COP5615
Operating System Principles
Fall 2007

Project 5

Date Assigned: November 08, 2007
Due Dates: November 26, 2007 (11:59 pm) for LOCAL Students (Sections 1093, 9333)
November 29, 2007 (11:59 pm) for EDGE Students (all other sections)

Introduction:

In this project you will implement the Basic Gossip Protocol for Data and File Replication. If updates are less frequent than reads and the order of the updates can be relaxed, updates can be lazily propagated among the replicas. In the gossip architecture, both read and update operations are directed by a file service agent (FSA) to any replica manager (RM). The FSA shields the replication details from the end-users / clients. Although there are two approaches to Gossip Update Propagation namely "Basic Gossip Protocol" and "Causal Order Gossip Algorithm" for the purposes of this project you will be limited to "Basic Gossip Algorithm". For detailed discussion on Gossip Update Propagation model, please refer to page 223 of your primary textbook.

Basic Gossip Protocol:

Each RM i is associated with a timestamp TS_i which represents the last update of the data object. Each FSA also maintains a timestamp TS_f which indicates the timestamp of the last successful access operation. There are 3 operations: read, update, and gossip.

- Read: TS_f of the FSA is compared with TS_i of the RM contacted. If $TS_f <= TS_i$, the RM has more recent data, this value is returned to the FSA and TS_f is set to TS_i. Otherwise FSA contacts some other RM.
- Update: FSA increments TS_f. It sends the update to one of the RM. If $TS_f > TS_i$, the update is executed at that RM. TS_i is set to TS_f to reflect the successful update. If $TS_f <= TS_i$, then RM replies back with the more recent value of the object, the FSA follows the rules specified for Read operation above, updates its timestamp, then increments its timestamp by 1 and attempts the update operation once again.
- Gossip: A gossip message carrying a data value from replica manager j to replica manager i is accepted if $TS_j > TS_i$. Otherwise it is ignored.

The Gossip Protocol specified above is specific to this project, and may vary very slightly (if any) from the one given in the book.

Detailed Specification:

You would design your project with 3 replica managers and 3 remote clients. We would not distinguish between FSA and end clients and in fact for the purposes of this project you would consider end clients / FSA to be one and the same. So in effect each end client / FSA would know about the server details of every RM. Your implementation would not contain manifest constants of any kind (*presence of manifest constants in your code, if detected, would result in significant penalty from your total grade*). So how would FSA know about the RM socket details if no manifest constants are allowed? Each RM when started on the remote CISE machine would dump the socket connection details like host IP address and server's port number in the local directory which would be later read by the FSA running on some other remote machine to build up its internal table of replica managers in the network.

Make use of any 6 of the sun114-xx machines for this project where xx could be replaced with 11-16, 31-36, and 41-46. Do not use any other machines. Your choice of 6 sun114-xx machines could be hardcoded in your start.java file. (start.java is with lowercase s and not capital S).

You will use UDP as communication protocol to communicate between FSA_x and RM_y and between RM_i and RM_j. The protocol message format is not specified in this project specification and is left open for your own independent implementation. Please do include protocol design as a separate section in your project report 'report.txt' and not as a separate file.

Each replica manager must be initialized with 2 data objects X and Y with initial object values of 5 and 9 and object timestamps of 0 and 0 respectively. Remember, each replica manager maintains independent timestamps for every data object it stores.

Also, each process maintains independent log files where it dumps all its outputs. Log files are to be named as FSA1.log, FSA2.log, FSA3.log, RM1.log, RM2.log, and RM3.log. These names are case sensitive so be careful while coding. Your output to log files must follow strictly the log format sample which is specified later.

At each FSA, the read request and the update request that would be made to the remote RM at each execution cycle must be randomly decided using these probabilities.

$P_{read} = 0.75$
$P_{update} = 0.25$

Detailed Behavior Specification for FSA:

Each file service agent / client (FSA) when started on the remote machines would first create a table of available replicas by reading the corresponding connection dump created by the replica managers when they were started, of course this necessitates that the replicas are first started before starting any of the FSA on remote machines. These connection dumps could simply be text files (ex. Sample connection dump could be called RM1_connect.txt). After this each FSA client goes into loop (check for termination trigger) and decides whether to read or update object values using the probabilities defined above. The object selection (X or Y) is also done randomly (with $P_X = 0.5$) where P_X is the probability for accessing object X. It selects which RM

to contact randomly with equal probability. After sending appropriate message to the selected RM, it waits for the response (FAIL or SUCCESS) back from the RM (the RM always responds back with appropriate message). And before going into the next loop cycle sleeps for random amount of time. The sleep duration is randomly selected between (2500 – 5000 milliseconds). For the very **first time** an object value is accessed by any FSA, **it must be a read** operation.

Sample log file entry for FSA 1 –

```
Process Log: FSA 1
Self Host Details: 128.227.248.166 Port: 15281
-----------------------------------------------------------------------
Initiating RM Table Seed Process:
[RM1]-> 128.227.248.167 Port: 8251  [From: RM1_connect.txt]
[RM2]-> 128.227.248.169 Port: 6154  [From: RM2_connect.txt]
[RM3]-> 128.227.248.171 Port: 16523 [From: RM3_connect.txt]
RM Table Seed Process Finished Successfully
-----------------------------------------------------------------------
[ITER1] Sent READ to RM2 for OBJ-X
[ITER1] RECV from RM2 OBJ-X: 5 TS2-X: 0
[ITER1] SELF OBJ-X: NULL TS-OBJ-X: NULL <> RECV VAL MORE RECENT
[ITER1] UPDATE SELF SET OBJ-X: 5 TS-OBJ-X: 0
[ITER1] RANDOM SLEEP: 4862 milliseconds
-----------------------------------------------------------------------
[ITER2] Sent READ to RM1 for OBJ-Y
[ITER2] RECV from RM1 OBJ-Y: 4 TS1-Y: 1
[ITER2] SELF OBJ-Y: NULL TS-OBJ-Y: NULL <> RECV VAL MORE RECENT
[ITER2] UPDATE SELF SET OBJ-Y: 4 TS-OBJ-Y: 1
[ITER2] RANDOM SLEEP: 2900 milliseconds
-----------------------------------------------------------------------
.
.
. (After some iteration) this line is not part of the log file
.
.
-----------------------------------------------------------------------
RECV TERMINATION TRIGGER
WAITING FOR ALL THREADS TO TERMINATE
EXITTING NOW
```

Detailed Behavior Specification for RM:

Each replica manager (RM) when first started starts UDP Server on some random port and then dumps the server connection information in an appropriately named connection dump (it could be a simple text file). It initializes two objects and corresponding timestamps (details above). It then goes into server listen phase where it waits for incoming read / update messages from remote FSA s and gossip messages from peer RM s and acts in accordance with the algorithm specified above. It must also check for termination trigger in order to shut itself down gracefully. It must also run a separate thread through which it propagates the gossip messages for objects X and Y to randomly selected RM peers. This gossip propagation thread runs in a loop with sleep duration randomly selected between [3500 – 5500 milliseconds] between loop cycles. Both objects data is sent in the gossip message to the peer RM. Remember, message protocol design has been left out and it is up to you to implement your own message protocol scheme between RM s and FSA s. In order to send gossip messages to peer RM, one must know the connection details of the peer RM as well. This is left for you to solve, you could either use MULTICAST or some other method of your desire.

Sample log file entry for RM 1 –

```
Process Log: RM 1
Self Host Details: 128.227.248.167 Port: 8251
Initializing OBJ-X OBJ-Y
Server Ready
-----------------------------------------------------------------------
[RECV-READ]-> REQUESTED OBJ-X | ACCESS OBJ-X [VAL: 5 TS: 0]
[SENT-READ]-> Host: 128.227.248.168 Port: 7253 OBJ-X [VAL: 5 TS: 0]
[RECV-GOSP]-> OBJ-Y [VAL: 9 TS: 0] OBJ-X [VAL: 5 TS: 0]
[ACTN-GOSP]-> SELF TS-OBJ-Y >= RECV TS-OBJ-Y [GOSSIP IGNORED]
[ACTN-GOSP]-> SELF TS-OBJ-X >= RECV TS-OBJ-X [GOSSIP IGNORED]
[SENT-GOSP]-> Host: 128.227.248.171 Port: 16523 [OBJ-X: 5:0 OBJ-Y: 9:0]
[RECV-UPDT]-> OBJ-Y [VAL 4 TS: 1] from Host: 128.227.248.170 Port: 8923
[ACTN-UPDT]-> SELF TS-OBJ-Y < RECV TS-OBJ-Y [UPDATE ACCEPT]
[RECV-READ]-> REQUESTED OBJ-Y | ACCESS OBJ-Y [VAL: 4 TS: 1]
[SENT-READ]-> Host: 128.227.248.166 Port: 15281 OBJ-Y [VAL: 4 TS: 1]
.
.
. (After some iteration) this line is not part of the log file
.
.
RECV TERMINATION TRIGGER
WAITING FOR ALL THREADS TO TERMINATE
EXITTING NOW
```

The sample log files above are in no way complete, you must extend the missing actions' corresponding log entries on the similar lines. Failure to follow the log format will result in point penalty from your final grade.

Report:

You are to provide a file named ***report.txt*** that should include:

- Problem definition, proposed solution and how it works
- How you designed and implemented the system
- What are the results you got (in plain English, discuss the output of your system, do not copy & paste the program output)
- All the bugs or problems known, any missing items and limitations of your implementation – IF ANY. (honesty deserves additional points)
- Any additional sections you see necessary

Please note that your reports MUST consist of your own sentences, if you have to copy anything from anywhere you have to quote it and provide reference.

Also a perfect program does not necessarily deserve full points if it is not complemented with a good report. A good report is a brief one that helps its reader to understand the system thoroughly from the problem definition through the limitations.

Submission Guidelines:

- Only submit your Java code files, do not include .class files in your submissions
- Use **tar cvf proj5.tar <*file list*>** to tar your submission.
- Include 1 - 3 page project documentation in .txt format. Name it **report.txt**

- You must submit the tar file after logging into your UF-IBA account. Select ***Project 5*** from the drop down menu and select the appropriate tar file on your local computer before submitting.
- UF-IBA allows file uploads up to 1 Mb in size; make sure your tar file does not exceed the size.
- Please preserve the submission confirmation generated by UF-IBA after successful upload.

Execution Requirements:

- Your Java file containing main() should be named ***start.java***.
- Your program must compile and execute properly on CISE **sand** and **rain** (Solaris) machines. Your projects will **not** be tested on any other machines.
- Execution must begin with command: **java start**
- All your code files must reside at the root level of the project directory and it is advised not to use java ***package*** primitive at the start of your code.
- You will assume all your code files will be expanded at the **root level directory** containing no other subdirectory.
- We will use these script to test your project:

```
---------------------------------------- SCRIPT: RUN----------------------------------------
#!/bin/sh
tar xvf proj5.tar; rm *.class
rm *.log; rm *.txt
javac *.java; java start
---------------------------------------- SCRIPT: PRINT----------------------------------------
#!/bin/sh
cat FSA1.log
cat FSA2.log
cat FSA3.log
cat RM1.log
cat RM2.log
cat RM3.log
cat report.txt
----------------------------------------SCRIPT: CLEAN----------------------------------------
#!/bin/sh
rm *.java; rm *.class
rm *.txt; rm *.tar
rm *.doc; rm *.log
ssh sun114-11 skill java
ssh sun114-12 skill java
ssh sun114-13 skill java
ssh sun114-14 skill java
ssh sun114-15 skill java
ssh sun114-16 skill java
ssh sun114-41 skill java
ssh sun114-42 skill java
ssh sun114-43 skill java
ssh sun114-44 skill java
ssh sun114-45 skill java
ssh sun114-46 skill java
ssh sun114-31 skill java
```

```
ssh sun114-32 skill java
ssh sun114-33 skill java
ssh sun114-34 skill java
ssh sun114-35 skill java
ssh sun114-36 skill java
clear
```

Be careful in using the **clean.sh** shell script because if you are not careful, you could lose all your java files. Do use the scripts provided above after making proper backups of your work. TAs will be not responsible for loss of work because of oversight on your part.

Grading Criteria:

Implementation/Execution/Graceful Termination/Nonexistence of Run-away processes	*75%*
Report	*20%*
Stylish Code/Readability/Comments/Structure and Readability of Output	*5%*
TOTAL:	*100%*

Java Resources and note on MD5:

You may find tutorials and sample code on general Java programming, Threads and TCP/IP programming in Java by following the link http://java.sun.com and searching for specific information you need.

Note on Run-away processes:

Your threads should terminate gracefully. While testing your programs run-away processes might exist. However these should be killed frequently. Since the department has a policy on this matter, your access to the department machines might be restricted if you do not clean these processes.

To check your processes running on a UNIX:	*ps -u <your-username>*
To kill all Java processes easily in UNIX, type:	*skill java*
To check runaway processes on remote hosts:	*ssh <host-name> ps -u <your-username>*
And to clean:	*ssh <host-name> skill java*

Project Termination Trigger:

After all the process have been successfully started on remote machines, **pressing of ENTER key** in start.java execution on sand machine must be treated as termination trigger. How you communicate this trigger to remote processes is left for you to decide. After the termination trigger has been issued, all remote processes must be gracefully terminated and then the start.java should terminate. Make sure that there are no runaway processes as existence of these will definitely result in severe penalty.