CESC 327
Assignment 4: Map-Reduce

Professor Oscar Morales

Due date May 11th

1 Objective

The objectives of this project are:
1) Identify the fundamental concepts of Map Reduce.
2) Understand how distributed sorting reduces the sorting time

The task is to develop a map-reduce framework over the distributed file system of the previous lab as we saw in class. It accepts a distributed file where each row has a long and a string separated by comas (Long,String). To store and sort the values in the map and reduce phase use the data structure $TreeMap$. Let $BMap = TreeMap < Long, List < String >>$ and $BReduceTreeMap < Long, String >$.

The interface Context will be the main responsible for checking when the phases have been completed.

```
public interface ContextInterface  extends Remote {
  public void setWorkingPeer(Long page);
  public void completePeer(Long page, Long n) throws RemoteException;
   public Boolean isPhaseCompleted();

  public void reduceContext(Long source, ReduceInterface reducer,
         Context context) throws RemoteException;
  public void mapContext(Long page, MapReduceInterface mapper,
        Context context) throws RemoteException;
}
```

```
public interface MapReduceInterface  {
  public void map(Long key, String value) throws IOException;
  public void reduce(Long key, String value[]) throws IOException;
}
```

2 Remote Procedures

- emitMap(Long key, String value): During the map phase, the implementation of map will call emitMap in the local peer.

  Store (key,value) in the peer the responsible with $locateSuccessor(key).emitMap(key, value)$. $emitMap$ stores (key, value) locally in $BMap$. The implementation of BTree does not directly support duplicate keys, but it is possible to handle duplicates by inlining or referencing an object collection as a value.

- emitReduce(Long key, String value): During the map phase, the implementation of map will call emitReduce in the local peer. No key repetition.

  Similar as before, store (key,value) in the peer the responsible with $locateSuccessor(key).emitReduce(key, value)$. $emitReduce$ stores (key, value) locally in $BReduce$.

- mapContext(Long page, MapReduceInterface mapper, Context context): Opens the page (page), read line-by-line and execute $mapper.map(key, value, context)$. When it has read the complete file, it calls $context.completePeer(page, n)$ where n is the number of rows. You have to create a new thread to avoid blocking. Observe that context is the instance of the coordinator or initiator.

- reduceContext(Long source, ReduceInterface reducer, Context context): If $source \neq guid$, call $context.add(guid)$ and then call $successor.reduceContext(source, reducer, context)$. Then create a new thread to avoid blocking in which you have to read in order $BReduce$, and execute $reducer.reduce(key, value[], context)$. Make sure that the tree is stored in persistent memory. When it completes reading $BReduce$, it calls $context.complete(guid, n)$ where n is the number or rows and $guid$ is the guid of the peer.

Note: It must exist a metafile called "fileName_reduce" where fileName is the original logical file that you are sorting with n pages. Each peer create a page (id) with the data in $BReduce$ and insert into "fileName_reduce".

The implementation of context should look like:

```
public class  Context extends ContextInterface {
  Long n = 0;
  Set<Long> set;
  public void setWorkingPeer(Long page)
  {
      set.add(page);
  }
  public void completePeer(Long page, Long n) throws RemoteException;
  {
      this.n += n;
      set.remove(page);
  }
  public Boolean isPhaseCompleted()
  {
    if (set.isEmpty())
      return true;
    return false;
  }
   public void reduceContext(Long source, ReduceInterface reducer,
         Context context) throws RemoteException
         {
            // TODO
         }
  public void mapContext(Long page, MapReduceInterface mapper,
        Context context) throws RemoteException
         {
            // TODO
         }
 }
```

Any peer can initialize the service by calling the method $runMapReduce(file)$ in the distributed file system. The algorithm is as follows:

```
context= new Context()
mapreduce = new MapReduceInterface();
// map Phases
for each page in metafile.file
   context.add(page);
   let peer be the process responsible for storing page
   peer.mapContext(page, mapreduce, context)
 wait until context.hasCompleted() = true
 // reduce phase
 reduceContext(guid, mapreduce, context);
 wait until context.hasCompleted() = true;
```

The following algorithm implement a MapReduce to count the number of words:

```
public class Mapper extends MapReduceInterface  {
  public void map(Long key, String value) throws IOException
  {
   For each word in value
   emit(md5(word), word +":"+1);
  }

  public void reduce(Long key, String values[]) throws IOException
  {
    word = values[0].split(":")[0]
    emit(key, word +":"+ len(values));
  }

}
```

3 Grading

Criteria	Weight
Documentation of your program	15%
Source code (good modularization, coding style, comments)	15%
Execution	70%

The documentation must be generated using Doxygen or Javadocs.

4 Deliverables

Compress the source and documentation in a zip file and upload to beachboard. The zipfile must contain two folders:
1.- Src: All the source code to compile it
2.- Docs: HTML with the documentation. It has to contain the definitions of the methods with a short description, parameters and output of the method.