CESC 327
Assignment 3: Distributed File System

Professor Oscar Morales

Due date April 2d

1 Distributed File System

The task is to develop a distributed file system based on Chord. A data structure, called metadata, will store all the description of the files. The attributes are: file name, size, read and write timestamp, a reference count (number of processes that have the file open), number of pages (or chunks), size of the pages and the description of each page. Observe that files are split into several pages. For example:

```
 {
"file" :
{"name":"MusicJson","size":"2291", "writeTS":"1256953732",
"readTS":"1256953732", "writeTS":"1256953732",
"referenceCount":"1",  "numberOfPages":"3",
"maxPageSize":"1024",, "pages":
[{"guid":"46312", "size": "1024", "writeTS":"1256933732",
   "readTS":"1256953732", "writeTS":"1256953732", "referenceCount":"0"},
{"guid":"22412",  "size": "1024","writeTS":"1256933732",
   "readTS":"1256953732", "writeTS":"1256953732", "referenceCount":"0"},
{"guid":"93719",  "size": "243", , "writeTS":"1256933732",
   "readTS":"1256953732", "writeTS":"1256953732", "referenceCount":"0"}]}
}
```

All files are stored using a Global Unique Identifier (GUID). GUID is the key obtained by applying MD5 of a unique feature, i.e.,

$$md5(uniqueFeature)$$

Consider as unique feature, the file name concatenated with a timestamp for example.

The metadata will be stored in the peer-to-peer using the static name "Matadata", i.e., md5("Metadata") We refer to the files in the distributed file system as a distributed file meanwhile the file in your hard-disk as physical files. We also refer to the "file" key value in Metadata as JSONFile.

In all the operations, first, you need to read the Metadata from Chord, parse and apply the changes if needed and write back the modified Metadata accordingly.

We say that the metadata is a local file when you read the metadata and global if it is in the Chord. The distributed file system must support:

- Creation(filename). When a new distributed file is created, a new JSONFile is created in Metadata with the values initialized accordingly including the timestamps, the size of the file and the number of pages must be 0 and store globally.
  $metadata.addFile(filename)$

- Delete(filename). Delete the distributed file (including pages and data) and delete JSONFile with key filename from Metadata. That is, for each page k of filename
  $p = peer.locateSuccessor(k)$
  $p.delete(k)$
  Then you can delete the entry in metadata:
  $metadata.delete(filename)$

- Read(filename, page). Return the content of the page of the distributed file filename. Page is the index of the page. The read timestamp is set accordingly. Let k be the guid of page *page* of filename. Then you can read the content as follows:
  $p = peer.locateSuccessor(k)$
  $p.get(k)$

- Update(filename, page, newContent). Let k be the guid of page *page* of filename. This method updates the content of k by $newContent$. You can rewrite the content of k as follows:
  $p = peer.locateSuccessor(k)$
  $p.put(k, newContent)$
  The write timestamp is set accordingly.

- Append(filename, content). Append a new page to filename at the end of the JSONFile and uploads content. Recall that you obtain the guid of the page by applying MD5 to a unique feature, such as file "filename + timestamp". The write timestamp is set accordingly.
  $k = md5(filename + timestamp)$
  $metadata.appendPage(filename, k)$
  $p = peer.locateSuccessor(k)$
  $p.put(k, content)$

- Rename(fileName, newfilename). Rename the logical file fileName to newfilename in JSONFile. The write timestamp is set accordingly.
  $metadata.rename(filename, newfilename)$

There must be a command line interface that runs at the server side that allows to execute the previous commands. The command line iterface is used to add files and setting up the Distributed File system (Chord).

The music streaming server must read the file from the DFS to serve the requests. To simplify the task, each node of the Chord can implement the server. You need to randomly split the music.json into several pieces, say 5, to create the distribute file.

To set up the sytem, you need to run at least three process using different port (each process runs an instance of the Chord and your server). Suppose you program is called DFS that accepts one paramters. The first is the port that is listen. Then you run the program using command line:
DFS 2000
DFS 2001
DFS 2002
In your command line interface of peer with port 2001 you join peer 2000.
In your command line interface of peer with port 2002 you join peer 2000 or 2001.
This process creates a ring fo size three. Suppose you split music.json in the files (localmusic1.json, localmusic2.json, ... localmusic5.json). In your command line of any peer you create music.json as follows:
Creation music.json
append music.json localmusic1.json
append music.json localmusic2.json
...

append music.json localmusic5.json

Appendices

A Chord

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 MD5 digest. Chord overlay uses an ordered logical ring of size 2^m. A key k is assigned to the closest successor peer. 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 $succ(i + 2^{x-1})$.

B Basic P2P 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 (successor, predecessor, fingers).

The following code of CHORD 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 not 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) $predecessor \leftarrow \perp$.