Programming Assignment 1 (MP1)
CS 425/ECE 428 Distributed Systems
Spring 2014

Due: By 7:00 p.m. on February 28, 2014
To be submitted electronically.

Instead of the standard 48-hour extension, a 1-week of extension is granted for this assignment. However, note that the second programming assignment may be released soon after February 28, 2014.

Each programming assignment should be performed by a group consisting of two students. If you would like to do a programming assignment individually, please contact the instructor for approval.

The goal of this assignment is to implement a version of the Distributed Snapshot algorithm by Chandy and Lamport, along with logical and vector timestamps. The goal of the MP is to verify whether certain invariants hold true during the execution of a suitable application.

- Implement a simple message-passing application that you will use to test and demonstrate your implementation of distributed snapshot. This application should consist of at least 4 processes that communicate by message-passing. There should be two *invariants* that are true in each consistent global state of the application. For instance, such an application is illustrated in 14.12 of the textbook – for this example application, the invariants are that the total amount of *money* and *widgets* are constant in any consistent global state. The number of processes in the application should be a command-line parameter to the application. The processes should communicate by message-passing, implemented using sockets.

  Your application should be piecewise-deterministic, with the only non-determinism arising from the order of message delivery. Thus, if you use a random number generator, make sure that it is "pseudo-random", meaning that its behavior is repeatable (the application may take a seed as an input parameter)

- Implement Lamport timestamps, and assign a Lamport timestamp to each *message send* and *message receive* event.

- Also implement vector timestamps, and assign a vector timestamp to each *message send* and *message receive* event.

- Implement distributed snapshot mechanism based on the algorithm by Chandy and Lamport. When a process *records its state*, it is adequate to record the state variables needed to verify the invariants of your application.

Your implementation should be able to record multiple snapshots, and distinguish the different snapshots from each other (you may achieve this by attaching sequence numbers to the marker messages). Optionally, consider the possibility of piggybacking markers on application messages (i.e., not sending separate markers). You may designate one of the processes as the initiator for each snapshot.

- You will need to demonstrate an execution in which the processes record at least 5 separate snapshots, with the number of snapshots being a parameter to the application.

- Each process should record its own state and the state of each incoming channel in a file named *snapshot.id*, where id is the identifier of the process. Each process should record this file on its local disk. Since information related to multiple snapshots is being recorded in the same file, each record in the file should be distinguished by a *snapshot number*. Use suitable print statements to that the information in the file is readable as text.

  Print the local state of the process as a separate line in the above file, and similarly record each message (that is part of a channel state) as a separate line, to make it convenient to use commands like "grep" to search through the file. You may make your own format – here is an example of the file at one of the processes of the above example application after the first snapshot is recorded. Logical and vector clock when process state is recorded should be included in the file (as illustrated below). Similarly, logical and vector timestamp associated with a message send event should be recorded along with the message.

  ```
  id 3 : snapshot 1 : logical 2 : vector 0 0 1 1 : money 5  widgets 3
  id 3 : snapshot 1:  logical 1 : vector 0 1 0 0 : message 2 to 3 : money 10 widgets 100
  ```

- After the application records the snapshots and terminates, we want to be able to search the above files for information recorded by all the processes, as part of a particular snapshot number. This will require you to implement a separate utility that lets you search files on all the machines where the files are recorded. For instance, if the utility is called "search-all" then by using the command below (or something similar to that)

  ```
  search-all "snapshot 1"
  ```

  it should be possible to print on the screen all the information that belongs to snapshot 1, including channel and process state recorded by all the processes. You may include additional parameters above (or use an input file) to specify the machine names, and the filepath for the files to be searched. You may use commands such as "grep" to implement the above utility. You may find this utility useful in future assignments as well (for debugging purposes).

The above specification leaves some flexibility in deciding exactly how the above application and utility may be implemented.

You will need to demonstrate your code to the course staff. The following aspects need to be demonstrated:

- Ability to control the number of snapshots taken (using a command-line parameter).

- The ability to search the recorded files for information related to a specific snapshot.

- Specify the invariants of your application, and show that each snapshot satisfies the invariants of your application. To be able to easily verify the invariants, we recommend that you use an easy to understand format for saving the state (e.g., such a format is illustrated above).

Programming Language: Please select from C, C++, Java, Python.

Grading Policy:

An application with two invariants: 15%
Logical and vector timestamp implementation: 10%
Snapshot algorithm (implementation and correctness): 30%
Snapshot output stored in human-readable (text) format: 5%
Search utility: 25%
Miscellaneous (comments, coding style, readme): 5%
Demo: 10%

Submission instructions: Please submit a zip file including all source code & readme & results. The zip file should be named as “netid1_netid2_mp1.zip” where netid1 and netid2 are identifiers of the teammates. **Additional submission instructions to be provided later.**

Readme.txt: Please include a readme.txt to include the information of group members, simple introduction to the algorithm you used, direction to compile the source code, and command line input format.

Demo: Each group should sign up for a time slot for MP demo (a sign up page will be released later). All group members should present at the demo.