CESC 327
Assignment 3: Peer-to-peer

Professor Oscar Morales

Due date April 5th

1 Objective

The objectives of this project are:
1) Identify the advantages RPC for programming distributed applications.
2) Understand the use of middlewares (in particular Java RMI) to implement distributed operative systems
3) Identify the advantages of peer-to-peer for sharing resources
4) Identify fault tolerance measures for handling failures

The task is to develop a peer-to-peer system for storing files. It is based on Chord. This assignments can be seen as an extension of the distributed chat implemented in the previous assignment. The processes and the files are mapped to an unique identifier using a hash function that distributes them uniformly. Thus, when a node joins or leaves the systems, only $O(\log n)$ keys are needed to be move where n is the number of processes. Each process stores the IP and port of the peers that have joined.

In Chord, process and files are hashed to an m-bit unique identifier where m is sufficiently large so that the probability of collisions during the hash is negligible, for example using an MD5 digest. The Chord overlay uses an ordered logical ring of size 2^m. A key k gets assigned to the first node such that its node identifier equals or follows the key identifier of k in the common identifier space. Each process i maintains a routing table, called the finger table, with at most $O(\log n)$ distinct entries, such that the x-th entry $(1 \leq x \leq m)$ is the process of the node $succ(i + 2^{x-1})$. The following code uses an RPC notation.

```
(variables) integer: $successor \leftarrow initial\_value$;
integer: $predecessor \leftarrow initial\_value$;
integer $finger[1...m]$;
integer: $next\_finger \leftarrow 1$;
(1) $i.Locate\_Successor(key)$; where $key \neq i$:
(1a) if $key \in (i, successor]$ then
(1b)   return($successor$)
(1c) else
(1d)   $j \leftarrow Closest\_Preceding\_Node(key)$;
(1e) return ($j.Locate\_Successor(key)$.
(2) $i.Closest\_Preceding\_Node(key)$; where $key \neq i$:
(2a) for $count = m$ down to 1 do
(2b) if $finger[count] \in (i, key]$ then
(2c)    $break()$;
(2d) return($finger[count]$).
(3) $i.Create\_New\_Ring$:
(3a) $predecessor \leftarrow \perp$;
(3b) $successor \leftarrow i$.
(4) $i.Join\_Ring(j)$, where $j$ is any node on the ring to be joined:
(4a) $predecessor \leftarrow \perp$;
(4b) $successor \leftarrow j.Locate\_Successor(i)$.
(5) $i.Stabilize$:   executed periodically to verify and inform successor
(5a) $x \leftarrow successor.predecessor$;
(5b) if $x \in (i, successor)$ then
(5c)   $successor \leftarrow x$;
(5d) $successor.Notify(i)$.
(6) $i.Notify(j)$:   $j$ believes it is predecessor of $i$
(6a) if $predecessor = \perp$ or $j \in (predecessor, i)$ then
(6b)   transfer keys in the range $[j, i)$ to $j$;
(6c)   $predecessor \leftarrow j$.
(7) $i.Fix\_Fingers$:   executed periodically to update the finger table
(7a) $next\_finger \leftarrow next\_finger + 1$;
(7b) if $next\_finger > m$ then
(7c)   $next\_finger \leftarrow 1$;
(7d) $finger[next\_finger] \leftarrow Locate\_Successor(i + 2^{next\_finger-1})$.
(8) $i.Check\_Predecessor$:   executed periodically to verify whether predecessor still exists
(8a) if $predecessor$ has failed then
(8b)   $predecessor \leftarrow \perp$.
```

2 Basic Programming Interface

- $Join(ip, port)$: Join the system. First, peers obtain the GUID base on the IP and Port.
- $put(GUID, data)$: Stores data in the processor responsible for storing the object.
- $value = get(GUID)$: Retrieves the data associated with GUID from one of the nodes responsible for it.
- $remove(GUID)$: Deletes all references to GUID and the associated data.
- $print$: Print the state of the systems.

The GUID is the file name is the MD5 of the file module 2^{31}, i.e., $GUID_{fileName} = MD5(fileName) \bmod 2^{31}$

3 User interface

Processes are identified using the guid = md5(port). Each process contains a local filesystem (guid) and a private directory, called repository (guid/repository), that is used only by the chord system. We identify the local filesystem of each process with the guid.

- *join*. Joins the Peer-to-peer.

  The skeleton implements it partially. You have to implement a fault tolerant measure to guarantee that the active peers have access to all the files. Essentially, you have to transfer the objects that the new member will be responsible. See line (6b) of the algorithm.

- *write*. Upload a file from the local filesystem to the chord.

  The user provides the filename of the file that wants to share. To identify the processor, first you obtain the GUID k of the object using the MD5. The file will be stored in the first processor such that its process identifier equals or follows k:
  $InputStream inputStream = new InputStream(filename)$
  $int guid \leftarrow md5(filename)$

$ChordInterface\ c = Locate\_Successor(guid)$
$c.put(guid, inputStream);$

- *read*. Download a file from the chord to the local filesystem.

  Everyone in the ring is able to download a previous uploaded file in the ring. Giving a filename, we can obtain the guid, i.e., $guid \leftarrow md5(filename)$. The guid is used to locate the peer responsible of storing the file
  $ChordInterface\ c = Locate\_Successor(guid)$. To download the file you use $OuputStream\ out = c.get(guid)$.

- *delete*. Delete a file from the chord.

  Similarly we can remove a file from the chord giving the filename.

- *leave*. Leave the chord.

  All the objects that the process contains are sent to the predecessor. You have to implement a fault tolerant measure to deal with the case when a user closes the process without executing *leave*. You have to catch the signal that is sent when the process is going to be deleted and execute the *leave*. This will guarantee that the objects are accessible for all the active participants of the chord.

4 Grading

Criteria	Weight
Documentation of your program	15%
Source code (good modularization, coding style, comments)	15%
Execution	
Writing, reading and deleting of remote files	40%
Update objects at joining	10%
Update objects at leaving	10%
Leaving clean (even when ctrl+c is used)	10%

The documentation must be generated using Doxygen or Javadocs.

5 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.