CSCI 565: Distributed Systems

Programming Assignment: Implementing Maekawa's Mutual Exclusion Algorithm

1 Overview

In the previous programming assignment, you have gotten practice communicating between processes in a distributed system.

In this programming assignment, you will move on to implement some classic (failure-free) distributed protocols. You will implement a mutual exclusion solution for a distributed system using two protocols: vector timestamps and Maekawa's algorithm version 1. By the time you finish this assignment, you will not only understand how these protocols work, but you will have experience implementing theoretical algorithms in practice.

You are allowed to complete this assignment in Java, C++, or Python.

Before starting this assignment, you should review relevant textbook chapters on vector clocks (Chapter 6.3) and mutual exclusion (Chapter 7.2.3).

2 Assignment Details

In this programming assignment, you will implement vector timestamps (and output as specified) and implement Maekara's algorithm.

The machines will be trying to obtain a distributed lock, which will be managed by Maekawa's algorithm. We will use a peer to peer architecture. The code you will write will be identical on each machine, and any machine will be capable of requesting the lock.

You should structure your code in this way: The Maekawa's algorithm will be implemented in the form of a library, and you also write outside code that calls your implementation of the Maekawa's algorithm. Your library code (i.e., the implementation of the Maekawa's algorithm) will not execute as a separate process. Rather, it will be linked in with an application that can call the library methods to achieve what it needs. The goal here is to make it so the application does not need to include any networking or distributed systems code. The library's initialization methods should take care of establishing any network connections that are necessary and the lock/unlock methods should take care of communicating across those connections to achieve mutual exclusion in the system.

Your library code should be flexible enough to allow any number of processes to run the system. You may assume that there will only be one participating process running on any particular machine. You can also assume that the number of processes involved will be fixed at startup. We will not add processes to the system after it is up and running. You should implement your library so that it will allow programs to run on multiple machines.

We recommend that you use sockets for this programming assignment.

We next outline each of the algorithms.

2.1 Vector Clocks

You will be implementing vector clocks as described in lecture. Whenever a message is sent in Maekawa's algorithm, the local vector clock entry should be incremented and a timestamp should be attached to the message. Whenever a message is received, the local vector clock should be updated by taking an element-wise maximum of the local clock and the message's timestamp, then incrementing the local entry in the vector clock.

We will test your vector clock implementation using your printed output (see the "Output" section below), and the vector clocks will also be used when implementing Maekawa's algorithm. Aside from its use in the mutual exclusion algorithm, there will be no other messages that will affect the vector timestamps. Also note that the vector clocks should be updated properly over the life of the process; they should not clean up whenever the mutual exclusion algorithm does.

Assume that the clocks are initialized to all zeroes. The number of processes in the system can be determined via the global initialization method, and the order of processes is determined using the priority ordering (see the "Interface" section below).

2.2 Maekawa's Algorithm

You must implement Maekawa's algorithm (also called *maekawa1*) as described in the textbook. A loose outline of the algorithm follows (see the attached original paper for details):

Requestor

- Multicast request to other processes in my group

Participants receiving request

- If in CS, enqueue request
- If my vote has been given to someone else, enqueue request
- Else vote for requestor by replying OK

Requestor Entry

- Once received OK from all other processes in my group, enter CS

Exit

- On exit from CS, multicast Release message to group

Participant receiving release message

- If my queue has requests, dequeue and vote for the first entry by replying OK
- Else, my vote is now free

Each requestor looks for all of the votes in its quorum, and enters the critical section once it has them. Safety is guaranteed in Maekawa's algorithm by cleverly choosing quorums for each process

such that they are pairwise non-disjoint. You need to implement these groups. Section 7 of the original paper [1] has proposed two different methods to create these groups. You can implement any one of them.

2.3 Interface

Your implementation of these algorithms will be in a library that is accessed using the IDistributedMutex interface described below.

```
interface IDistributedMutex {
public void GlobalInitialize(int thisHost, InetSocketAddress[] hosts);
public void QuitAndCleanup();

//Maekawa methods
public void MInitialize(int[] votingGroupHosts);
public void MLockMutex();
public void MReleaseMutex();
public void MCleanup();
}
```

These functions will be called as follows:

- **GlobalInitialize()** will be called once when the process is started, and it will not be called again. The array *hosts* will contain information for all of the participants in the system (host, and port). These hosts will be ordered in terms of priority, with lower index having higher priority. This means that the process at the first position *hosts[0]* will have the highest priority. Assume that this array will be identical and sorted the same at each process. *thisHost* contains the position (index) of the current host in the *hosts* array.
- **QuitAndCleanup()** will be called once when you are done testing your code.
- **MInitialize()** will be called once when starting to test Maekawa's. May be recalled multiple times, as long as the corresponding cleanup algorithm. **MCleanup()** is done after each initialization. The *votingGroupHosts* array contains an index (to the *hosts* array) for each host in the voting set/group of the current host.
- **MLockMutex()** initiates a request for the critical section for Maekawa's algorithm. **MReleaseMutex()** exits the critical section once it is obtained.

You need to implement a class named CDistributedMutex that implements this interface.

You should implement the MLockMutex() methods as blocking calls that only return to the caller once the lock has been obtained. You may assume that an application will never generate more than one request for a critical section at a time. An application will also not try to release a mutex that it has not previously requested.

For a particular process, you can think of the *hosts* array index like a process ID. For vector timestamps, the first entry should be for the highest priority process (*hosts[0]*), the second for the next highest priority process (*hosts[1]*), and so on.

Also note that the *votingGroupHosts* may change between different initializations of Maekawa's on the same process.

Use the global and algorithm specific initialization and cleanup methods however you like. A typical execution of library code by an application will have calls in the following order:

1. *GlobalInitialize(...)*
2. *MInitialize(...)*
3. some number of calls to *MLockMutex()* and *MReleaseMutex()*
4. *MCleanup()*
5. ... repeat steps 2-4 arbitrarily many times ...
6. *QuitAndCleanup()*.

3 Output and Testing

You're required to output to console (stdout) *whenever a vector timestamp is updated.* This will occur whenever a message is sent or received in either one of your algorithms. Here is some generic example output for *host[0]* which is at "ctb60-11.mines.edu" with 4 processes in the system (note that these particular timestamps may never arise for the algorithms that we are considering):

```
ctb60-11.mines.edu:(1,0,0,0)
ctb60-11.mines.edu:(2,0,0,0)
ctb60-11.mines.edu:(3,0,4,2)
ctb60-11.mines.edu:(4,0,4,2)
ctb60-11.mines.edu:(5,2,4,3)
```

To make grading easier, your output should be formatted identically to the above (i.e., no spaces, each entry on a new line). In addition, please print out your group formation.

To test your code, we will run several processes (each running on its own machine) each of which will be linked with your library code. Our test application will then call the appropriate initialization methods and begin trying to access the critical section using the methods of the library.

4 Deliverables

You need to hand in all of your source files, a README file, and a Makefile if you use C/C++. Zip up these files and submit to Canvas. Do not submit your object files or executables!

Your README should describe your design, list any assumptions you made, and describe how to test your project in detail. If we cannot figure out how to compile and run your project based on what you have written in the README, we will grade it as if it is broken. In addition to documenting your design in the README, you should also thoroughly comment your code such that with the background information given in the README and the comments written in the code, we can figure out exactly what your code is doing.

if you use C/C++, your Makefile should correctly compile the source files and produce any necessary files. If necessary, you may provide any particular instructions in the README. If you use Python, please include Python version number in your README.

5 Grading

- README (along with Makefile for C/C++ implementation): 20 points
- Implementation: 80 points (including 20 points for vector timestamps and 60 points for Maekawa's algorithm).

  If you have any questions, feel free to post them to the forums on Ed Discussion. We prefer that you post questions to Ed Discussion rather than sending emails because others in the class might also benefit from the answers. However, please do not post any source code there.

  You're strongly encouraged to form a group of two for this programming assignment. Everyone in the same project group will get the same grade for this assignment.

6 Hints and Clarification

- Read the original paper [1] of the Maekawa's algorithm (version 1) for details.
- We are assuming there are no failures!
- The mechanisms behind Ricart-Agrawala and Maekawa's are very similar, so it may be useful to look into how Ricart-Agrawala works before analyzing and implementing Maekawa's.
- Once you have thought through this assignment, here is one possible approach to implementing it all:
  1. Setup point-to-point messaging between two machines.
  2. Setup multicasting from one machine to a group.
  3. Implement vector timestamps
  4. Implement Maekawa's, using the vector timestamps implementation.
- You should test your clients and servers on ALAMODE lab machines. Please refer to the first programming assignment on how to access those machines.

References

[1] Mamoru Maekawa, "A $\sqrt{N}$ algorithm for mutual exclusion in decentralized systems", ACM Transactions on Computer Systems, Vol. 3, No. 2, May 1985, Pp. 145-159.