Project

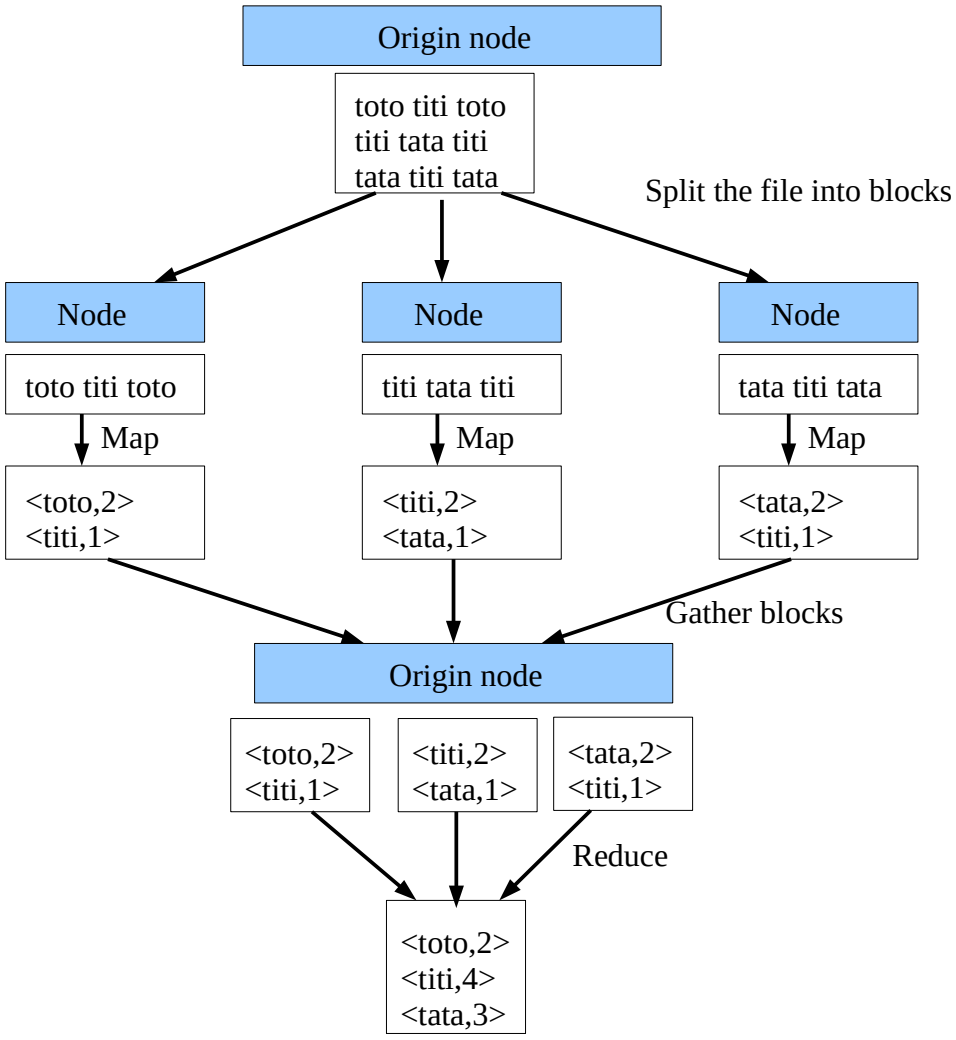
Daniel Hagimont - Daniel.Hagimont@enseeiht.fr

USTH – Teaching Unit MI 2.01

March 2022

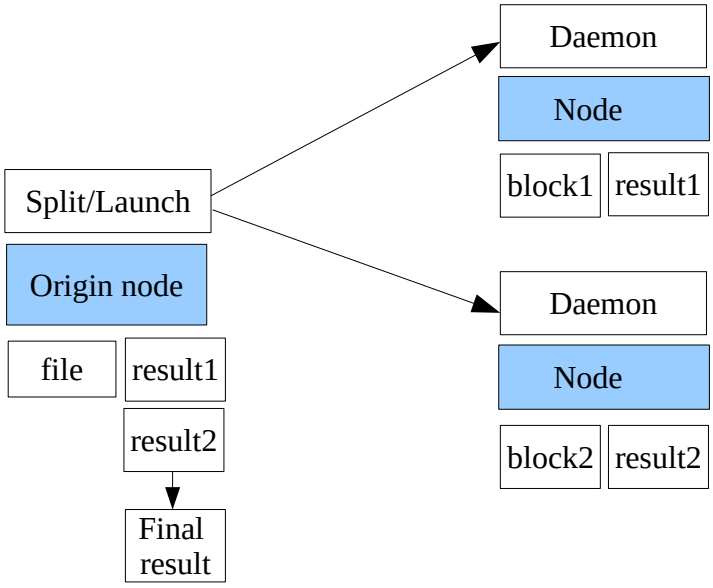
Objectives

The objective of this project is to implement a tools which allows the execution of applications following the map-reduce principle. The map-reduce principle is used in big data processing for speeding up the treatment of very large data sets (large files).
This principle is to split a large file into blocks (files) which are distributed on several nodes. Then an application (we call it Map) can be executed on each of these blocks in parallel. All the results (files) from these computations are then gathered on the original node where another application (we call it Reduce) is executed, in order to compute the final result (one file).
The typical application we will use is the wordcount application. This application counts the occurrences of each word in a big file. Each Map counts the occurrence of words in each block, producing a table <word, count>. Then the Reduce aggregates the produced tables in order to obtain the final result (a table <word, count>).
The following figure illustrates the execution of wordcount following the map-reduce principle.



Method

In order to implement this scenario, you must implement the following architecture.



We describe each component and the interfaces are given below.

Daemon

This class is executed on every Node (not the Origin node). It can be called by 2 programs, Split and Launch.

- Split can connect (with TCP) to Daemon in order to upload a block of data to the node. Then the block is stored locally in a file.
- Launch can invoke (with RMI) a method call() of Daemon in order to execute a Map on the node. This call() method (see interfaces below) receives as parameter an instance (m) of the class which includes the method (executeMap()) which has to do the Map processing, the filename (blockin) of the block to process locally and the filename (blockout) of the local file where results must be stored.
- Launch can connect (with TCP) to Daemon in order to download a block of data (results) from the node.

An important remark is that the call() method of the Daemon must create a thread for the execution of the Map. If you don't do that, the calls on the Daemons would be executed sequentially (they are called sequentially by Launch). By creating a thread, they are executed in parallel. You then need a way to notify Launch the a Map computation is completed. The Callback parameter of call() provides a means to notify Launch that a Map is completed (Callback is remote). Therefore Launch must wait for the completion of all the Maps before starting Reduce.

Split

This class is executed on the origin node (before we launch the execution) to split the big file into blocks and send them to the nodes. It takes as parameter (args[0]) a file name (the big file to process in parallel). It splits the file into blocks, opens connections with the different Daemons

and sends the blocks to the Daemons. We assume that if there are 3 Daemons, the file is split into 3 blocks. Be careful not this cut the file in the middle of a line.

Launch

This class is executed on the origin node to launch the execution. It invokes with RMI all the Daemons to start the execution of Maps with the call() method. Then it connects (with TCP) to the Daemons to download the result blocks. Finally, it executes the Reduce code to process the final result.

WordCount

This class implements the wordcount application. It implements the MapReduce interface. The executeMap() method is executed on Nodes, and it reads lines from blockin, computes the count table (a table <word, count>), and stores it in blockout. The executeReduce() method is executed on the origin node, and it reads the count tables from blocks and computes the final count table which is stored in finalresults.

Interfaces

```
public  interface Daemon extends Remote {
        public void call(Map m, String blockin, String blockout, CallBack cb)
                        throws RemoteException;
}
public interface MapReduce extends Serializable {
        public void executeMap(String blockin, String blockout);
        public void executeReduce(Collection<String> blocks, String finalresults);
}

public interface CallBack extends Remote {
        void completed() throws RemoteException;
}
```

Instructions

You must implement these programs in Java with sockets and RMI.
You must demonstrate it with a wordcount application.

You must sent to me an email describing your achievement (1 page) with the source code of your contributions and the instructions (very short) to run it on my local machine (with different ports on localhost). **The dead-line is April 2022, the 5th (23:59 Hanoi time).**

Classes to help you

```
public class CallBackImpl extends UnicastRemoteObject implements CallBack {
  int nbnode;
  public CallBackImpl(int n) throws RemoteException { nbnode = n; }
  public synchronized void completed() throws RemoteException {
    notify();
  }
  public synchronized void waitforall() {
    for (i=0;i<nbnode;i++) wait();
  }
}


public class WordCount implements MapReduce {
    public static final String SEPARATOR = " - ";
    public void executeMap(String blockin, String blockout) {
        // read from blockin, compute count table, write to blockout
        HashMap<String,Integer> hm = new HashMap<String,Integer>();
        try {
            BufferedReader br= new BufferedReader(
                            new InputStreamReader(new FileInputStream(blockin)));
            BufferedWriter bw = new BufferedWriter(
                            new OutputStreamWriter(new FileOutputStream(blockout)));
            String line;
            while ((line=br.readLine()) != null) {
                StringTokenizer st = new StringTokenizer(line);
                while (st.hasMoreTokens()) {
                    String tok = st.nextToken();
                    if (hm.containsKey(tok)) hm.put(tok, hm.get(tok)+1);
                    else hm.put(tok, 1);
                }
            }
            for (String k : hm.keySet()) {
                bw.write(k+SEPARATOR+hm.get(k).toString());
                bw.newLine();
            }
            br.close();
            bw.close();
        } catch (Exception e) {
            e.printStackTrace();
        }
    }
    public void executeReduce(Collection<String> blocks, String finalresults) {
        // read all files in blocks, merge and write to finalresults
        HashMap<String,Integer> hm = new HashMap<String,Integer>();
        try {
            for (String block : blocks) {
                BufferedReader br= new BufferedReader(
                                        new InputStreamReader(new FileInputStream(block)));
                String line;
                while ((line=br.readLine()) != null) {
                    String kv[] = line.split(SEPARATOR);
```

```
                String k = kv[0];
                int v = Integer.parseInt(kv[1]);
                if (hm.containsKey(k)) hm.put(k, hm.get(k)+v);
                else hm.put(k, v);
            }
            br.close();
        }
        BufferedWriter bw = new BufferedWriter(
                                    new OutputStreamWriter(new FileOutputStream(finalresults)));
        for (String k : hm.keySet()) {
            bw.write(k+SEPARATOR+hm.get(k).toString());
            bw.newLine();
        }
        bw.close();
    } catch (Exception e) {
        e.printStackTrace();
    }
}
public static void main(String[] args) {
    WordCount wc = new WordCount();
    wc.executeMap("data.txt", "result.txt");
    Collection<String> blocks = new ArrayList<String>();
    blocks.add("result1.txt");
    blocks.add("result2.txt");
    wc.executeReduce(blocks, "finalresult.txt");
}
}
```

CallBack scheme