Corso di Laurea Magistrale in
Computer Engineering

Federated Learning Architecture

Alessandro Sieni
Amedeo Pochiero
Roberto Magherini

Indice

Capitolo 1

Specifiche

1.1 Descrizione Problema

Il sistema da noi presentato ha come obiettivo quello di sfruttare il concetto di Federated Learning per un sistema distribuito composto da più nodi e un sink centrale. Lo scopo è quello di usare la potenza di calcolo disponibile ai bordi della rete per un'elaborazione iniziale dei dati che verranno usati per tecniche di Machine Learning. L'idea è quella di permettere ai nodi di creare un proprio modello a partire dai dati che ricevono dai propri sensori, in modo che sia successivamente inoltrato al sink che si occuperà di unire tutti i modelli ricevuti dai nodi, ottenendo così un modello più accurato, eventualmente inviato ai nodi.

1.2 Comunicazione Asincrona Nodi-Sink

I nodi dovranno comunicare con il sink in modo asincrono in quanto non è possibile stabilire a priori il momento esatto in cui i nodi stessi invieranno il modello. Questo è dovuto al fatto che la generazione del modello non avviene ad un ritmo costante ma dipende dal numero di dati ricevuti, i quali sono rilevati ad intervalli variabili.

1.3 Comunicazione Asincrona Sink-Nodi

Una volta che i modelli sono stati uniti sul sink con un algoritmo di merging apposito, si può decidere di comunicarlo ai nodi in modo che essi lo possano utilizzare sui dati che rileveranno in futuro. Anche questa comunicazione è di tipo asincrono, in quanto il merging dipende dall'arrivo asincrono di tutti i modelli sul sink. Pertanto i nodi

devono essere costantemente pronti a ricevere e a sostituire il proprio modello con quello del sink, considerato più accurato. Per fare ciò si prevede l'utilizzo di un meccanismo Publish&Subscribe in cui i nodi si comporteranno come subscribers che si abbonano al servizio offerto dall'unico publisher del sistema, vale a dire il sink.

1.4 Gestione concorrenza della ricezione dati sui nodi

I nodi sono in grado di ricevere dati o eventi da una o più sorgenti. In questo scenario è necessaria una corretta gestione della concorrenza che viene a crearsi tra le molteplici sorgenti, le quali andranno a scrivere sulla stessa destinazione. Una volta che il nodo ha ottenuto una quantità di dati sufficienti, si può procedere alla creazione o all'aggiornamento del modello. Il momento in cui la modifica del modello viene eseguita dipende non solo dalla quantità di dati ricevuti, ma anche dal valore stesso, in quanto alcuni valori potrebbero non influenzarlo mentre altri potrebbero essere abbastanza significativi da farlo variare.

1.5 Variazione del numero dei nodi

Il sistema dovrà essere in grado di poter variare il numero di nodi in tempo reale, mantenendo attiva la comunicazione con il sink centrale. Si prevede quindi l'utilizzo di tecnologie apposite con protocolli di tipo Publish&Subscribe, in cui i nodi recitano la parte dei publisher, in quanto generano i modelli da essere inviati, mentre il sink risulta essere l'unico subscriber del sistema dato che si occupa della raccolta dei modelli.

Capitolo 2

Gestione dei Dati

2.1 Concorrenza in Scrittura

I nodi ricevono dati da una o più sorgenti, andandoli a salvare all'interno della stessa *Repository*. Questo aspetto ha come problema l'accesso concorrente in scrittura.
Per risolvere questo problema è stato deciso di utilizzare un meccanismo di locking, in particolare è stato usato il Fair Lock, dato che permette di concedere il lock con una politica di tipo FIFO ai thread che devono inserire il valore generato dal sensore, in questo modo, utilizziamo i dati con lo stesso ordine in cui arrivano e inoltre evitiamo la *Starvation* dei Thread.

2.2 Implementazione

Per avere una migliore gestione della concorrenza dei Thread è stato deciso di utilizzare come linguaggio implementativo Java

2.2.1 Data Collector

Modulo usato per avviare i Thread che simulano la generazione dei dati dei sensori e avvia anche il modulo che si occupa del protocollo di comunicazione con il Sink. Si occupa anche di creare un'istanza di **RepositoryHandler**, la quale si occupa della gestione della Repository.

```
1 package it.unipi.cds.federatedLearning.node;
2
3 import it.unipi.cds.federatedLearning.Config;
4 import it.unipi.cds.federatedLearning.Log;
```

```
5
6  import java.io.IOException;
7  import java.util.concurrent.ExecutorService;
8  import java.util.concurrent.Executors;
9  import java.util.concurrent.TimeUnit;
10
11 /**
12  * This class is used to start a simulated connection with simulated sensor
        nodes
13  * in order to collect data and to start the process that communicates with
        the sink
14  * using RabbitMQ.
15  * Here are stored some constants used to decide how many values will be
       stored, how many thread
16  * (simulating the sensors) will start and the various threshold used to
       decide if the machine
17  * learning algorithm, that generates the neural network, will be started
18  *
19  */
20 public class DataCollector {
21
22   /*
23    * Read Data variables
24    */
25   public final static int THRESHOLD = Config.SIZE_WINDOW;
26   public static int newValuesThreshold = (int) (THRESHOLD*Config.
       PERCENTAGE_OLD_VALUES);
27   /*
28    * Testing variables
29    */
30   public final static int numberOfThreads = 100;
31   public final static int numberOfWrites = 10;
32
33   public static boolean aModelIsBeingGeneratedNow = false;
34   public static NodeCommunicationModelHandler nodeCommunicationHandler;
35
36     /*
37      * The executor and the array with tasks are kept in private fields
38      */
39     private ExecutorService myExecutor;
40     private DataGenerator generator;
41
42
43   public DataCollector() {
```

```
44      RepositoryHandler repository = new RepositoryHandler(THRESHOLD,
        newValuesThreshold);
45
46      myExecutor = Executors.newFixedThreadPool(numberOfThreads);
47      generator = new DataGenerator(repository, numberOfWrites, false);
48    }
49
50
51
52
53    public static void main(String[] args) throws InterruptedException {
54      try {
55        nodeCommunicationHandler = new NodeCommunicationModelHandler(args[0])
        ;
56      } catch (ArrayIndexOutOfBoundsException e) {
57        Log.error("Sink", "Provide RabbitMQ Server's ip address as argument")
        ;
58      }
59
60      DataCollector dc = new DataCollector();
61
62      for(int i = 0; i < numberOfThreads; i++) {
63        /*
64         *We use execute instead of submit because we are not interested in
        the future variable related to the thread
65         *since the Class DataGenerator is an implementation of runnable
        instead callable and we don't check when the single
66         *threads are terminated
67         */
68        dc.myExecutor.execute(dc.generator);
69      }
70      /*
71       * Waiting the termination of all threads
72       */
73      dc.myExecutor.shutdown();
74      dc.myExecutor.awaitTermination(Long.MAX_VALUE, TimeUnit.SECONDS);
75      try {
76        DataCollector.nodeCommunicationHandler.callFunction("Leave");
77      } catch (IOException | InterruptedException e) {
78        e.printStackTrace();
79      }
80      System.exit(0);
81    }
82
```

```
83 }
```

2.2.2 Data Generator

Componente con il compito di generare i dati e di scriverli all'interno della Repository, simulando il comportamento dei dati generati dai sensori. Per fare questo vengono simulate sia la frequenza di arrivo dei vari valori, ossia ogni quanto viene rilevato e ricevuto da un sensore un valore, che il valore che può assumere la risorsa rilevata dal sensore.
Nel nostro caso abbiamo simulato:

- la frequenza di arrivo dei valori come un esponenziale con una frequenza di interarrivo pari a **lambda**, per simulare questa *attesa* si utilizza la **Thread.sleep(delay)**, con delay=valore successivo della sequenza esponenziale.
- il valore delle risorse, rappresentato da una coppia di valori e per generare valori separati è stato deciso di usare due Gaussiane con valor medio una di +**Config.MEAN**, l'altra −**Config.MEAN** ed entrambe con una deviazione standard pari a **Config.ST_DEV**.

```
1  package it.unipi.cds.federatedLearning.node;
2
3  import java.util.Random;
4  import java.util.concurrent.TimeUnit;
5
6  import it.unipi.cds.federatedLearning.Config;
7
8  /**
9   * This class is used to simulate the data generated by the sensors: to
       simulate the data generated
10  * we have used two IID Gaussian distribution (seeded with two different
       seeds); to simulate the
11  * interarrivals of the data we use an Exponential distribution IID for
       each sensor with a constant mean rate
12  *
13  */
14 public class DataGenerator implements Runnable{
15
16   private RepositoryHandler repository = null;
17   /*
18    * Used to have an exponential distribution of the interarrivals of the
       data collected
```

```
19      * exponentialSeed is used to generate a uniform distribution between 0
        and 1
20      * lambda is used as the mean rate of arrival of a message
21      */
22     private Random exponentialSeed;
23     private double lambda = 0.9;
24
25     /*
26      * Used to have a fixed number of writes
27      */
28     private int numberOfWrites;
29     /*
30      * Used to have infiniteWrites
31      */
32     private Boolean infiniteWrites;
33
34     /**
35      * Constructor initializes the basic attributes and generate and create
        the exponentialSeed with a unique seed each time that is called
36      * @param repository
37      * @param numberOfWrites
38      * @param infiniteWrites
39      */
40     public DataGenerator(RepositoryHandler repository, int numberOfWrites,
        Boolean infiniteWrites) {
41       this.repository = repository;
42       this.numberOfWrites = numberOfWrites;
43       this.infiniteWrites = infiniteWrites;
44       this.exponentialSeed  = new Random();
45     }
46
47     /**
48      * Here we generate the Gaussian distributions and in a loop after the
        exponential delay it simulates the sending of the data
49      */
50     public void run(){
51       Random gaussianX = new Random();
52       Random gaussianY = new Random();
53
54       for (int i = 0; i < numberOfWrites || infiniteWrites; i++) {
55         double delay = getNextExponentialDelay() + 1.0;
56
57         try {
58           TimeUnit.SECONDS.sleep((long) delay);
```

```
59        } catch (InterruptedException ex) {
60          System.err.println(Thread.currentThread().getName() + " has been
      interrupted!");
61          Thread.currentThread().interrupt();
62        }
63        Double sensedDataX = gaussianX.nextGaussian();
64        Double sensedDataY = gaussianY.nextGaussian();
65        sensedDataX *= Config.ST_DEV;
66        sensedDataY *= Config.ST_DEV;
67        if(Thread.currentThread().getId()%2 == 0) {
68          sensedDataX += Config.MEAN;
69          sensedDataY += Config.MEAN;
70        }else {
71          sensedDataX -= Config.MEAN;
72          sensedDataY -= Config.MEAN;
73        }
74
75
76        try {
77          repository.write(sensedDataX, sensedDataY);
78        } catch (InterruptedException e) {
79          System.err.println(e.getMessage());
80        }
81      }
82    }
83
84    /**
85     * Used to generate a real exponential distribution starting from a
       Uniform distribution (exponentialSeed) with a mean value of lambda
86     * @return the simulated time necessary to the sensor to sense the data
87     */
88    public double getNextExponentialDelay() {
89        double result =  Math.log(1-exponentialSeed.nextDouble())/(-lambda);
90        if(result < 0)
91          result = -result;
92        return result;
93    }
94
95
96
97  }
```

2.2.3 Repository Handler

Modulo delegato a gestire l'accesso concorrente in scrittura alla repository, implementa il Fair Lock, e decide anche se chiamare il modulo **ModelCaller**, per richiedere la generazione di un nuovo modello. Per prendere questa decisione sono usate delle soglie **threshold** all'inizio e **newValues** successivamente, nello specifico la prima è pari ad alla dimensione di partenza della finestra di valori su cui calcolare un nuovo modello **Config.SIZE_WINDOW**, invece la seconda assume un valore variabile pari ad una percentuale (**Config.PERCENTAGE_OLD_VALUES**) del numero di campioni totali raccolti attualmente.

```
1  package it.unipi.cds.federatedLearning.node;
2
3  import it.unipi.cds.federatedLearning.Config;
4
5  import java.io.*;
6  import java.nio.file.*;
7  import java.util.*;
8  import java.util.concurrent.atomic.*;
9
10 /**
11  * This class is used as a receiver of the data generated by the simulated
       sensors, it handles
12  * the concurrency of the writes with a Fair Lock and start the ModelCaller
       to perform the REST request when it is necessary
13  *
14  */
15 public class RepositoryHandler {
16
17   /*
18    * The first time the Machine Learning algorithm will be called only if
       there are a number of values higher than a certain threshold
19    */
20   private int threshold;
21   /*
22    * Used to know how many values were stored when the ML was called the
       last time
23    */
24   private int oldNumberOfSamples = 0;
25   /*
26    * When we have a number of data higher than the threshold, ML called
       when there are a certain number of new values
27    */
```

```
28    private AtomicInteger newValues = new AtomicInteger(0);
29    /*
30     * Used to store the actual number of values
31     */
32    private static AtomicInteger numberOfSamples = new AtomicInteger(0);
33
34    /*
35     * Constants used to store the files path
36     */
37    private final String samplePath = Config.PATH_NODE_COLLECTED_DATA+
        DataCollector.nodeCommunicationHandler.getNodeID()+ ".txt";
38    private final String readySamplesPath = Config.PATH_NODE_READY_DATA+
        DataCollector.nodeCommunicationHandler.getNodeID()+ ".txt";
39
40
41    /*
42     * Variables used to manage concurrency with a FairLock implementation
43     */
44    private boolean isLocked = false;
45    private Thread lockingThread = null;
46    private List<String> waitingThreads = new ArrayList<>();
47
48    /**
49     * Constructor of the class, it sets its attributes, and check if the
        directory used to store the data exist, if not it is created, with the
        necessary files
50     * @param threshold integer used as first threshold to decide when the
        machine learning must be started
51     * @param newValues integer used after threshold, it represents the
        number of new values, from the
52     * moment the machine learning has being called, necessary to start again
         the machine learning
53     */
54    public RepositoryHandler(int threshold, int newValues) {
55      this.threshold = threshold;
56      this.newValues.set(newValues);
57      this.oldNumberOfSamples = 0;
58      try {
59          File folder = new File(Config.PATH_NODE_BASEDIR);
60          if(!folder.exists())
61            folder.mkdir();
62          File cd = new File(samplePath);
63          File rd = new File(readySamplesPath);
64          if(!cd.exists())
```

```
65            cd.createNewFile();
66          if(!rd.exists())
67            rd.createNewFile();
68      }catch(IOException e) {
69        System.out.println(e.getMessage());
70        return;
71      }
72    }
73
74    /**
75     * Lock used for the Fair Lock, if the lock is free or the thread is at
       the beginning of the queue
76     * used for the implementation of the Fair Lock, then the current thread
       can get the lock, otherwise it
77     * must wait its turn
78     * @throws InterruptedException because it is used the function Thread.
       wait()
79     */
80    public void lock() throws InterruptedException{
81
82      String activeThread = Thread.currentThread().getName();
83
84      synchronized(this) {
85        waitingThreads.add(activeThread);
86        while(isLocked || waitingThreads.get(0) != activeThread) {
87          synchronized(activeThread) {
88            try {
89              activeThread.wait();
90            }catch(InterruptedException e){
91              waitingThreads.remove(activeThread);
92              throw e;
93            }
94          }
95        }
96        waitingThreads.remove(activeThread);
97        isLocked = true;
98        lockingThread = Thread.currentThread();
99      }
100   }
101
102   /**
103    * Unlock used for the Fair Lock, here the current thread check if it has
       the lock and if it
104    * has the lock, the current thread proceeds to release it, otherwise it
```

```
     throws an IllegalMonitorStateException
105   * @throws IllegalMonitorStateException when a lock that has not get the
      lock tries to release it
106   */
107  public void unlock() {
108    if(!isLocked || this.lockingThread != Thread.currentThread()) {
109      throw new IllegalMonitorStateException("Calling thread has not locked
      this lock");
110    }
111    this.isLocked = false;
112    lockingThread = null;
113    if(waitingThreads.size() > 0) {
114      String sleepingThread = waitingThreads.get(0);
115      synchronized (sleepingThread) {
116        sleepingThread.notify();
117      }
118    }
119  }
120
121  /**
122   * Function used to simulate the reception of the data from a sensor node
      . Here we use
123   * the Fair Lock to handle the concurrency of the write operations in a
      single file.
124   * If the number of values written is above the threshold then we call
      the read.
125   * @param sensedDataX first value sensed of the pair of value sensed
126   * @param sensedDataY second value sensed of the pair of value sensed
127   * @throws InterruptedException because of the lock function
128   */
129  public void write(Double sensedDataX, Double sensedDataY) throws
      InterruptedException{
130
131    this.lock();
132
133    int instantNumberOfSamples = numberOfSamples.incrementAndGet();
134
135    try(
136      FileWriter fw = new FileWriter(samplePath, true);
137        )
138    {
139      fw.append(sensedDataX.toString() + "," + sensedDataY.toString() + "\n
      ");
140    }catch(IOException e) {
```

```
141         System.out.println(e.getMessage());
142         return;
143       }
144
145       if(!DataCollector.aModelIsBeingGeneratedNow) {
146         int value = newValues.get();
147         if((oldNumberOfSamples == 0 && instantNumberOfSamples >= threshold)
148             || (oldNumberOfSamples > 0 && instantNumberOfSamples -
      oldNumberOfSamples >= value)
149         ){
150           DataCollector.aModelIsBeingGeneratedNow = true;
151           if(oldNumberOfSamples == 0) {
152             this.read(threshold);
153           }
154           else {
155             this.read(value);
156           }
157           oldNumberOfSamples = instantNumberOfSamples;
158         }
159       }
160       this.unlock();
161     }
162
163     /**
164      * Function used to read and move the value stored from the sensors to
       another file that will be used
165      * by the machine learning algorithm, this is made to avoid the fact that
        new values can arrive while the
166      * machine learning algorithm is working, and for all the new value we
       must check before starting the algorithm
167      * if it is an outlier or not. After moving the data it starts
       ModelCaller
168      * @param valuesToRead used to specify the actual number of values inside
        the file used to store the data from the sensors
169      *
170      */
171     private void read(int valuesToRead){
172       try(FileWriter fw = new FileWriter(readySamplesPath, true);
173           ){
174
175         String readyData = new String (Files.readAllBytes(Paths.get(
      samplePath)));
176
177         if(oldNumberOfSamples == 0)
```

```
178             fw.write(readyData);
179           else
180             fw.append(readyData);
181           /*
182            * Call the function to send the synchronous REST request
183            */
184           Runnable caller = new ModelCaller(valuesToRead);
185           new Thread(caller).start();
186           newValues.set((int) (numberOfSamples.get()*Config.
      PERCENTAGE_OLD_VALUES));
187           //We flush the file with the sample not ready to reduce redundancy of
       the data
188           try(FileWriter fw2 = new FileWriter(samplePath);){
189             fw2.write("");
190           }catch(IOException ex) {
191             System.out.println(ex.getMessage());
192           }
193
194         }catch(IOException e) {
195           System.out.println(e.getMessage());
196         }
197
198       }
199     }
```

2.2.4 Model Caller

Modulo incaricato di richiedere la creazione del nuovo modello, tramite una chiamata sincrona di tipo REST. Si occupa di decidere sia la dimensione della finestra dei valori su cui ricalcolare il modello, che del *coefficiente di Fading*. A questo punto possiamo distinguere due casi:

1. Numero di nuovi valori inseriti è inferiore alla dimensione di partenza della finestra: La dimensione della finestra sarà pari a **Config.SIZE_WINDOW** e il *coefficiente di fading* sarà pari a 0

2. Numero di nuovi valori inseriti maggiore della dimensione di partenza della finestra: La dimensione della finestra sarà pari al numero di valori nuovi inseriti ed entrerà in gioco il *coefficiente di Fading* che assumerà un valore compreso in un intervallo tra $[0, 1]$, col quale verranno presi in considerazione una percentuale di valori al di fuori della finestra **fadingCoefficent** $= \frac{(nuovi\_valori\_inseriti - Config.SIZE\_WINDOW)}{nuovi\_valori\_inseriti}$

```
1  package it.unipi.cds.federatedLearning.node;
2
3  import com.mashape.unirest.http.HttpResponse;
4  import com.mashape.unirest.http.Unirest;
5  import com.mashape.unirest.http.exceptions.UnirestException;
6
7  import it.unipi.cds.federatedLearning.Config;
8  import it.unipi.cds.federatedLearning.Log;
9
10 /**
11  * This class is used to call the machine learning with a REST call
12  *
13  */
14 public class ModelCaller implements Runnable{
15
16   int values;
17
18   /**
19    * Constructor
20    * @param valuesToRead the number of new values present in the file used
       by the machine learning algorithm
21    */
22   public ModelCaller(int valuesToRead) {
23     values = valuesToRead;
24   }
25
26   /**
27    * Start a synchronous REST call to start the machine learning algorithm
       used to Train a neural network and waits for the response
28    * of the rest server, check the response if it is 201 or 204 there are
       no problem.
29    */
30   @Override
31   public void run() {
32     try {
33
34       /*
35        * Handle the varible window size and the fading coefficent
36        */
37       double fadingCoefficent = 0;
38       int sizeWindow = Config.SIZE_WINDOW;
39       if(Config.SIZE_WINDOW < values) {
40         fadingCoefficent = ((double)(values - Config.SIZE_WINDOW))/(double)
```

```
    (values);
41          sizeWindow = values;
42        }
43
44        HttpResponse<String> response = Unirest.post("http://127.0.0.1:5000/
    server")
45            .header("content-type", "application/json")
46            .body("{\n\t\"ID\":\""+DataCollector.nodeCommunicationHandler.
    getNodeID()+
47                "\",\n\t\"command\":\"Train\""
48                + ",\n\t\"values\":\""+values+"\","
49                + "\n\t\"Window\": \""+sizeWindow+"\""
50                + ",\n\t\"Coeff\": \""+fadingCoefficent+"\"\n}")
51            .asString();
52
53        Log.info("Train", "values " + values + " - sizeWindow " + sizeWindow
    + " - fading coefficent " +  fadingCoefficent);
54
55        DataCollector.aModelIsBeingGeneratedNow = true;
56        switch (response.getStatus()) {
57        case 201:
58          /*
59           * Model created
60           */
61          DataCollector.nodeCommunicationHandler.sendModel();
62          break;
63
64        case 204:
65          /*
66           * Correct response from the rest server but model not updated
67           */
68          Log.info("Model Caller", "No need to send the new model");
69          break;
70        default:
71          Log.info("Model Caller", "REST SERVER PROBLEM - STATUS:" + response
    .getStatus() + "!!");
72          return;
73        }
74        DataCollector.aModelIsBeingGeneratedNow = false;
75      } catch (UnirestException e) {
76        e.printStackTrace();
77      }
78    }
79
```

```
80 }
```

Capitolo 3

Comunicazione tra i Nodi e il Sink

3.1 Descrizione del Problema

L'architettura presentata è stata sviluppata con l'obiettivo di gestire un alto grado di asincronismo tra le varie parti del sistema. L'istante di trasmissione dei dati, infatti, è imprevedibile e può avvenire in qualsiasi momento, ciò è dovuto al fatto che l'arrivo dei dati sui nodi non è periodico e, di conseguenza, anche la creazione del relativo modello. Tutto ciò rende imprevedibile anche l'istante in cui i modelli vengono unificati dal Sink, il quale necessita dei modelli di tutti i nodi per effettuare il *Merging*.

Per rispettare questi requisiti è necessario addottare tecniche di comunicazioni asincrone, in grado di gestire la trasmissione e l'arrivo dei dati in un qualunque istante nel tempo. Un protocollo che si presta bene a queste esigenze è il Protocollo **AMQP** (*Advanced Message Queueing Protocol*), il quale garantisce funzionalità di messaggistica, accodamento, routing (con paradigmi Point-to-Point e Public&Subscribe).

Nel caso considerato, la comunicazione è bidirezionale in quanto da un lato i nodi inviano i modelli locali al Sink, dall'altro il Sink invia il modello Unificato a tutti i nodi, i quali aggiornano il proprio modello locale con quello Unificato, se considerato più accurato. Visto che entrambe le parti interpretano il ruolo sia di mittente che di destinatario, si è scelto di gestire le due comunicazioni con tecniche differenti, come spiegato nei paragrafi seguenti.

3.1.1 Comunicazione da Nodo a Sink

Un modello è prodotto da un Nodo (*Producer*) e inviato al Sink (*Consumer*), che rappresenta l'unico ricevitore. In uno scenario del genere è sufficiente adottare una comunicazione del tipo mostrata in figura **??** nella quale un produttore invia i propri dati sulla coda, successivamente consumati dal consumatore. Questa tipologia di comunicazione è usata da ogni Nodo, quindi, dal punto di vista del Sink, ci saranno N produttori che riempiranno la sua coda.

Figura 3.1: Comunicazione con un Producer e un Consumer

3.1.2 Comunicazione da Sink a Nodo

Il Sink produce il modello Unificato che deve essere distribuito all'intera pool di Nodi. Il paradigma di comunicazione migliore da seguire in questo caso è quello del *Public&Subscribe*, rappresentato in figura **??**. Tramite questo paradigma il Sink comunica ad un *Exchange* (il nodo in blu della figura) il modello e questo lo posta su ogni coda di ogni consumatore (i Nodi).

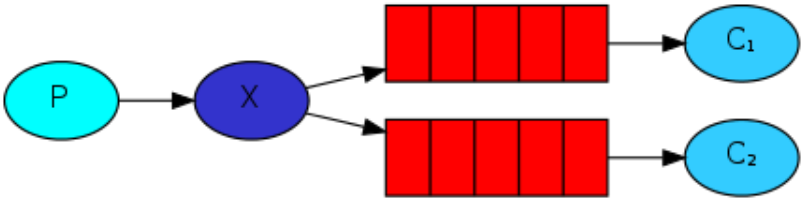

Figura 3.2: Comunicazione Public&Subscribe

3.1.3 Registrazione di un nuovo Nodo

Per rendere l'architettura dinamica e flessibile è stato sviluppato un meccanismo per incrementare o diminuire il numero dei Nodi nel sistema. A questo scopo, quando un nodo vuole unirsi alla pool, deve necessariamente informare il Sink in modo tale che quest'ultimo tenga costantemente traccia del numero di Nodi presenti, necessario per l'operazione di Merging dei modelli.
La soluzione migliore in questo caso è quella di sfruttare un sistema di **RPC** (*Remote*

Procedure Call) che permette al Nodo di effettuare delle chiamate di funzioni eseguite sul Sink. Come mostrato in figura **??**, questo meccanismo si compone di 2 code:

- *rpc_queue*: coda delle chiamate di funzioni
- *reply_to*: coda dei risultati delle funzioni

Nel caso considerato, le funzioni disponibili sono le seguenti:

- *Registration*: un nodo richiede l'accesso alla pool e ottiene un identificativo univoco come risposta
- *Leave*: un nodo richiede l'uscita dalla pool specificando il proprio identificativo nella richiesta

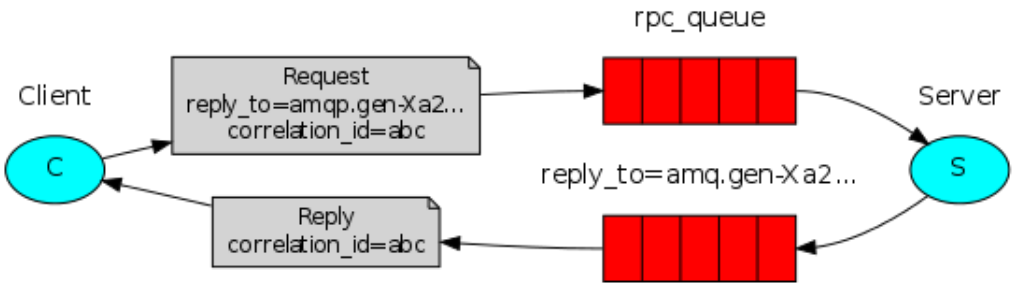

Figura 3.3: Remote Procedure Call

3.2 Implementazione

3.2.1 abstract class CommunicationModelHandler

Il software creato si appoggia sulle Java API di *RabbitMQ* che permette una facile gestione di message queueing e per via delle somiglianza tra le azioni da svolgere sia sul Sink che sui Nodi, è stata implementata una classe astratta *CommunicationModelHandler* **??** che mantiene delle informazioni utilizzati nelle interazioni con il server di RabbitMQ e i nomi delle *Queue*, comuni a tutti i nodi. Inoltre, il costruttore inizializza una connessione con il Server di RabbitMQ configurato sulla porta di default 5672 e richiama le 3 funzioni per l'inizializzazione delle strutture su cui verrano scambiati i dati:

- *initRPC()*

- *initSinkToNode()*
- *initNodeToSink()*

Ognuna di queste verrà definita dal Nodo e dal Sink in modo da rispettare il proprio ruolo rispetto alla struttura in questione.

```java
package it.unipi.cds.federatedLearning;

import com.rabbitmq.client.Channel;
import com.rabbitmq.client.ConnectionFactory;

public abstract class CommunicationModelHandler {

    protected final String RPC_NODE_TO_SINK_QUEUE_NAME = "RPC_QUEUE";
    protected final String NODE_TO_SINK_QUEUE_NAME = "MODELS_QUEUE";
    protected final String SINK_TO_NODE_EXCHANGE_NAME = "NEW_MODEL_QUEUE";

    protected ConnectionFactory factory;
    protected Channel channelNodeSink;
    protected Channel channelSinkNode;
    protected Channel channelRPC;
    protected ModelReceiver receiver;

    public CommunicationModelHandler(String hostname) {
        this.factory = new ConnectionFactory();
        factory.setHost(hostname);
        factory.setVirtualHost("cds/");
        factory.setUsername("cdsAdmin");
        factory.setPassword("cds");

        initRPC();
        initSinkToNode();
        initNodeToSink();
    }
    protected abstract void initNodeToSink();
    protected abstract void initSinkToNode();
    protected abstract void initRPC();
    public abstract void receiveModel(Model deliveredModel);
    public abstract void sendModel();
}
```

Listing 3.1: CommunicationModelHandler

3.2.2 NodeCommunicationModelHandler

Questa classe è l'implementazione della *CommunicationModelHandler* utilizzata su ogni Nodo per la gestione della communicazione con il Sink. Ai campi membri della super classe viene aggiunto un intero *NodeID* che contiene l'identificativo del nodo. La classe definisce le funzioni astratte nel seguente modo:

3.2.2.1 initRPC()

Questa funzione crea un canale per comunicare con il Server RPC presente sul Sink attraverso cui si chiama la Remote Procedure per «registrare» il Nodo nel sistema, il quale ottiene un identificativo univoco come risposta. Tale ID è usato per distinguere i nodi tra di loro e i loro relativi modelli.

```
1     @Override
2     protected void initRPC() {
3         try {
4             Connection connectionRPC = factory.newConnection();
5             channelRPC = connectionRPC.createChannel();
6             Log.info("NodeCommunicationHandler", "Creating RPC Client");
7
8             nodeID = Integer.parseInt(callFunction("Registration"));
9         } catch (IOException | TimeoutException | InterruptedException e) {
10            Log.error("Node", e.toString());
11        }
12    }
```

Listing 3.2: NodeCommunicationModelHandler.initRPC()

3.2.2.2 initNodeToSink()

É sufficiente dichiarare sul canale opportuno la coda del Sink sulla quale riceve i modelli, nominata **NODE_TO_SINK_QUEUE**, di tipo:

- non *durable*: la coda non sopravvive ad un restart del server
- non *exclusive*: non ristretta a questa connessione
- non *autoDelete*: il server non cancellerà la coda se non più in uso

```
1     @Override
2     protected void initNodeToSink() {
3         try {
```

```
4             Connection connectionNodeSink = factory.newConnection();
5             channelNodeSink = connectionNodeSink.createChannel();
6             Log.info("Node-" + nodeID, "Declaring NODE_TO_SINK_QUEUE");
7             channelNodeSink.queueDeclare(NODE_TO_SINK_QUEUE_NAME, false,
    false, false, null);
8         } catch (IOException | TimeoutException e) {
9             Log.error("Node-" + nodeID, e.toString());
10        }
11    }
```

Listing 3.3: NodeCommunicationModelHandler.initNodeToSink()

3.2.2.3 initSinkToNode()

In questo caso bisogna legare il nodo all'*Exchange* del Sink in modo da ricevere qualsiasi modello Unificato da esso prodotto. A questo scopo, si dichiara l'Exchange di nome **SINK_TO_NODE_EXCHANGE** di tipo *fanout*, che effettua il broadcast di tutti i messaggi che riceve a tutte le code che conosce, e da questo si ottiene una coda temporanea sulla quale il nodo riceve i messaggi. Eseguendo la *basicConsume(...)* il nodo inizia ad ascoltare i messaggi sulla coda che vengono processati dal *Consumer ModelReceiver* di cui si parla in **??**.

```
1     @Override
2     public void initSinkToNode() {
3         try {
4             Connection connectionSinkNode = factory.newConnection();
5             channelSinkNode = connectionSinkNode.createChannel();
6
7             Log.info("Node-" + nodeID, "Declaring SINK_TO_NODE_EXCHANGE");
8             channelSinkNode.exchangeDeclare(SINK_TO_NODE_EXCHANGE_NAME, "
    fanout", true);
9
10            String queueName = channelSinkNode.queueDeclare().getQueue();
11            channelSinkNode.queueBind(queueName, SINK_TO_NODE_EXCHANGE_NAME
    , "");
12
13            receiver = new ModelReceiver(this);
14            Log.info("Node-" + nodeID, "Starting consuming from " +
    queueName);
15            channelSinkNode.basicConsume(queueName, true, receiver);
16        } catch (TimeoutException | IOException e) {
17            Log.error("Node-" + nodeID, e.toString());
18        }
```

```
19      }
```

Listing 3.4: NodeCommunicationModelHandler.initSinkToNode()

3.2.2.4 receiveModel(Model deliveredModel)

Alla ricezione di un nuovo modello Unificato, il nodo deve comunicare al modulo del Machine Learning che è necessario valutare l'aggiornamento del modello locale con quello appena ricevuto. Si contatta il server Rest con il comando *Update*, specificando inoltre l'ID del nodo, attivando un thread apposito per evitare attese attive sulla risposta del server Rest.

```
1   @Override
2   public void receiveModel(Model deliveredModel) {
3       Log.info("Node-" + nodeID, "Updated model received");
4       deliveredModel.toFile(Config.PATH_NODE_UPDATED_MODEL);
5       Runnable notifier = () -> {
6           try {
7               Unirest.post("http://127.0.0.1:5000/server")
8                       .header("content-type", "application/json")
9                       .body("{\n\t\"command\":\"Update\",\n\t\"ID\":\""+
    nodeID +"\"\n}")
10                      .asString();
11          } catch (UnirestException e) {
12              Log.error("Node-" + nodeID, e.toString());
13          }
14      };
15      notifier.run();
16  }
```

Listing 3.5: NodeCommunicationModelHandler.receiveModel()

3.2.2.5 sendModel()

Per inviare un modello al Sink è sufficiente leggerlo dal file creato dal modulo di ML ed effettuare la funzione *basicPublish(...)* specificando la coda in cui inserire il modello e i byte da inviare.

```
1   @Override
2   public void sendModel() {
3       try {
4           Model model = new Model(nodeID, new String(Files.readAllBytes(
    Paths.get(Config.PATH_NODE_NEW_MODEL+nodeID+".json"))));
```

```
5             Log.info("Node-" + nodeID, "Publishing " + model.toString() + "
      on NODE_TO_SINK_QUEUE");
6             channelNodeSink.basicPublish("", NODE_TO_SINK_QUEUE_NAME, null,
      model.getBytes());
7         } catch (IOException e) {
8             Log.error("Node-" + nodeID, e.toString());
9         }
10    }
```

Listing 3.6: NodeCommunicationModelHandler.sendModel()

3.2.3 SinkCommunicationModelHandler

Questa classe è un'estensione della *CommunicationModelHandler* utilizzata dal Sink per la gestione della comunicazione con i Nodi. Ai campi membri della super classe, si aggiungono:

- *int nextID*: un intero incrementale che indica l'ID da assegnare al prossimo nodo che si unisce al sistema
- *HashMap<Integer,Boolean>* isNew: che permette di verificare se il Sink ha ricevuto un modello fresco da un determinato nodo
- *boolean merging*: indica se è in corso il merging o meno

3.2.3.1 initRPC()

Avvia il thread relativo a *RPCServer* di cui si parla in **??**.

```
1     @Override
2     protected void initRPC() {
3         Thread rpcServer = new RPCServer(this, channelRPC, factory.getHost
      ());
4         rpcServer.start();
5     }
```

Listing 3.7: SinkCommunicationModelHandler.initRPC()

3.2.3.2 initNodeToSink()

Si dichiara una coda come in **??** ma questa volta si esegue la *basicConsume(...)* per stare in ascolto dei modelli inviati dai nodi

```
1     @Override
2     protected void initNodeToSink() {
3         try {
4             Connection connectionNodeSink = factory.newConnection();
5             channelNodeSink = connectionNodeSink.createChannel();
6             channelNodeSink.queueDeclare(NODE_TO_SINK_QUEUE_NAME, false,
      false, false, null);
7             System.out.println("Waiting for models");
8
9             receiver = new ModelReceiver(this);
10            channelNodeSink.basicConsume(NODE_TO_SINK_QUEUE_NAME, true,
      receiver);
11        } catch (IOException | TimeoutException e) {
12            e.printStackTrace();
13        }
14    }
```

Listing 3.8: SinkCommunicationModelHandler.initNodeToSink()

3.2.3.3 initSinkToNode()

Si dichiara l'*Exchange* come in **??**, tuttavia senza avviare il consumo dei messaggi.

```
1     @Override
2     protected void initSinkToNode() {
3         try {
4             Connection connectionSinkNode = factory.newConnection();
5             channelSinkNode = connectionSinkNode.createChannel();
6             channelSinkNode.exchangeDeclare(SINK_TO_NODE_EXCHANGE_NAME, "
      fanout", true);
7             Log.info("SinkCommunicationModelHandler","Declaring
      SINK_TO_NODE_EXCHANGE");
8         } catch (TimeoutException | IOException e) {
9             e.printStackTrace();
10        }
11    }
```

Listing 3.9: SinkCommunicationModelHandler.initSinkToNode()

3.2.3.4 receiveModel(Model deliveredModel)

Alla ricezione di un nuovo modello questo viene conservato in 2 file:

- *dataSink/ModelNodeID.json* dove ID indica l'id del nodo che ha inviato il modello

- *dataSink/history/Node-ID/* e come nome del file si usa il timestamp di ricezione del modello. In questa cartella è conservato uno storico dei modelli inviati da nodo con quel determinato ID.

Si aggiorna la freschezza del modello nella mappa isNew e si controlla se avviare il processo di merging o meno, avviato solo se i modelli di tutti i nodi risultano essere freschi e se non vi è un merging già in corso. In caso positivo, si avvia il thread *ModelMerger* di cui si parla in **??**.

```
1    @Override
2    public void receiveModel(Model deliveredModel) {
3        deliveredModel.toFile(Config.PATH_SINK_RECEIVED_MODELS +
     deliveredModel.getNodeID() + ".json");
4        deliveredModel.toFile( Config.PATH_SINK_HISTORY + "Node-" +
     deliveredModel.getNodeID() + "/" + new SimpleDateFormat("yyyy-MM-dd HH.
     mm.ss").format(new Date()) + ".json" );
5        isNew.replace(deliveredModel.getNodeID(), true);
6
7        if (areAllNew() && !merging) {
8            // Start the merging of the models
9            merging = true;
10           ModelMerger mm = new ModelMerger(isNew.size(), this);
11           mm.start();
12       }
13   }
```

Listing 3.10: SinkCommunicationModelHandler.receiveModel(...)

3.2.3.5 sendModel()

Legge il modello Unificato creato dal modulo ML e lo pubblica sull'*Exchange* **SINK_TO_NODE_EXCHANGE**, resetta la freschezza di tutti i modelli nella mappa isNew e dichiara concluso il merging.

```
1    @Override
2    public void sendModel() {
3        try {
4            Model model = new Model(-1, new String ( Files.readAllBytes(
     Paths.get(Config.PATH_SINK_MERGED_MODEL))));
5            Log.info("Sink","Publishing " + model.toString() + " on
     SINK_TO_NODE_EXCHANGE");
6            channelSinkNode.basicPublish(SINK_TO_NODE_EXCHANGE_NAME, "",
     null, model.getBytes());
```

```
7
8               isNew.replaceAll((key, oldValue) -> Boolean.FALSE);
9
10              merging = false;
11          } catch (IOException e) {
12              e.printStackTrace();
13          }
14      }
```

Listing 3.11: SinkCommunicationModelHandler.sendModel()

3.2.3.6 registration()

Funzione chiamata da RPCServer quando quest'ultimo riceve una richiesta di registrazione da un nodo. La risposta alla richiesta contiene il valore ritornato da questa funzione, corrispondente all'ID assegnato al nodo entrante.

```
1       public String registration() {
2           isNew.put(nextID,false);
3           updateFileSizePool();
4           String nodeID = String.valueOf(nextID);
5           Log.info("Sink", "New node requesting registration. New node id: "
    + nodeID + " | Current nodes: " + isNew.size());
6           nextID++;
7           return nodeID;
8       }
```

Listing 3.12: SinkCommunicationModelHandler.registration()

3.2.3.7 removeNode(int nodeID)

Funzione chiamata da RPCServer quando riceve una richiesta di uscita dal sistema dal nodo. Rimuove il nodo con l'ID specificato dalla mappa isNew e ritorna **OK** se l'operazione è avvenuta con successo, altrimenti **NOT FOUND**.

```
1       public String removeNode(int nodeID) {
2           if (isNew.containsKey(nodeID)) {
3               isNew.remove(nodeID);
4               updateFileSizePool();
5               Log.info("Sink", "Node "+ nodeID + " is leaving. Current nodes:
     " + isNew.size());
6               return "OK";
7           }
```

```
8           Log.error("Sink", "Node "+ nodeID + " not found. Current nodes: " +
        isNew.size());
9           return "NOT FOUND";
10      }
```

Listing 3.13: SinkCommunicationModelHandler.removeNode()

3.2.4 ModelMerger

Effettua una richiesta al server Rest con il comando *Merge* e sta in attesa della risposta. La risposta può contenere i seguenti codici:

- 201: il modello è stato unificato con successo, quindi è possibile inviarlo ai nodi tramite la *SinkCommunicationModelHandler.sendModel()*
- 204: attualmente esiste solo un modello, quindi è inutile effettuare il merging
- 500: si è verificato un errore nel modulo ML
- altrimenti: codice non supportato

```
1  /**
2   * ModelMerger waits for the RestServer's response and the send a new model
        to nodes
3   */
4  public class ModelMerger extends Thread {
5
6       private int nodes;
7       private SinkCommunicationModelHandler sc;
8
9       /**
10       * @param nodes current number of nodes
11       * @param sc instance of a SinkCommunicationModelHandler
12       */
13      ModelMerger(int nodes, SinkCommunicationModelHandler sc) {
14          this.nodes = nodes;
15          this.sc = sc;
16      }
17
18      /**
19       * Post a request to the RestServer with command Merge and wait for the
        response. If the code status is CREATED,
20       * then send the merged model to the nodes
21       */
```

```
22      @Override
23      public void run() {
24          try {
25              Log.info("ModelMerger", "Model merger started");
26              HttpResponse<String> response = Unirest.post("http
    ://127.0.0.1:5000/server")
27                      .header("content-type", "application/json")
28                      .body("{\n\t\"command\":\"Merge\"\n}")
29                      .asString();
30              switch (response.getStatus()) {
31                  case 201:
32                      sc.sendModel();
33                      Log.info("ModelMerger", "Model merger started");
34                      break;
```

Listing 3.14: ModelMerger

3.2.5 RPCServer

RPCServer è un thread che sta continuamente in ascolto di richeste dei nodi in arrivo sulla coda **RPC_NODE_TO_SINK_QUEUE_NAME**. Le richieste vengono processate dal *DeliverCallback*, definito nella funzione run del thread, che richiama l'oppurtuna funzione del Sink (**??** oppure **??**) e risponde alla richiesta tramite la *basicPublish(...)*.

```
1  public class RPCServer extends Thread {
2
3      private Channel channelRPC;
4      private SinkCommunicationModelHandler sink;
5      private ConnectionFactory factory;
6
7      RPCServer(SinkCommunicationModelHandler sink, Channel channelRPC,
       String hostname) {
8          this.sink = sink;
9          this.channelRPC = channelRPC;
10
11         this.factory = new ConnectionFactory();
12         factory.setHost(hostname);
13         factory.setVirtualHost("cds/");
14         factory.setUsername("cdsAdmin");
15         factory.setPassword("cds");
16     }
17
```

```
18      public void run() {
19          try (Connection connectionRPC = factory.newConnection()) {
20              channelRPC = connectionRPC.createChannel();
21
22              Log.info("RPCServer", "Starting RPC Server");
23              String RPC_NODE_TO_SINK_QUEUE_NAME = "RPC_QUEUE";
24              channelRPC.queueDeclare(RPC_NODE_TO_SINK_QUEUE_NAME, false,
    false, false, null);
25              channelRPC.queuePurge(RPC_NODE_TO_SINK_QUEUE_NAME);
26              channelRPC.basicQos(1);
27              Object monitor = new Object();
28              DeliverCallback deliverCallback = (consumerTag, delivery) -> {
29                  AMQP.BasicProperties replyProps = new AMQP.BasicProperties
30                          .Builder()
31                          .correlationId(delivery.getProperties().
    getCorrelationId())
32                          .build();
33
34                  String response = "";
35
36                  try {
37                      String message = new String(delivery.getBody(),
    StandardCharsets.UTF_8);
38                      if (message.equals("Registration")) {
39                          response += sink.registration();
40                      } else if (message.startsWith("Leave")) {
41                          int nodeID = Integer.parseInt(message.split(":")
    [1]);
42                          response += sink.removeNode(nodeID);
43                      }
44
45
46                  } catch (RuntimeException e) {
47                      e.printStackTrace();
48                  } finally {
49                      channelRPC.basicPublish("", delivery.getProperties().
    getReplyTo(), replyProps, response.getBytes(StandardCharsets.UTF_8));
50                      channelRPC.basicAck(delivery.getEnvelope().
    getDeliveryTag(), false);
51                      // RabbitMq consumer worker thread notifies the RPC
    server owner thread
52                      synchronized (monitor) {
53                          monitor.notify();
54                      }
```

```
55                  }
56              };
57              Log.info("RPCServer", "Starting consuming from RPC_QUEUE");
58              channelRPC.basicConsume(RPC_NODE_TO_SINK_QUEUE_NAME, false,
    deliverCallback, (consumerTag -> {
59              }));
60
61              while (true) {
62                  synchronized (monitor) {
63                      try {
64                          monitor.wait();
65                      } catch (InterruptedException e) {
66                          e.printStackTrace();
67                      }
68                  }
69              }
70          } catch (TimeoutException | IOException e) {
71              e.printStackTrace();
72          }
73      }
74  }
```

Listing 3.15: ModelMerger

3.2.6 Classi comuni tra Nodo e Sink

3.2.6.1 Model

Contiene l'ID del nodo che ha generato il modello e una stringa che contiene la rappresentazione json del modello. In caso in cui il modello sia quello unificato prodotto dal Sink, il nodeID contiene -1.

```
1  public class Model implements Serializable {
2
3      private int nodeID;
4      private String json;
5
6      public Model(int id, String json) {
7          this.nodeID = id;
8          this.json = json;
9      }
10     Model(byte[] rawData) {
11         Object obj = null;
12         try (ByteArrayInputStream bis = new ByteArrayInputStream(rawData);
```

```
13              ObjectInputStream ois = new ObjectInputStream(bis)){
14              obj = ois.readObject();
15          } catch (IOException | ClassNotFoundException e) {
16              Log.error("Model", e.toString());
17          }
18          Model m = (Model) obj;
19          if (m != null) {
20              this.nodeID = m.getNodeID();
21              this.json = m.getJson();
22          }
23      }
24      public String getJson() {
25          return json;
26      }
27      public void setJson(String json) {
28          this.json = json;
29      }
30      public int getNodeID() {
31          return nodeID;
32      }
33      public void setNodeID(int nodeID) {
34          this.nodeID = nodeID;
35      }
36      public byte[] getBytes() {
37          byte[] data = null;
38          try (ByteArrayOutputStream bos = new ByteArrayOutputStream();
39               ObjectOutputStream oos = new ObjectOutputStream(bos)){
40              oos.writeObject(this);
41              oos.flush();
42              data = bos.toByteArray();
43          } catch (IOException e) {
44              e.printStackTrace();
45          }
46          return data;
47      }
48      public void toFile(String filename) {
49          File file = new File(filename);
50          file.getParentFile().mkdirs();
51          try (BufferedWriter writer = new BufferedWriter(new FileWriter(
    filename))) {
52              writer.write(json);
53          } catch (IOException e) {
54              Log.error("Model", e.toString());
55          }
```

```
56      }
57      @Override
58      public String toString() {
59          if ( this.nodeID == -1 )
60              return "UpdatedModel";
61          return "Model-" + nodeID;
62      }
63  }
```

Listing 3.16: Model

3.2.6.2 ModelReceiver

Questa classe implementa l'interfaccia Consumer di RabbitMQ e permette di definire le azioni da compiere in base all'evento ricevuto. In particolare, alla ricezione di un nuovo modello, viene chiamata la funzione *receiveModel* del relativo *CommunicationModelHandler*.

```
1  public class ModelReceiver implements Consumer {
2
3      private CommunicationModelHandler handler;
4      public ModelReceiver(CommunicationModelHandler handler) {
5          this.handler = handler;
6      }
7      public void handleConsumeOk(String s) {
8          Log.info("MordelReceiver", "Starting consuming");
9      }
10     public void handleCancelOk(String s) {
11         Log.info("MordelReceiver", "Consumer cancelled in the correct way:
       " + s );
12     }
13     public void handleCancel(String s) throws IOException {
14         Log.info("MordelReceiver", "Consumer cancelled in an incorrect way:
        " + s );
15     }
16     public void handleShutdownSignal(String s, ShutdownSignalException e) {
17         Log.error("MordelReceiver", "Either the channel or the underlying
       connection has been shut down: " + s );
18     }
19     public void handleRecoverOk(String s) {
20         Log.info("MordelReceiver", "Recovery ok: " + s );
21     }
22     public void handleDelivery(String s, Envelope envelope, AMQP.
       BasicProperties basicProperties, byte[] bytes) throws IOException {
```

```
23          Model deliveredModel = new Model(bytes);
24          int nodeID = deliveredModel.getNodeID();
25          Log.info("MordelReceiver", "New model delivered from : " + (nodeID
    ==-1? "Sink": nodeID) );
26          handler.receiveModel(deliveredModel);
27      }
28  }
```

Listing 3.17: ModelReceiver

3.2.6.3 Config

Classe che contiene le varie costanti usate nel progetto.

```
1  public class Config {
2      public static final String PATH_SINK_RECEIVED_MODELS = "dataSink/
       ModelNode";
3      public static final String PATH_SINK_MERGED_MODEL = "dataSink/
       MergedModel.json";
4      public static final String PATH_SINK_HISTORY = "dataSink/history/";
5
6      public static final String PATH_NODE_BASEDIR = "dataNodes/";
7      public static final String PATH_NODE_UPDATED_MODEL = PATH_NODE_BASEDIR
       + "NewUpdatedModel.json";
8      public static final String PATH_NODE_NEW_MODEL = PATH_NODE_BASEDIR + "
       newModel";
9      public static final String PATH_NODE_COLLECTED_DATA = PATH_NODE_BASEDIR
        + "collectedData";
10     public static final String PATH_NODE_READY_DATA = PATH_NODE_BASEDIR + "
       readyData";
11
12     public static final Double PERCENTAGE_OLD_VALUES = 0.5;
13     public static final int SIZE_WINDOW = 200;
14
15     /*
16      * Generating parameters
17      */
18     public static final int MEAN = 30;
19     public static final int ST_DEV = 5;
20 }
```

Listing 3.18: Config

Capitolo 4

REST Server

Per integrare gli algoritmi di clustering realizzati in python con il core del progetto realizzato invece in Java è stato di scelto di far comunicare i due linguaggi mediante l'utilizzo di un serve REST in grado di fornire le funzioni realizzate in python tramite messaggi REST appositamente realizzati.

4.1 Implementazione

Dal punto di vista implementativo è stato scelto di realizzare il server REST mediante l'utilizzo della libreria Flask, in quanto offre un servizio completamente funzionante e modificabile seguendo le preferenze del programmatore.

4.1.1 Gestione delle richieste

Per poter chiamare i metodi messi a disposizione dal server REST vengono effettuate delle richieste REST nella quale si specifica, mediante un messaggio json, il metodo da richiamare ed gli argomenti necessari, rimanendo che il risultato venga processato e che un codice corrispondente allo stato d'esecuzione del metodo venga rispedito al mittente

```
1 from flask import Flask,request, jsonify
2 from flask_restful import Api,Resource,reqparse
3 from FCM import FCM
4 from Utils import removeOldFiles
5
6 class Server(Resource):
7     def post(self):
8         if(request.json['command'] == "Train"):
```

```
9               return FCM().train(request.json['ID'],request.json["Coeff"],
    request.json["Window"],request.json['values'])
10          elif (request.json['command'] == "Merge"):
11              return FCM().merge()
12          elif (request.json["command"] == "Update"):
13              return FCM().update(int(request.json["ID"]))
14          else:
15              return "Command not available",200
16
17  removeOldFiles()
18  app = Flask(__name__)
19  api = Api(app)
20  api.add_resource(Server, '/server')
21  app.run(debug=True)
```

Capitolo 5

Analisi dei Dati

I dati prodotti sul nodo vengono analizzati localmente in modo da non dover inviare le informazioni raccolte attraverso la rete, in modo da ridurre il carico di informazioni sul nodo centrale e di preservare la privacy dell'utente, in quanto l'unica informazione resa pubbblica sarà il modello generato, ma non i dati necessari a generare tale modello.

In particolare lo scopo delle nostre analisi è stato quello di effettuare il clustering di una serie di punti generati casualmente, e per ottenere questo risultato ci siamo affidati ad un Fuzzy C-Means, in modo non solo da individuare per ciascun dato a quale cluster appartenesse, ma anche il suo grado di appartenenza a tale cluster, al fine di effettuare future analisi ulteriormente più precise.

In aggiunta abbiamo inserito alcuni meccanismi atti a migliorare la creazione dei cluster, in particolare:

- E' stato realizzato un meccanismo di finestra scorrevole utile ad analizzare solo una porzione dei dati complessivi, in modo da dare un'importanza relativa (che può essere scelta dall'utente) ai valori storici rispetto a quelli appena ottenuti.
- E' stato anche inserito un controllo riguardo la posizione dei punti stessi rispetto ai precedenti centri dei cluster, in modo da accertarci prima di rigenerare un modello che i punti che si andranno ad analizzare siano validi e non degli outlier che sporcherebbero il modello finale.

5.1 Implementazione

Per realizzare queste analisi ci siamo avvalsi del linguaggio Python e di una libreria esterna che offriva un algoritmo di Fuzzy C-Means già completo e funzionante.

5.1.1 Training

Una volta generati i valori sul nodo verrà calcolato il modello come specificato nel seguente codice:

```
            with open(MERGED_MODEL_PATH,"w") as mergedModelFile:
                json.dump(mergedModel,mergedModelFile)
            return "Models merged",201
        else:
            return "not sufficient files",204

    def train(self, id,coeff,window,values):
        #Retriving the dataframe related to the generated file
        df = pd.read_csv(BASE_DATA_PATH+id+".txt",names=["X","Y"],header=
    None,dtype={"X":float,"Y":float})
        #Computing the effective dimension of the window
        dim = int(int(window)*(1+float(coeff)))
        START_WINDOW = dim * (-1);
        if df.shape[0] == int(values):
            result = True
        else:
            #Checking if the model must be computed
            NEW_VALUES = int(values) * -1
            newValues = df[NEW_VALUES:]
            #Deleting from the original dataframe the new values and the
    previous window
            df = df[:NEW_VALUES]
            [df,result] = self.isModelNeeded(id,df,newValues)
        if(result):
            #Selecting only the desired window
            if (START_WINDOW * (-1)) < df.shape[0]:
                df = df[START_WINDOW:]
            #Training the FCM with the array just obtained
            points = np.array(df)
            cntr,u_orig, _, _, _, _, _ = fuzz.cluster.cmeans(points.T,
    CLUSTERS,2,error=ERROR_THRESHOLD,maxiter = MAX_ITER)
            #Creating the JSON with the information of the created model
            model = {}
            model["centers"] = cntr.tolist()

            jsonToSave = {}
            jsonToSave["points"] = points.tolist()
            jsonToSave["centers"] = cntr.tolist()
            save(0,int(id),"trainResult"+str(id)+"_"+str(time.time()),
    jsonToSave)
```

```
37
38
39              #Saving the JSON in the file
40              with open(BASE_MODEL_PATH+id+".json","w") as newModelFile:
```

5.1.2 Validazione dei punti

Come spiegato precedentemente però non sempre un modello deve essere ricalcolato, in quanto ci possono essere dei punti appena generati che sono outlier e che quindi possono sporcare il modello (nelle analisi abbiamo considerato come limite di outlier accettabile un quarto dei nuovi valori inseriti). Questo processo ovviamente non si applica alla generazione del primo modello, in quanto non avendo un base di partenza, tutti i punti vengono considerati buoni.

```
1                   newModelFile.write(json.dumps(model))
2
3               #Returning the OK code
4               return "Model created",201
5           else:
6               return "",204
7       def isModelNeeded(self,id,df,df2):
8           if os.path.isfile(BASE_MODEL_PATH+id+".json"):
9               with open(BASE_MODEL_PATH+id+".json","r") as modelFile:
10                  #Load the centers from the model saved in the file
11                  centers = np.array(json.load(modelFile)["centers"])
12                  #Compute the distance between the new point and each center
     and find
13                  #the minimum distance for each new value
14                  minDistances = np.amin(cdist(np.array(df2.values),centers,
    metric='euclidean'),axis=1)
15                  #Finding the correct points and the outliers
16                  correct = (minDistances <= DISTANCE_THRESHOLD)
17                  outliers = np.invert(correct)
18                  #Creating a dataframe from that tuples
19                  df = pd.concat([df,df2.loc[correct]])
20                  #Writing on file the new dataframe
21                  df.to_csv(BASE_DATA_PATH+id+".txt",index = None,header =
    None)
22                  #checking if the number of outliers is above the threshold
23                  if df2.loc[outliers].shape[0] <= int(df2.shape[0] * 0.5):
24                      return df,True
```

5.1.3 Merging

Il sink invece ha il compito di unire tutti i modelli ricevuti in un unico generico, che sia in grado di migliorare (qualora fosse possibile) la precisione dei modelli generati singolarmenti dai nodi ai bordi della rete.

Per effettuare questa unione è stato deciso di riapplicare un Fuzzy C-Means avendo come dati in input i centri stessi ricevuti, al fine di ottenere dei nuovi centroidi che siano migliori, sfruttando le informazioni ricevute. In aggiunta per garantire un'accuretezza migliore nei confronti dei nodi, dopo aver generato questo nuovo modello, si andrà a calcolare quanto differisce da ciascun modello di partenza, in quanto ci possono essere dei modelli che sono totalmente differenti dalla maggior parte, in quanto i valori possono essere stati generati in diverse condizioni. Per garantire anche a questi nodi di aver il miglior modello possibile viene inviato, insieme al modello unito, anche un peso per ciascun nodo - che varia tra 0 e 1 - indicante la rilevanza del modello unito rispetto a quello di partenza.

```
1     def merge(self):
2         with open("../../dataSink/nodeList.txt","r") as nodeListFile:
3             nodeList = nodeListFile.readline().split(",")
4         if len(nodeList) > 1:
5             #Obtaining the centers
6             with open(BASE_MODEL_SINK_PATH+str(nodeList[0])+".json","r") as
      model:
7                 centers = np.array(json.load(model)["centers"],dtype=float)
8             for i in range(1,len(nodeList)):
9                 #Opening the file and concatenating the centers
10                with open(BASE_MODEL_SINK_PATH+str(nodeList[i])+".json","r"
    ) as model:
11                    nodeCntrs = np.array(json.load(model)["centers"],dtype=
    float)
12                    centers = np.vstack((centers,nodeCntrs))
13
14            cntr,u_orig, _, _, _, _, _ = fuzz.cluster.cmeans(centers.T,
    CLUSTERS,2,error=ERROR_THRESHOLD,maxiter = MAX_ITER)
15            mergedModel = {}
16            mergedModel["centers"] = cntr.tolist()
17
18            #Computing the mean Minumum distance for the new centers from
    the old centers
19            for i in range(0,len(nodeList)):
20                with open(BASE_MODEL_SINK_PATH+str(nodeList[i])+".json","r"
    ) as model:
```

```
21                      oldcntrs = np.array(json.load(model)["centers"])
22
23                  indexes = np.argmin(cdist(cntr,oldcntrs,metric='euclidean')
    ,axis=1)
24                  #Calculating the minimum value along the row
25                  minDistances = np.amin(cdist(cntr,oldcntrs,metric='
    euclidean'),axis=1)
26                  minDistancesIndex =  np.argmin(cdist(cntr,oldcntrs,metric='
    euclidean'),axis=1)
27                  #Check if there are at least a repetition
28                  if(np.unique(minDistancesIndex).shape[0] ==
    minDistancesIndex.shape[0]):
29                      #Compute the mean of the distances
30                      meanDistance = np.mean(minDistances)
31                      mergedModel[str(nodeList[i])] = self.associate(
    meanDistance)
32                  else:
33                      mergedModel[str(nodeList[i])] = 0
34
35              jsonToSave = {}
36              jsonToSave["newcenters"] = cntr.tolist()
37              jsonToSave["oldcenters"] = centers.tolist()
38              save(1,0,"MergedModel_"+str(time.time()),jsonToSave)
```

5.1.4 Updating

Infine, una volta che ciascun nodo ha ricevuto il modello unito dal sink, il nodo avrà il compito di aggiornare il proprio modello in base alle informazioni appena ricevute, in particolare la discriminante è il peso che il sink stesso ha associato al suo modello che indica:

- 0: Il nodo deve usare il proprio modello invece che quello unito, in quanto è troppo differente
- 0.5: Il nodo deve usare un modello che è la media aritmetica tra il proprio e quello ricevuto
- 0.75: Il nodo deve usare un media pesata tra i modelli - il suo modello ha peso 1 mentre quello ricevuto ha peso 3 - .
- 1 : Il nodo deve usare solo il modello ricevuto dal sink, e scartare il proprio.

```
                else:
                    return df,False
        else:
            return df,True
    def update(self,id):
        with open(BASE_MODEL_PATH+str(id)+".json","r") as oldModelFile:
            oldModel = np.array(json.load(oldModelFile)['centers'])
        with open(MERGED_MODE_NODE_PATH,"r") as mergedModelFile:
            dict = json.load(mergedModelFile)
            mergedModel = np.array(dict['centers'])
            weight = float(dict[str(id)])
        distances = cdist(mergedModel,oldModel,metric="euclidean")
        minDistancesIndex = np.argmin(distances,axis=1)
        updatedPoint = []
        for i in range (0,mergedModel.shape[0]):
            IncrementX = float(mergedModel[i,0])*weight + float(oldModel[
    minDistancesIndex[i],0])*(1-weight)
            IncrementY = float(mergedModel[i,1])*weight + float(oldModel[
    minDistancesIndex[i],1])*(1-weight)
            updatedPoint.append([IncrementX,IncrementY])

        jsonToSave = {}
        jsonToSave["oldModel"] = oldModel.tolist()
        jsonToSave["mergedModel"] = mergedModel.tolist()
        jsonToSave["updatedPoint"] = updatedPoint
        save(0,int(id),"updatedPoint"+str(id)+"_"+str(time.time()),
    jsonToSave)

        dict = {}
```

Capitolo 6

Testing

6.1 Scenario

Per testare il funzionamento dell'architettura sviluppata abbiamo simulato uno scenario descritto dal seguent elenco:

- 5 nodi ai bordi della rete dove ognuno dei quali:
  - Generava valori con 100 thread avendo come tempo medio attesa tra due valori circa 1 secondo - il tempo di interarrivo seguiva una distribuzione esponenziale con valor medio 0.9 -.
  - Ogni thread generava valori seguendo due distinte distribuzioni guassiane, una con valore medio 30 e varianza 5 e una con valor medio -30 e varianza 5.
  - Il numero di valori generati da ogni thread era pari a 1000, per un totale quindi di 1000000 valori per generati da ciascun nodo.
  - Generavano un modello avente due cluster, successivamente all'arrivo di un numero di nuovi valori pari alla metà del numero di vecchi valori.
  - Sempre riguardo la generazione del modello è stato scelto come errore massimo del Fuzzy C-Means 0.005 e come massimo numero di iterazioni 1000.
  - Per determinare se un punto è o meno un outlier invece è stata considerata una distanza massima dal centroide precedente - qualora sia presente - pari a 30.
- Per quanto riguarda il nodo Sink sono scelti i seguenti parametri:
  - Per la generazione del modello unificato sono stati usati gli stessi parametri usati anche dai nodi.

- Per far si che tutti i nodi abbiano il miglior modello possibile sono state scelte i seguenti pesi in funzione delle distanze:
  - Se la distanza media tra i centri del nodo e quella del modello unito è minore o uguale a 3 viene dato peso 1
  - Se la distanza media tra i centri del nodo e quella del modello unito è tra 3 e 5 viene dato peso 0.75
  - Se la distanza media tra i centri del nodo e quella del modello unito è tra 5 e 7 viene dato peso 0.5
  - Se la distanza media tra i centri del nodo e quella del modello unito è maggiore di 7 viene dato peso 0
- Generavano un modello avente due cluster, successivamente all'arrivo di un numero di nuovi valori pari alla metà del numero di vecchi valori

6.2 Risultati ottenuti

6.2.1 Generazione dei punti

Nello scenario precedentemente descritto i punti vengono generati dai nodi come mostrato dalle seguenti figure **??**,**??**,**??**:

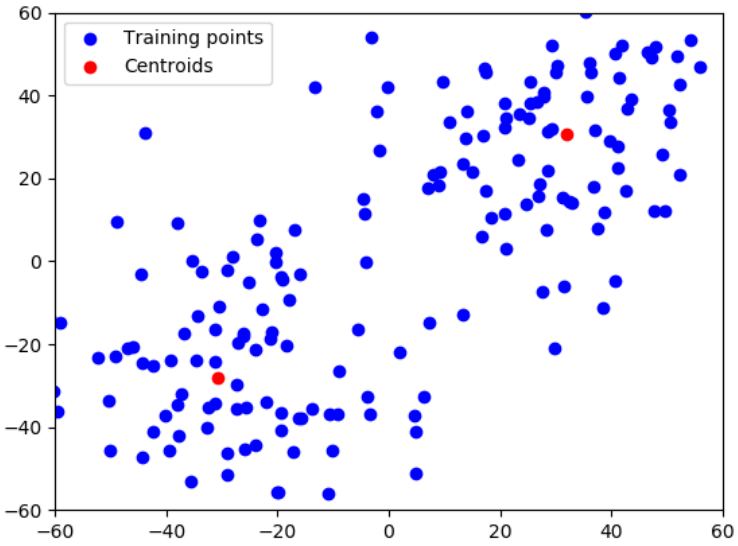

Figura 6.1: Dati generati alla prima generazione del modello

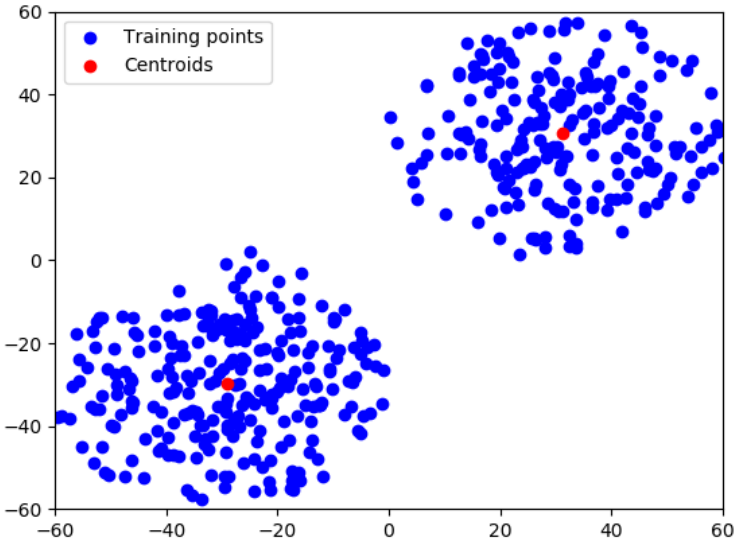

Figura 6.2: Dati generati alla quinta generazione del modello

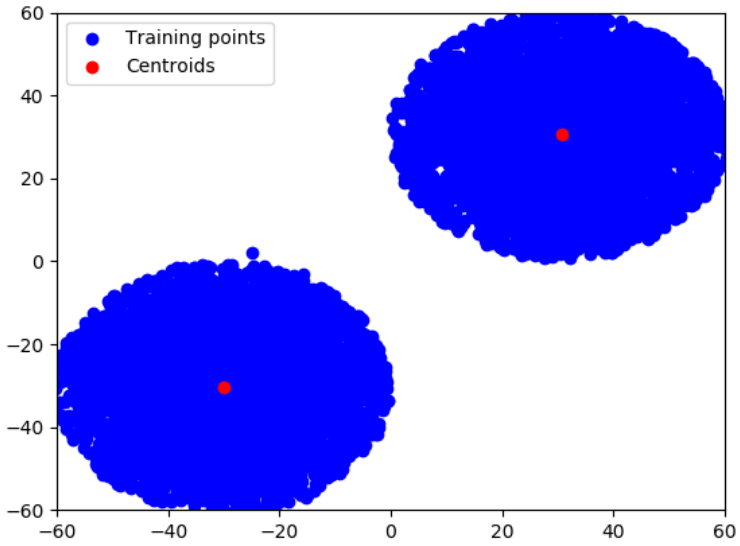

Figura 6.3: Dati generati all'undicesima - e ultima - generazione del modello

6.2.2 Unione dei modelli

Per quanto riguarda l'unione dei modelli il nostro algoritmo si è comportato in accordo alle figure ??,??,??:

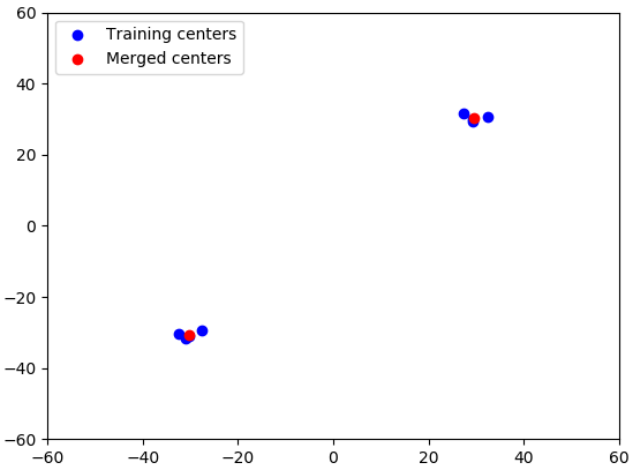

Figura 6.4: Modelli uniti in seguito alla prima generazione

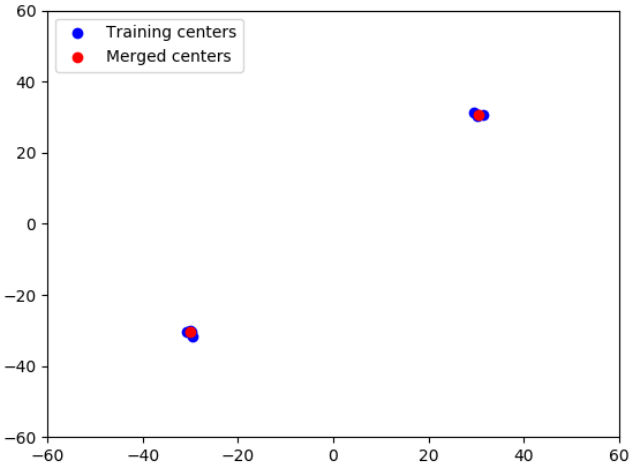

Figura 6.5: Modelli uniti in seguito alla quinta generazione

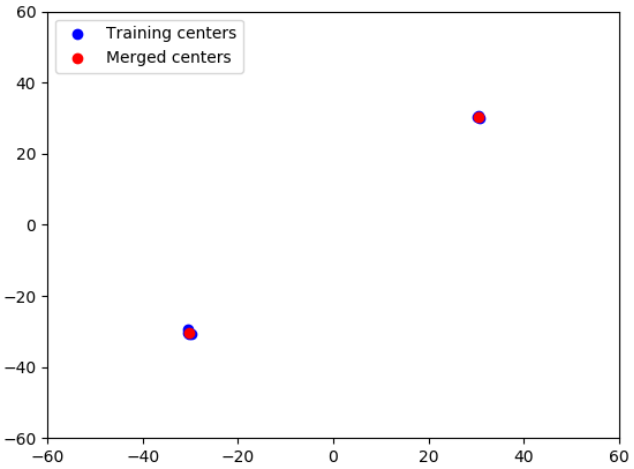

Figura 6.6: Modelli uniti in seguito alla nona - e ultima - generazione

Come si può notare dalle immagini all'aumentare del numero di iterazioni i centri tendono a convergere verso un unico punto, portando quindi ad un modello comune considerabile preciso - in quanto calcolato singolarmente - per tutti i nodi.

6.2.3 Aggiornamento dei modelli

Per quanto riguarda l'unione dei modelli il nostro algoritmo si è comportato in accordo alle figure **??**,**??**,**??**:

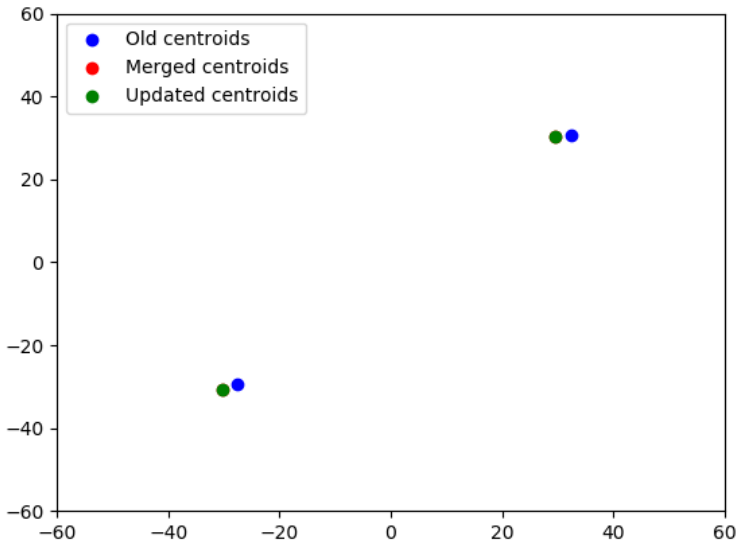

Figura 6.7: Modelli uniti in seguito alla prima generazione

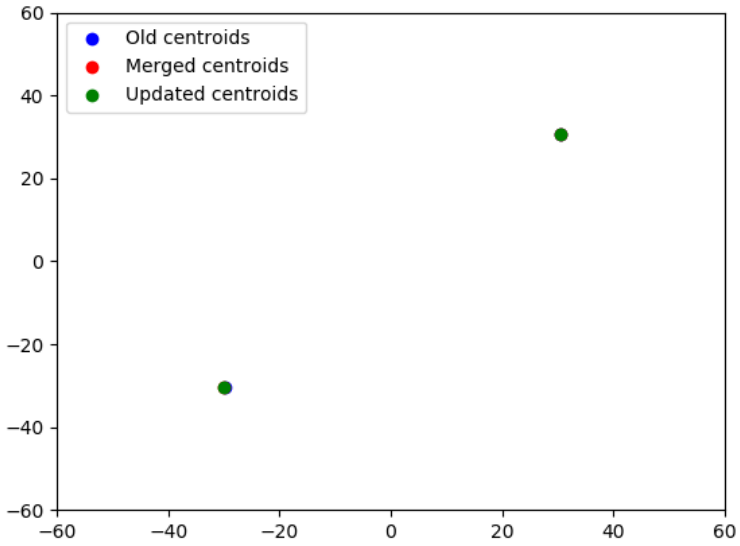

Figura 6.8: Modelli uniti in seguito alla quinta generazione

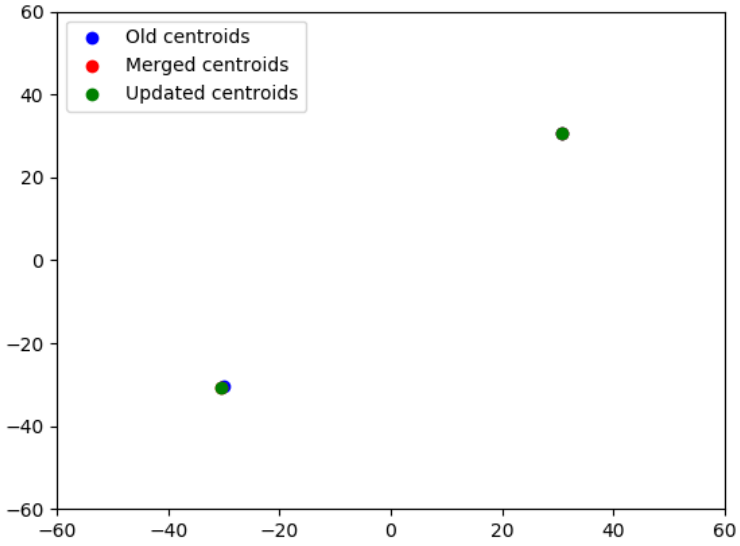

Figura 6.9: Modelli uniti in seguito alla nona - e ultima - generazione

Come si può notare invece da queste figure i punti del modello generati dal nodo erano sempre molto vicini ai punti del modello unificato, ed infatti i punti aggiornati

- quelli in verde - si sovrapponevano sempre ai punti del modello unificato - in rosso -. Si può notare inoltre che lo stesso risultato presentato nell'unione dei modelli si ottiene anche qui, ovvero che i centri del modello generato dal nodo all'aumentare delle iterazioni convergono verso i centri del modello unificato.