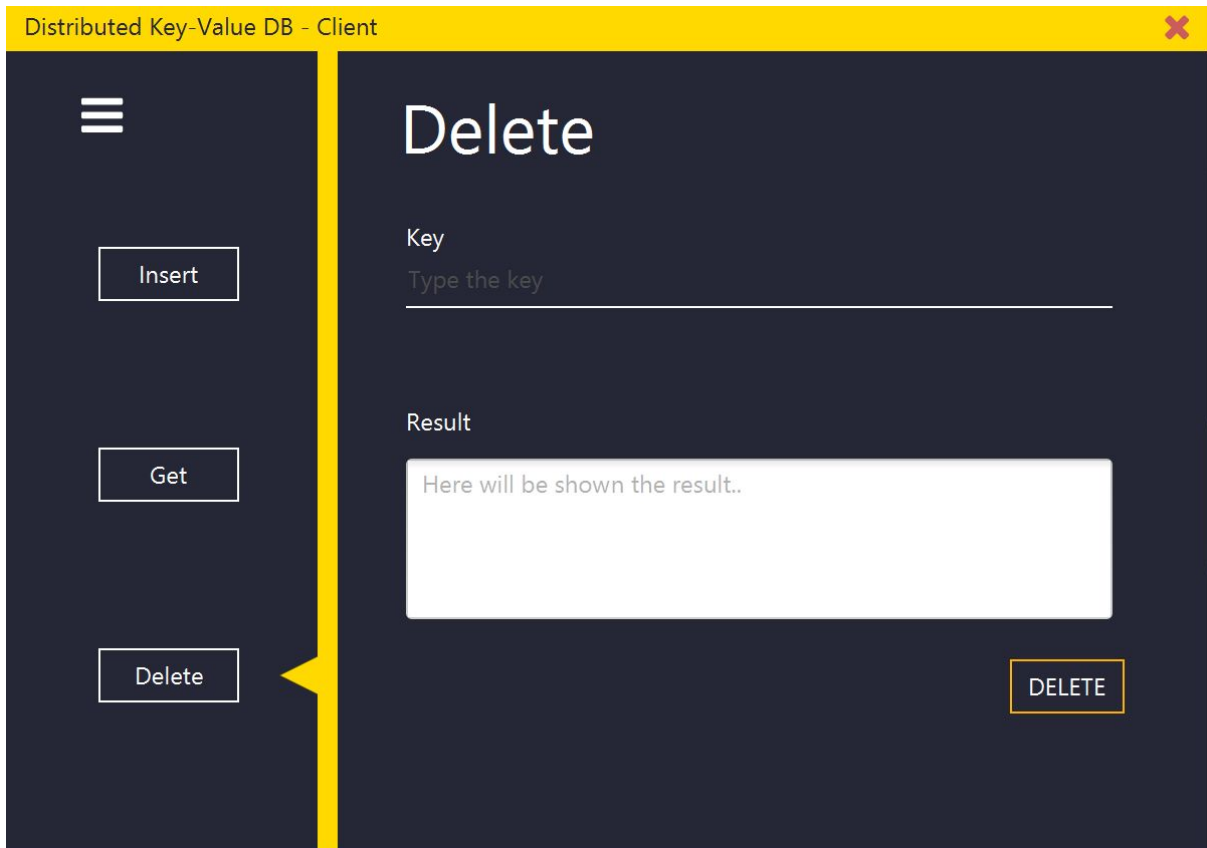
In the distributed key-value database the user can retrieve the value of the object by the key that he previously chose for the key-value pair.

The user can type the key in the corresponding textfield and complete the retrieve operation pressing the *GET* button located at the bottom-right of the window.

The result of the operation will be shown in the corresponding textarea.

If the result is a valid file content, a "save as file" button will appear at the right of the result textarea and the user can save the result as a file by pressing on it.

5.3 - Delete object



The screenshot shows a web application window titled "Distributed Key-Value DB - Client". The interface has a dark blue background with a yellow header bar. On the left, there is a sidebar with a menu icon (three horizontal lines) and three buttons: "Insert", "Get", and "Delete". The "Delete" button is highlighted with a yellow arrow. The main area is titled "Delete" in large white text. Below the title, there is a "Key" label and a text input field with the placeholder text "Type the key". Below that, there is a "Result" label and a large white text area with the placeholder text "Here will be shown the result..". At the bottom right of the main area, there is a yellow button labeled "DELETE".

The delete operation window.

In the distributed key-value database the user can remove the key-value entry from it using the key that he previously chose.

The user can type the key in the corresponding textfield and complete the delete operation pressing the *DELETE* button located at the bottom-right of the window.

The result of the operation will be shown in the corresponding textarea.

6 - Testing

Tests were performed following a bottom-up approach.

The various components of the distributed application were developed and tested in artificial and isolated environments in order to identify intrinsic problems such as bugs and configuration errors.

Since the development of the webapp, the central node, the data node and the client started independently the following approaches have been adopted:

- A fake central node was built in Erlang to test the REST webapp. It was simply used to reply with fixed responses.
- The first skeleton of the data node was used to create a fake data node to test the central node. It was simply used to reply with fixed responses too.
- The data node was tested using commands sent manually from an Erlang shell.
- The first skeleton of the web app was used to test the client application. It was used to reply with fixed error messages or fake data.

Once the single components were successfully tested, the overall application was built joining them together. Then, the testing phase was split into three further steps.

At first, all the components were run on a single host to identify all the problems related to module interfacing.

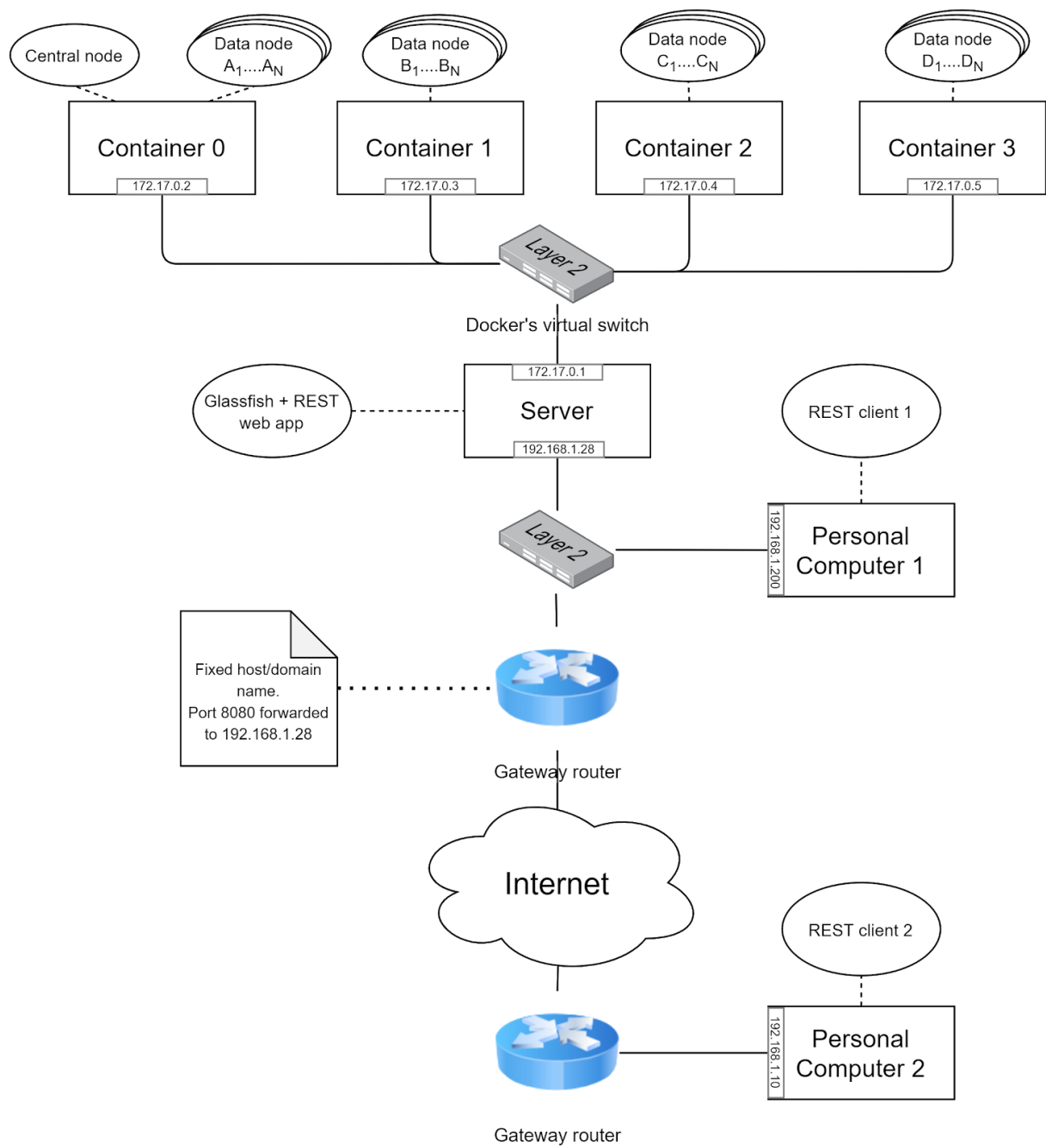
Then a more realistic scenario was created exploiting virtualization techniques to identify all the possible problems related to module distribution.

The architecture is described graphically thanks to the following image.

As last step, although the use of the service by humans did not reveal difficulties for the system, a strategy was adopted that could make a more intensive use of the system: graphical clients have been replaced by Python scripts aimed at maximizing the system throughput performing consecutive REST requests.

All issues highlighted during testing have been fixed with minor changes to the source codes.

During the last test, good overall performances were achieved despite the hardware available was not enterprise level.



7 - The key-value store

7.1 - Central Node

The Erlang central node acts like the main coordinator of the system. It is implemented as a classic server in Erlang. Indeed, it is a process which always recursively executes a function, keeping its state in its arguments and receiving incoming messages from its mailbox. The solution without generic servers was chosen to have fine grained control on the mailbox and to easily implement a selective receive when needed, exploiting the saved mailbox mechanism offered by Erlang.

The central node is responsible for:

- accepting incoming requests from the clients by receiving messages from the access node;
- accepting new data nodes joining the system. When a new data node wants to join the system, it can easily send a joining request to the central node, which is easily reachable given that its pid is registered at start up;
- maintaining the state of the system, which is a data structure containing the NodeId and the Pid of the data node servers. In particular, a list of tuples was chosen as data structure;
- routing the requests to the correct server, which is the one whose NodeId is the first successor of the key involved in the operation. In fact, by keeping the list of servers sorted by NodeIds in ascending order, the algorithm to choose it was really simple to implement. In practice, given a hashed key, the sorted list is kept scanning until a server whose NodeId is higher than that key is found. If no such server is found, the first server in the list is the owner of that key. This way, the concept of ring is implemented;
- monitoring the state of data nodes. Given that the central node keeps the state of the system, it is useful to let it monitor the data nodes and to let it detect whether one of them crashed, modifying the state accordingly. To do that, we decided to use the *monitor* function offered by Erlang: whenever a data node crashes, a message to inform the central node of its failure is sent.

7.2 - Central Node Monitor

As it can be easily guessed, the central node represents a single point of failure of the system. If that node goes down, the entire system will not work. As such, it is worth monitoring it and detecting whether it is still up and running or not, in every moment. Again, we decided to implement a monitor by exploiting Erlang itself, through the usage of the *monitor* function. The central node process is spawned by the monitor, which can start it with a given initial state or an empty initial state. When the system is started, the central node is started with an empty list of servers. Anytime the central node modifies its state (on joining requests or on crashing or leaving of a data node), it informs of that update also its monitor process. This way, the system keeps a backup copy of its state, so that if it crashes in the future, the monitor can respawn a brand new central node process with (hopefully) the same state it had at the time of failure, solving the single point of failure problem.

7.3 - Data Node

The Erlang data nodes are the last tier of the distributed web app. They are in charge of maintaining the key-value pairs stored in the system by the clients and offer the functionalities to insert, retrieve or delete them.

Data nodes are processes which always recursively execute a function, keeping their state in the arguments of the function and receiving incoming messages from their mailbox.

The state, in particular, is composed of a dictionary, which is used to store the key-value pairs, the node id of the data node, computed by the central node, and the location of the latter, so as to send messages to it.

As soon as a data node is spawned, it immediately sends a join request to the central node and waits for the permit. It also eventually asks for keys and operations to its successor, with respect to the ring structure, and then terminates the join and enters the infinite loop of the server.

Each data node is responsible for:

- accepting insert requests from the central node and inserting in its local dictionary the new key-value pair;
- accepting get requests from the central node and retrieve the value associated with the specified key;
- accepting delete requests from the central node and delete the key-value pair associated with the specified key;
- accepting a leaving request from an administrator of the distributed system. In this case the data node must ask the central node for the permit to leave the system, and if it is the case it has to send all its keys and operations to its successor.

When a user query is received by the data node, it is also responsible for replying back the outcome of the operation to the access node.

7.4 - Reader

This is a simple module that reads a specific line from a file.

It has been implemented to give to the data nodes the possibility to retrieve the location of the central node when they are spawned. In practice the data nodes use the module to read from a file, written by an administrator, the specific line that contains the name and ip address of the central node.

This setting is specific of the chosen deployment for the whole application: since each process can be spawned into one of 4 containers, named "Cont0", "Cont1", "Cont2" and "Cont3", when the data nodes are turned on, it is specified the id of the container where the central node has been deployed, to the start function.

Thanks to the Reader module, the data node will read the correct line of the configuration file that contains all the possible locations for the central node, based on the given id.