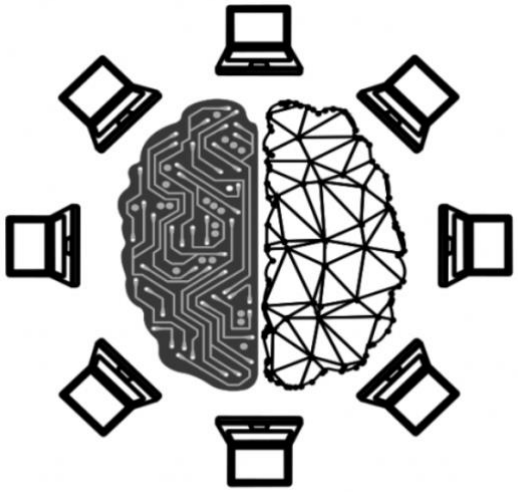


Horizontal Federated Learning

Distributed Systems
and Middleware Technologies

University of Pisa

Fabio Buchignani

Franco Terranova

Antonio Patimo

SUMMARY

1. ABSTRACT 3
2. INTRODUCTION AND REQUIREMENTS 4
2.1 FUNCTIONAL REQUIREMENTS 4
2.2 NON-FUNCTIONAL REQUIREMENTS 5
3. HORIZONTAL FEDERATED KMEANS ALGORITHM 6
3.1 ERLANG COMMUNICATION LAYER 7
3.2 SUPERVISOR PROCESS 8
3.3 SERVER PROCESS 8
3.4 CLIENT PROCESS 9
3.5 CORRECTNESS EVALUATION 9
3.6 SUPPORT ON MULTIPLE MACHINE LEARNING ALGORITHMS 10
3.6.1 SERVER MODULE 11
3.6.2 CLIENT MODULE 12
4. WEB APPLICATION 13
4.1 PROJECT ARCHITECTURE 13
4.2 PROJECT STRUCTURE 14
4.3 SERVLETS AND FILTERS 15
4.4 SERVICES 15
4.5 SYNCHRONIZATION 16

1. ABSTRACT

Federated learning is a technique that allows to train a Machine Learning Algorithm through the use of decentralized devices or servers that maintain its own data without the need to exchange it. This goes beyond the use of local models that make predictions by bringing model training to the host as well. As we know, standard Machine Learning approaches require the training data to be centralized, and this has serious practical problems, such as high communication costs, large consumption of device batteries, and the risk of violating the privacy and security of user data.

Federated Learning allows to improve privacy and open up the access to new data that would otherwise be out of reach when building machine learning models, ensuring us to reach smarter models, lower latency, less power consumption, all while ensuring the privacy of our data. The establishment of data security regulations will in fact contribute to a more civilized society but will also pose new challenges to the procedures of data handling typically used in AI.

2. INTRODUCTION AND REQUIREMENTS

Our web application allows the user to execute machine learning algorithm in a distributed setting. The web application is deployed on a tomcat web server and make use of an Erlang layer that manage the communication and coordination with a python layer that executes the machine learning algorithms.

2.1 FUNCTIONAL REQUIREMENTS

- A user can configure a new experiment by specifying the name of the experiment, the dataset to be used, % of participants, whether these participants could be different at each Federated Learning round, the machine learning algorithm to be used and the specific parameters for the algorithm.
- A user can see in real-time the progress of the execution of the experiment with a plot of the evolution of the cluster against two different features of the dataset, a plot of the frobenius norm variation, the number of rounds needed, the clients involved at each round, the number of crashes, the reason of termination, the time for the execution and the logs of the execution.
- A user can export the results of an experiment in a TXT format.
- A user can see all the already configured experiments with all their information.
- A user can search experiments filtering by name, dataset, user and algorithm.
- A user can edit the configuration of an experiment and if it changes important parameters, the experiment will be re-executed.
- A user can delete the configuration of an existing experiment.
- A user can change its account information.
- An administrator user can edit the default parameters, like the client hostnames, the maximum number of crashes and the maximum number of rounds at each experiment and parameters regarding the division of the dataset in chunks.

2.2 NON-FUNCTIONAL REQUIREMENTS

- Availability – It is necessary to have a distributed structure that guarantees high availability.
- Flexibility - The application should support different kinds of Machine Learning algorithms.
- Speed - The application must have short response time.
- Usability – The application must be simple to use.
- Erlang Communication Layer - The communication layer must be written in Erlang.
- Fault Tolerance - The application must be fault tolerant.

3. HORIZONTAL FEDERATED KMEANS ALGORITHM

Horizontal federated learning, or sample-based federated learning, is introduced in the scenarios that data sets share the same feature space but different in sample.

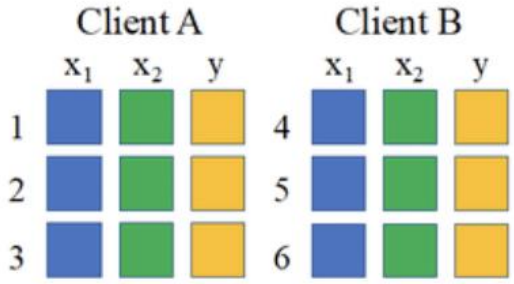


Figure 1: Horizontal federated learning

In this project, we realized a communication layer using Erlang for a Horizontal Federated Learning Algorithm, taking in consideration KMeans as target Algorithm.

In our implementation, we've taken in consideration the Lloyd algorithm for K-Means clustering.

The server oversees the initialization stage, starting from the number of clusters, it randomly initializes their centers.

Then, the execution stage is iterated until the stop condition occurs. At each round, the server transmits the current array of cluster centers to each data owner. Locally, each data owner computes the linear sum and the number of objects for each cluster and transmit them to the server.

$$LS(c)i \ \forall c \in \{1, \ldots, k\};$$

$$N(c)i \ \forall c \in \{1, \ldots, k\};$$

The server can thus update the cluster centers by exploiting the information shared by all the data owners:

$$\frac{LS(c)i}{N(c)i} \forall c \in \{1, \ldots . k\}$$

Afterwards, the server evaluates the termination condition and, if the Frobenius norm of the difference in the cluster centers between two consecutive rounds is lower than a given threshold, the execution terminates.

It can be proven that the algorithm produces the same clusters that would be produced by the classical K-means applied to the entire dataset in a centralized system.

To develop the project, the Erlang application used an existing Python logic which defines all the operations, for client and server, involved for the training, described by the following pseudo-code:

Algorithm 1 Main algorithm

```
procedure HORIZONTAL_CMEANS_FEDERATED(parameters)
    n_clients ← parameters["n_clients"]
    dataset ← parameters["dataset"]
    client_params ← get_client_params(parameters)
    server_params ← get_server_params(parameters)
    dataset_chunks ← generate_dataset_chunks(n_clients, dataset)
    clients ← initialize_clients(n_clients, dataset_chunks, client_params)
    server ← initialize_server(parameters, server_params)
    while server.next_round() do
        centers ← server.get_centers()
        client_responses ← call_evaluate_cluster_assignment(clients, centers)
        server.process_clustering_results(client_responses)
    return server.get_centers()
```

Figure 2: algorithm's pseudo code

3.1 ERLANG COMMUNICATION LAYER

In our implementation we can distinguish four different types of processes for each experiment, the Supervisor Process, the Server Process, the Client Processes and the Pyrlang Processes.

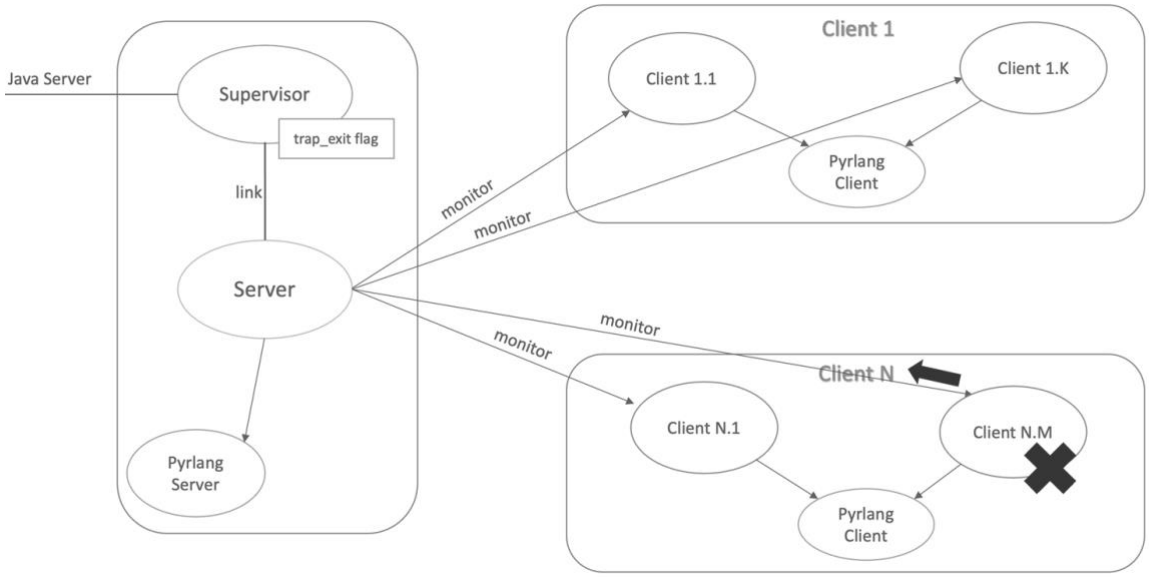


Figure 3: supervision tree

The supervisor process and the server process will be hosted in the same node, while the client processes can be hosted in the same or different nodes.

One pyrlang process will be available in each of the client nodes.

Pyrlang is a Python library which implements Erlang distribution protocol and creates an Erlang-compatible node in your Erlang cluster.

Further information can be found at https://pyrlang.github.io/Pyrlang/.

3.2 SUPERVISOR PROCESS

The supervisor is in charge of supervision & control for a designated experiment.
The supervisor process sets its trap_exit flag and acts a system process.
It first spawns the server process, providing the correct information, links to it and then wait for information to be received at each round.
This information will be sent to the Java server for further usage.
In case the server fails, the supervisor will try to restart the server if possible.
A specific parameter will provide the maximum number of attempts to use for re-spawning a server for a specific experiment.
If the number of attempts has been overcome, it terminates with a message error and the experiment fails.

3.3 SERVER PROCESS

The server is in charge of monitoring the execution of rounds.
It first creates a pyrlang process, erlang process used for running the python functions, and asks for the generation of the chunks of the dataset.
These chunks will be generated by dividing the dataset into N splits, where N is the number of clients involved in the experiment.
The clients are now spawned by the server, assigning a different chunk to each of them.
The server will also create a monitor to each client for supervision purposes.
The iterations are then started, with an initial generation of random centers.
At each round, the server decides which are the clients to be involved.
For the experiment we can decide to involve a subset of the clients or all the clients.
If a subset of the clients is involved, we can also specify if the clients should be different at each round.
At each round, the server asks the clients to start the computation and wait for the responses.
If a non-client process sends a message, the message is rejected.
The results provided by the clients are opportunely handled by the server.
If one of the clients crashes, the server will try to spawn another process in the same location.
If a certain number of client crashes in the same location, the server selects another location for spawning the client process. This location was not previously selected during the Federated Learning round.

When a fixed number of cumulative crashes occurs, the server stops and notifies the supervisor.
In case the results are provided by all clients, the server joins the results and send them to the supervisor process that will be in charge of forwarding them to the java server.
The server will then determine if the termination condition is satisfied, and this happens if the Frobenius norm has overcome the threshold or if the maximum number of rounds for an experiment are reached.
The server process will then ask the client processes to shutdown.

3.4 CLIENT PROCESS

The client process will be in charge of executing the algorithm rounds.
First, the client determines if the pyrlang process of its location is alive and if not, it restarts the pyrlang process.
We have in fact one single pyrlang process at each location and it could have been already started from another client process in that location.
Then it waits for the round execution to be requested, invoke the corresponding python function and provide back the results.
The client will shutdown if requested by the server.
A timeout value will be used to determine the shutdown of the client if no messages from the server arrives until the timeout threshold.

3.5 CORRECTNESS EVALUATION

Using the dataset https://raw.githubusercontent.com/deric/clustering-benchmark/master/src/main/resources/datasets/artificial/xclara.arff we evaluated the correctness of the results by comparing the cluster centers computed with our Federated learning algorithm with respect to the cluster centers provided by Weka.

The following results are obtained by using a epsilon tolerance equal to 0.05.

FL / Weka	**Feature 1**	**Feature 2**
Cluster 1	0.4974 / 0.4975	0.7796 / 0.7797
Cluster 2	0.7283 / 0.7284	0.2274 / 0.2274
Cluster 3	0.2507 / 0.2507	0.3912 / 0.3912

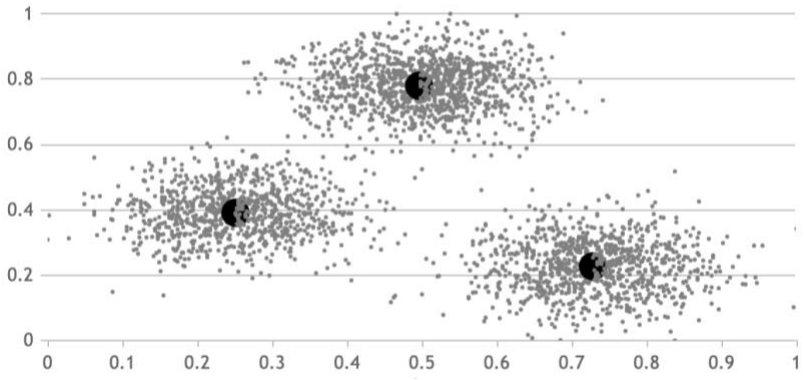


Figure 4:K-means results

3.6 SUPPORT ON MULTIPLE MACHINE LEARNING ALGORITHMS

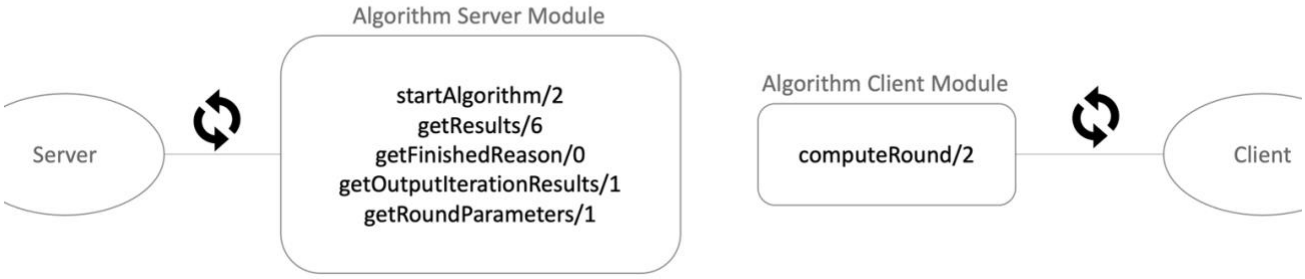
We defined our erlang modules in such a way to be independent of the specific Machine Learning Algorithm used. The supervisor, client and server's modules are written in such a way to provide general functionalities for the federated learning process.

The cMeansClient and cMeansServer erlang modules and the python files contains instead specific functions for the KMeans Algorithm.

The dynamic code loading functionality of the Erlang language can be used to support another Machine Learning algorithm without stopping the system.

The erlang modules of any Machine Learning algorithm can be deployed by complying to some specific interface constraints.

The **API** to be provided is the following:



3.6.1 SERVER MODULE

- startAlgorithm(AlgorithmSpecificParameters, NumberOfFeatures)

  The startAlgorithm/2 function will execute some initialization operations and provide the elements to be used in the first iteration.
  KMeans Implementation: The KMeans module generates initial random centers and provide the correct elements for the first iteration.

- getResults(AlgorithmSpecificParameters MaximumNumberOfRounds, NumberOfFeatures, ClientResponses, ExecutedRounds, CurrentIterationCumulativeElements)

  The getResults/6 function will take in input algorithm's specific parameters in the form of a tuple, the maximum number of rounds, the number of features, the client responses, the number of rounds executed and the cumulative values until the current iteration. The result of this function should be a tuple {NewIterationElements, Finished} with the elements of the next iteration and a variable 'Finished', equal to 0 if the reason of termination of the specific algorithm is reached.
  KMeans Implementation: The KMeans module apply the clustering algorithm exploiting the algorithm specific parameters and return the tuple {{NewCenterList, NewRoundCenters, NewFNormValue}, Finished}.

- getFinishedReason()

  The getFinishedReason/0 function returns the algorithm's reason of termination.
  KMeans Implementation: The KMeans module returns the norm_under_epsilon atom.

- getOutputIterationResults(NextIterationElements)

  The getOutputIterationResults/1 function returns the elements of the next iteration to be returned to the java server, useful for visualization.
  KMeans Implementation: The KMeans module returns the initial centers of the next iteration and the Frobenius norm value.

- getRoundParameters(NextIterationElements)

  The getRoundParameters/1 function returns the parameters to be used in the round starting from the current iteration elements.
  KMeans Implementation: The KMeans module returns the initial centers of this iteration.

3.6.2 CLIENT MODULE

- computeRound(Chunk, AlgorithmSpecificParameters)

  The computeRound/2 function run the specific algorithm's operation on the chunk and provide the results if it terminates correctly, while it determines the crash of the client if an error occurs. It returns the response to be sent to the server.
  KMeans Implementation: The KMeans module run the clustering algorithm on the specific chunk.

4. WEB APPLICATION

4.1 PROJECT ARCHITECTURE

Our web application uses a client/server structure: users can access the application and communicates with the server using a web browser.
The web application uses the three-tier architecture:

- Presentation Layer. This layer interacts with the client and it's built using JSPs, HTML, CSS and Javascript.
- Logic Layer. This layer contains the logic of the application. One part of this layer is written in Java and uses a Tomcat Server, while the other is written using Erlang.
- Data Access Layer. This layer consists of a Key/Value Repository and the corresponding DBMS.

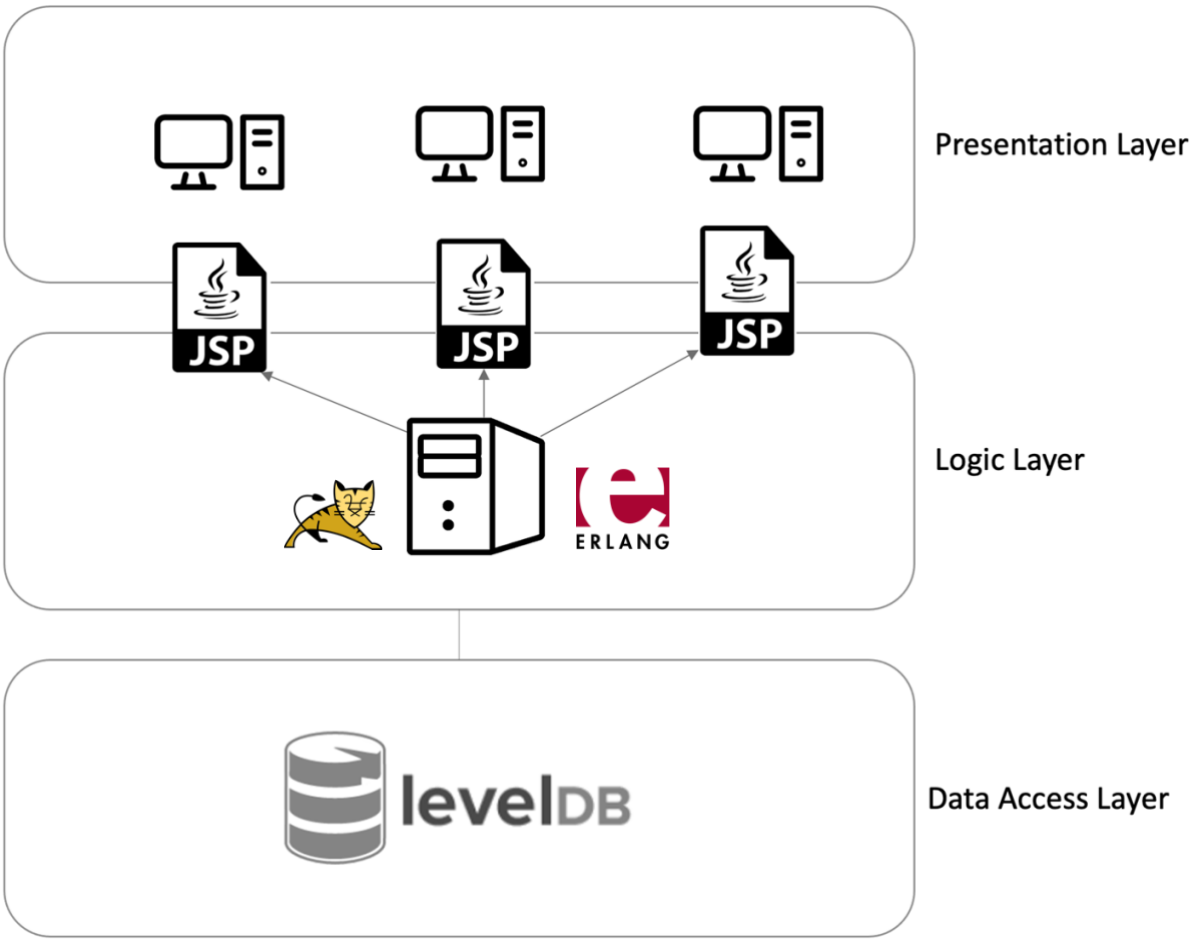


Figure 5: application architecture

4.2 PROJECT STRUCTURE

The web application is deployed on a tomcat web server, version 9.0.54. The following figure shows the folder structure of the project.
The software design pattern used is the well-known MVC (Model-View-Controller) pattern. To address the separation of concerns we used JSPs for the view part of the application meanwhile we used Filters and Servlets for the controller part.

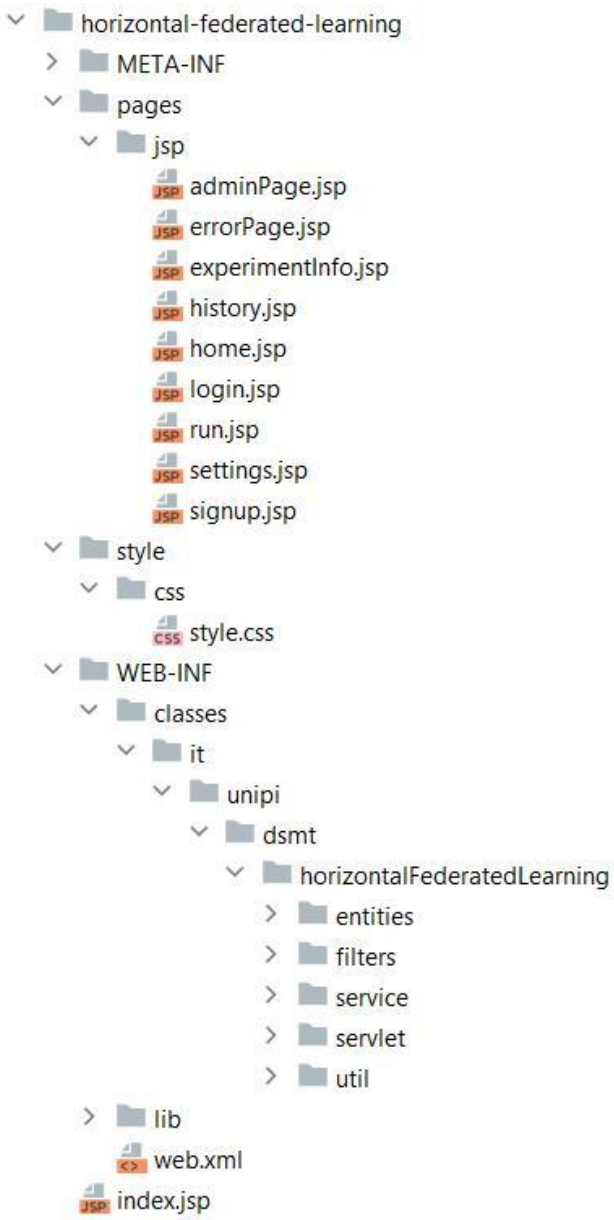


Figure 6: structure of webapp

4.3 SERVLETS AND FILTERS

The main Servlets of the application are:

- The Home servlet, which shows the home page and contains the logic to accept all the parameters coming from the client and to build a new experiment with these parameters. The Home servlet then passes the logic of the execution of the experiment to the Run servlet.
- The Run servlet, which takes a new experiment from the Home servlet and executes it exploiting the ExperimentProcess class. It then takes the results of the experiment (received from the communication layer) and builds the results to be shown by the JSP run.jsp.

The Filters are used for the authentication and validation part of the application.

4.4 SERVICES

In the services of our application we can find some services for the communication with LevelDB, a Key/Value DBMS used to store the data of the application, and the ExperimentProcess class that is in charge of enabling the communication with the Erlang layer.
To reach this, we used the Jinterface package.
Furthermore, what we needed was to be able, given the parameters of an experiment, to recreate the Erlang layer shown before that was capable to communicate with Java methods, taking in input the parameters of the experiment and returning as output the results.
To perform this operation, we used:

- An Erlang node located at server-side, that contains the relative supervisor and server processes. This node is obtained making using of the Java Runtime class and the exec method.
- A Jinterface OtpSelf class instance to send the RPC request to the Erlang node (the call to the method start of the supervisor and the parameters of the experiment). This instance is unique across the whole server and is managed through singleton pattern from different threads.
- A Jinterface OtpNode class instance to receive results and information about ongoing experiments. To be able to demultiplex information about concurrent experiments, for each experiment a new OtpMbox instance was needed (it can be seen as the Java counterpart for spawning multiple processes inside the same Erlang node). The OtpNode instance was

instead unique across the whole server and thus managed making use of singleton pattern again.

- A thread to collect the output of the Erlang node. That was necessary for logging purposes. To obtain this we exploited the return value of the Runtime.exec method, which gives us an instance of a Process. Using the method getInputStream on this instance we were able to get the output of the Erlang node (which is the collective output of supervisor, server and client processes) and write it on the log file.

4.5 SYNCHRONIZATION

In order to ensure thread-safe concurrency to our database we implemented the Singleton Pattern to ensure that all the threads use the same instance to perform operation on the database. This has been done because LevelDB manages thread-safety internally. Furthermore, the classes ExperimentService and UserService, that manages the communication with LevelDB to store data about experiments and users relatively, had static counters to keep track of the highest id in the database, so that new entities could be added to the database with the right identifier. To ensure the correct update of this static variable and avoid races between different threads, all the operations made on it are enclosed in synchronized blocks, so that a thread can read and update it in an atomic way.

Also, OtpSelf and OtpNode instances required synchronization: in particular, being these nodes unique across the whole server, in a multithreaded setting we needed to ensure that just one node of each type was created. Therefore, we managed them again with the singleton pattern, initializing them the first time they are needed and reusing them afterwards. Moreover, operations that dealt with them, such as the createMbox method of OtpNode, have been enclosed in synchronized blocks, because they operate on a shared instance, and we have no details from the documentation about the thread-safety of these instances.