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 $BReduce = TreeMap < Long, String >$.

We extend the ChordMessageInterface to execute the main functionality of the Map Reduce.

```
public interface ChordMessageInterface extends Remote
{
   ...

    public void setWorkingPeer(Long page) throws IOException;
    public void completePeer(Long page, Long n) throws RemoteException;
    public Boolean isPhaseCompleted() throws IOException;
    public void reduceContext(Long source, MapReduceInterface reducer,
              ChordMessageInterface context) throws RemoteException;
    public void mapContext(Long page, MapReduceInterface mapper,
```

```
            ChordMessageInterface context) throws RemoteException;


    public void emitMap(Long key, String value) throws RemoteException;
    public void emitReduce(Long page, List<String> value) throws RemoteException;

}
```

2 Remote Procedures

- $emitMap(Longkey, Stringvalue)$: During the map phase, the implementation of map will call $context.emitMap$.

  if $(isKeyInOpenInterval(key, predecessor.getId(), successor.id()))$ then store (key,value) locally in $BMap$. Otherwise call $locateSuccessor(key).emitMap(key, value)$. 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.

- $emitReduce(Longpage, List < String > value)$: During the reduce phase, the implementation of reduce will call $context.emitReduce$. if $(isKeyInOpenInterval(key, predecessor.getId(), successor.id()))$ then store (key,value) locally in $BReduce$. Otherwise call $locateSuccessor(key).emitReduce(key, value)$. Make sure that the tree is stored in persistent memory.

- mapContext(Long page, MapReduceInterface mapper, ChordMessageInterface context): call setWorkingPeer(page) then opens the page (guid), read line-by-line, parse 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, MapReduceInterface reducer, ChordMessageInterface context): If $source \neq guid$, call $successor.reduceContext(source, reducer, context)$. Then, create a new

thread to avoid blocking in which you have to read in order $BMap$, and execute $reducer.reduce(key, value, context)$.

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 creates a page (guid) with the data in $BReduce$ and insert into "fileName_reduce".

The implementation of Chord.java should look like:

```
public class Chord extends java.rmi.server.UnicastRemoteObject
implements ChordMessageInterface
     ...

    private Long numberOfRecords
    private Set<Long> set;
    private Map< Long, List< String >> BMap;
    private Map< Long, String > BReduce;


    public void setWorkingPeer(Long page) throws IOException
    {
        set.add(page);
    }
    public void completePeer(Long page, Long n) throws RemoteException{
        this.numberOfRecords += n;
        set.remove(page);

    }

    public Boolean isPhaseCompleted() throws IOException
    {
        return set.isEmpty();
    }

    public void reduceContext(Long source, MapReduceInterface reducer,
                ChordMessageInterface context) throws RemoteException
    {
      //TOD
```

```
}

public void mapContext(Long page, MapReduceInterface mapper,
          ChordMessageInterface context) throws RemoteException
{
   //TODO

}

public void emitMap(Long key, String value) throws RemoteException
{
      if (isKeyInOpenInterval(key, predecessor.getId(), successor.getId()))
      {
          // insert in the BMap. Allows repetition
          if (!BMap.containsKey(key))
          {
                List< String > list = new  List< String >();
                BMap.put(key,list);
          }
          BMap.get(key).add(value);

      }
      else
      {
          ChordMessageInterface peer = this.locateSuccessor(key);
          peer.emitMap(key, value);
      }
}

public void emitReduce(Long key, String value) throws RemoteException
{
      if (isKeyInOpenInterval(key, predecessor.getId(), successor.getId()))
      {
          // insert in the BReduce
          BReduce(key, value);
      }
      else
      {
```

```
            ChordMessageInterface peer = this.locateSuccessor(key);
            peer.emitReduce(key, value);
        }

    }

public interface ChordMessageInterface
{

public void map(Long key, String value,
    ChordMessageInterface context) throws IOException;
public void reduce(Long key, List< String > value,
    ChordMessageInterface context) throws IOException
};
```

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

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

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

```
public class Mapper extends MapReduceInterface  {
  public void map(Long key, String value,
      ChordMessageInterface context) throws IOException
  {
   context.emitMap(key, value);
  }

  public void reduce(Long key, List< String > value,
```

```
   ChordMessageInterface context) throws IOException
  {
     context.emitReduce(key, word +":"+ value.length());
  }

}
```

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.