Distributed Mutual Exclusion

In this project, you will implement a version of a distributed mutual exclusion algorithm. The distributed mutual exclusion algorithm can be based on Lamport clock or Vector clock. This distributed algorithm will be used to obtain the permissions to update a central database, which, in our case, resides on /aos space in one machine. Several processes will communicate with each other in different machines, and this will be done through servers running on these machines. The servers connect to each other using sockets, and implement the processes requesting the critical section.

The potential applications for distributed ME are numerous and include: sign-up sheet for the project, distributed control of scheduling appointments, distributed control of DBs (e,g., airline reservation system), distributed administration of mailing lists.

Technical Details

There is a central database (e.g., a file called /aos/cs/schedule.txt) that contains information to be updated (e.g., a sign-up sheet for demos of the project).

Each of the *n* machines will have a server, and this server communicates with other servers in other machines that belong to the same group. You may assume that the machines are well known to all the servers in the group. You can use the select system call in order to check which server wants to communicate at a particular time.

Each process that needs to update or read the database, must go into a negotiation phase, during which the distributed ME algorithm will be executed. After getting permission to enter the Critical Session (CS), the server can tell the client it is the client's term to update the database. Upon update completion, the server is notified. Since there are several clients connecting to each server in a particular machine, the server will order the requests from the client as a sequencer (e.g., centralized FIFO lock manager).

Since many clients may be simply reading the database, your protocol should be able to (consistently) resolve the conflicts of the read/write possibilities. It should not block the readers if there are only readers accessing the database (readers/writers problem).

Requirements

The program should be able to

1. update a file residing in aos space, without overwriting the work of others
2. allow multiple readers to read the file, concurrently
3. disallow readers when updates are underway.

You should also hand in three documents

1. a half-page explanation of your protocol for the readers/writers solution, that is, why you choose to implement your specific priority scheme
2. a document with a performance analysis of your implementation, including scalability
3. your code and test harness

Extra Credit (25 points)

In the class, we also discuss the leader election in distributed systems. We can implement the mutual exclusion by letting the leader decide who enters CS. You can implement the leader election using either brutal algorithms or ring-based algorithm. The program requirements are the same as distributed mutual exclusion.

You should also hand in three documents for extra credit

1. a half-page explanation of your protocol for the readers/writers solution
2. a document with a performance analysis of your implementation, including scalability
3. your code and test harness