CS/CE/TE 6378: Advanced Operating Systems
Section 002
Project 2

Instructor: Neeraj Mittal

Assigned on: Tuesday, February 15, 2017
Due date: Tuesday, March 22, 2017

This is a group project. *Code sharing among groups is strictly prohibited and will result in disciplinary action being taken.* Each group is expected to demonstrate the operation of this project to the instructor or the TA. You will be graded not only on the correct behavior of your project, but also on your responses to questions concerning your implementation which the instructor or TA may ask during your demo.

You can do this project in C, C++ or Java. Since the project involves socket programming, you can only use machines dcXX.utdallas.edu, where XX $\in$ {01, 02, .., 45}, for running the program. Although you may develop the project on any platform, the demonstration has to be on dcXX machines; otherwise you will be assessed a penalty of 20%.

1 Project Description

This project consists of three parts.

1.1 Part 1

Implement a distributed system consisting of n nodes, numbered 0 to $n-1$, arranged in a certain topology. The topology and information about other parameters will be provided in a configuration file.

All channels in the system are bidirectional, reliable and satisfy the first-in-first-out (FIFO) property. You can implement a channel using a reliable socket connection (with TCP or SCTP). For each channel, the socket connection should be created at the beginning of the program and should stay intact until the end of the program. All messages between neighboring nodes are exchanged over these connections.

All nodes execute the following protocol:

- Initially, each node in the system is either *active* or *passive*. **At least one node must be active at the beginning of the protocol.**
- While a node is *active*, it sends anywhere from minPerActive to maxPerActive messages, and then turns *passive*. For each message, it makes a uniformly random selection of one of its neighbors as the destination. Also, if the node stays *active* after sending a message, then it waits for at least minSendDelay time units before sending the next message.

- Only an *active* node can send a message.
- A *passive* node, on receiving a message, becomes *active* if it has sent fewer than maxNumber messages (summed over all *active* intervals). Otherwise, it stays *passive*.

We refer to the protocol described above as the MAP protocol.

1.2 Part 2

Implement the Chandy and Lamport's protocol for recording a consistent global snapshot as discussed in the class. Assume that the snapshot protocol is always initiated by node 0 and all channels in the topology are bidirectional. Use the snapshot protocol to detect the termination of the MAP protocol described in Part 1. The MAP protocol terminates when all nodes are *passive* and all channels are *empty*. To detect termination of the MAP protocol, augment the Chandy and Lamport's snapshot protocol to collect the information recorded at each node at node 0 using a converge-cast operation over a spanning tree. The tree can be built once in the beginning or on-the-fly for an instance using MARKER messages.

Note that, in this project, the messages exchanged by the MAP protocol are *application* messages and the messages exchanged by the snapshot protocol are *control* messages. The rules of the MAP protocol (described in Part 1) only apply to application messages. They do not apply to control messages.

Testing Correctness of the Snapshot Protocol Implementation

To test that your implementation of the Chandy and Lamport's snapshot protocol is correct, implement Fidge/Mattern's vector clock protocol described in the class. The vector clock of a node is part of the local state of the node and its value is also recorded whenever a node records its local state. Node 0, on receiving the information recorded by all the nodes, uses these vector timestamps to verify that the snapshot is indeed consistent. Note that only application messages will carry vector timestamps.

1.3 Part 3

Design and implement a protocol for bringing all nodes to a halt after node 0 has detected termination of the MAP protocol.

2 Submission Information

All the submission will be through eLearning. Submit all the source files necessary to compile the program and run it. Also, submit a README file that contains instructions to compile and run your program. You should also include `launcher.sh` and `cleanup.sh` scripts which run your program on the machines as configured and cleanup your processes similarly to the prior project.

3 Configuration Format

Your program should run using a configuration file in the following format:

The configuration file will be a plain-text formatted file no more than 100kB in size. Only lines which begin with an unsigned integer are considered to be valid. Lines which are not valid should be ignored. The configuration file will contain $2n + 1$ valid lines. The first valid line of the configuration file contains **SIX** tokens. The first token is the number of nodes in the system. The second and third tokens are values of minPerActive, and maxPerActive respectively. The fourth and fifth tokens are values of minSendDelay and snapshotDelay, in milliseconds. snapshotDelay is the amount of time to wait between initiating snapshots in the Chandy-Lamport algorithm. The sixth token is the value of maxNumber. After the first valid line, the next n lines consist of three tokens. The first token is the node ID. The second token is the host-name of the machine on which the node runs. The third token is the port on which the node listens for incoming connections. After the first $n + 1$ valid lines, the next n lines consist of a space delimited list of at most $n - 1$ tokens. The k^{th} valid line after the first line is a space delimited list of node IDs which are the neighbor of node k. Your parser should be written so as to be robust concerning leading and trailing white space or extra lines at the beginning or end of file, as well as interleaved with valid lines. The # character will denote a comment. On any valid line, any characters after a # character should be ignored.

You are responsible for ensuring that your program runs correctly when given a valid configuration file. Make no additional assumptions concerning the configuration format. If you have any questions about the configuration format, please ask the TA.

Listing 1: Example configuration file

```
# six global parameters (see above)
5 6 10 100 2000 15

0 dc02 1234    # nodeID hostName listenPort
1 dc03 1233
2 dc04 1233
3 dc05 1232
4 dc06 1233

1 4       # space delimited list of neighbors for node 0
2 3 4     # space delimited list of neighbors for node 1
0 1 3     # ...                                   node 2
0 4       # ...                                   node 3
1 3       # ...                                   node 4
```

4 Output Format

If the configuration file is named `<config_name>.txt` and is configured to use n nodes, then your program should output n output files, named in according to the following format:
`<config_name>-<node_id>.out`, where `node_id` $\in \{0, ..., n - 1\}$.

The output file for process j should be named `<config_name>-j.out` and should contain the following: If your program took m snapshots, then each output file should contain m lines. The i^{th} line should contain the vector timestamp of the i^{th} snapshot *as seen by process* j. Each line of the output file should contain n space delimited tokens, each of which should be a non-negative integer. In each line, the timestamps must appear in increasing order of process id. That is, the k^{th} number in the i^{th} line should be the timestamp value for process k. An example file is described below.

Listing 2: Example output file for 7 nodes

```
0 4 3 6 0 2 3
3 7 6 7 2 4 4
6 9 11 10 5 7 5
8 12 14 23 8 10 7
```

In this example, the first snapshot has vector clock value $\langle 0,4,3,6,0,2,3\rangle$; the value of process 0's clock was 0, and the value of process 3's clock was 6.

During your demonstration, the TA will provide a bash script which will be used to check the consistency of the timestamps in the output files your program produces. This script will be named `grade.sh` and will take one command line argument: the relative path to a configuration file. The script will expect to find properly formatted output files which have been named correctly in the same directory from which it is run. The TA will provide you with a dummy script which you can use to ensure you know how to use the script during your demonstration.