Matt Viele, mtv364
Royce Li, rl26589
Robbie Zuazua, raz354

EE 360P Term Paper

Abstract: An important control function for distributed systems is the process of electing a leader. Oftentimes, leader election algorithms do not select their leader fairly which leads to predictable leader selections. Our algorithm aims to solve this issue by using randomization to select leaders. This method led to unpredictable leader selections every iteration of the algorithm which made leader election more fair and secure.

Many distributed systems execute control functions through a process, also known as a "leader", that serves as the coordinator for problems such as deadlock or mutual exclusion. The election process can be compared to breaking symmetry in a system to solve a problem. In the mutual exclusion problem, solutions often involved either a token or a coordinator. While the mutual exclusion problem is similar to the leader election problem in which there is one "special" process doing the work, deciding which process actually holds the token or is the coordinator is exactly what the leader election problem is. Currently, there exists leader election algorithms such as the Chang-Roberts algorithm that, while efficient, do not give every process a fair chance at being the leader. Our solution uses randomization to make the leader selection process less predictable while still maintaining algorithm efficiency.

In this paper, we begin by providing a more complete description of our project. We introduce our algorithm and compare it with ones previously mentioned. In the next section, we discuss different approaches we took in implementing randomization and other characteristics of our algorithm. Last but not least, we evaluate the performance and efficiency of our algorithm and how it compares to existing ones.

Matt Viele, mtv364
Royce Li, rl26589
Robbie Zuazua, raz354

As mentioned previously, the goal of our algorithm is to elect a leader out of various processes through randomization. Similar to the Chang-Roberts algorithm, every process has an unique ID, called a process ID. However, we added a second ID, called the election ID, that is a random number between 0 and 2147483647 (2^31-1). This means it could be any number within the entire range of possible int values. Both of these characteristics belong to every one of our processes, or “electors”, in a class called RandomElector. Each elector also has a boolean awake variable and leaderID. The awake variable determines if the process is running or not. The leaderID keeps track of the current leader (if there is one). In our constructor method for every RandomElector, we initialize the awake variable to false, the leaderID to -1, and randomize an electionID.

When creating our algorithm, we were able to divide it into three main methods: the initiateElection method, handleElectionMessage method, and handleLeaderMessage method. In the initiateElection method, the process sends a message to the other processes that it is waking up and sets its awake variable to true. The goal of the handleElectionMessage method is to determine who the leader is. The current process checks its electionID against the electionID of the process sending the message. If the current election ID is larger, it’s not the leader and sends the message to the next process. If the current election ID is smaller, it sends a new message that has its own process ID and election ID. If the election IDs are equal, process IDs are checked. Every process is guaranteed to have a unique process ID. If the current process ID is larger, it’s not the leader and sends the message to the next process. If the current process ID is smaller, it sends a new message that has its own process ID and election ID. If the process and election IDs are equal, then the current process knows that it is the leader because it has received its own message back and sends a leader message to be

Matt Viele, mtv364
Royce Li, rl26589
Robbie Zuazua, raz354

passed around all the processes. The handleLeaderMessage method is called when a leader sends a leader message around to the processes. The process that receives this message acknowledges the leader by setting the leaderID variable to the ID of the leader. Then, if it is not the leader, it passes the message on. These three methods handle the bulk of the functionality for our algorithm.

While implementing our algorithm, we considered various alternatives regarding how we wanted to implement randomization. We first considered only having one ID that was randomized from the start. While this could have worked for most cases, especially since there is a very small possibility of a tie when randomizing in the range of 0 to 2147483647, this implementation would have no way of breaking a tiebreaker between two ids. Therefore, we decided to add an additional id called the process id. No two processes could have the same process ID. While creating our algorithm, we used some similar code to what we had in lab 4 because we felt like the two labs had similarities that we could take advantage of. This gave us a great starting point as we already knew how to make processes communicate with each other and only had to worry about the main algorithm.

When testing ourcode, we created a Process.java class to create multiple processes and test if our algorithm worked. First, we started with two processes with unique election IDs and ensured that they could choose a leader between themselves. From there, we slowly upped the process count to make sure it wouldn’t crash at a certain number of processes. Through these tests, our algorithm worked like how it was supposed to and always chose the correct leader based on election IDs. Next, we had to test our tiebreaker system, where when two processes have the same election ID the process with a smaller process ID would have priority to be the leader.

These tests worked great as well for any number of processes. While the algorithm worked for any number of processes that are up and running, we discovered an improvement that could be made in a future iteration of our code. One big improvement that could be made is making our algorithm fault tolerant. Currently, it only works if every process is up and running, but will not start until that happens. That means every process needs to be online and ready before we can select a leader. Ideally, we would be able to start and wait for either that process to get online or choose a leader without it. From a time standpoint, our algorithm was also efficient in that regard as we do not have many time consuming elements in our code.

In conclusion, current leader-elect algorithms do not do a good enough job of selecting a leader fairly and oftentimes will have predictable results. The objective of our algorithm was to use randomization for selecting a leader fairly and we did this by randomizing an elector ID for every process. Tests run on our algorithm so far have all passed, but a big improvement that can be made is making the algorithm fault-tolerant. Our algorithm accomplished our goal and can be used to elect a leader in programs that run into deadlock or mutual exclusion problems.

Matt Viele, mtv364
Royce Li, rl26589
Robbie Zuazua, raz354

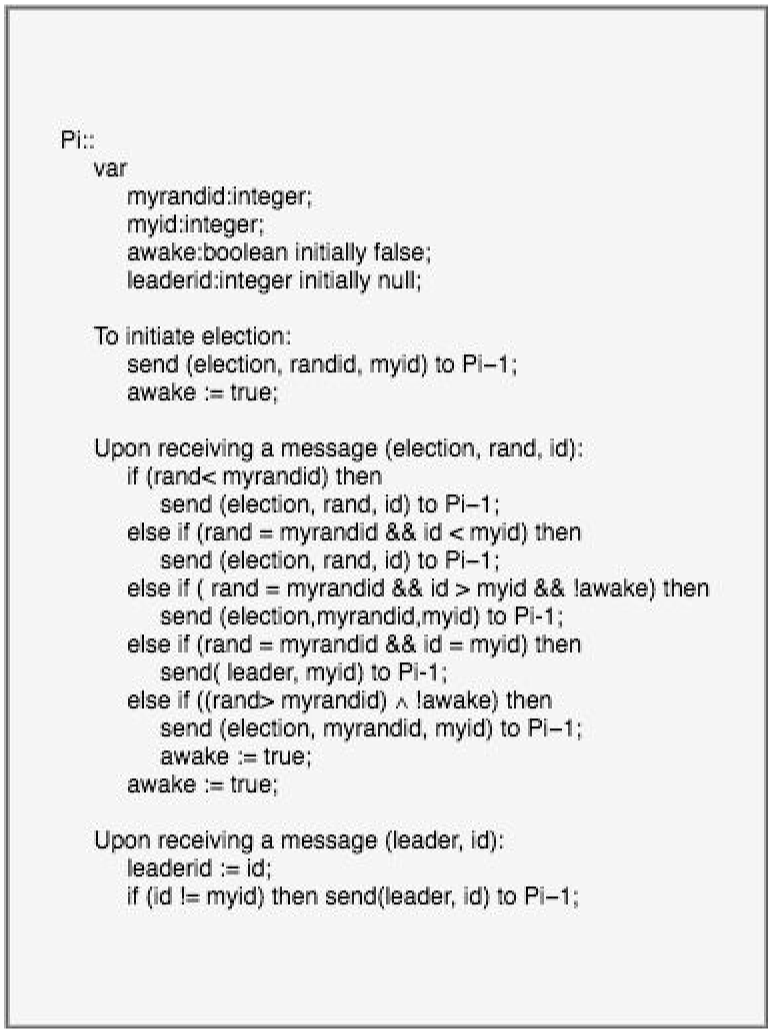
Appendix A - Educational Resources: Supplementary Material to Chapter 13

13.2.3 Random Leader Election Algorithm

The Chang-Roberts and Hirshberg-Sinclair algorithms choose leaders based on process ID's. Notice that this method represents an unfair system since the processor with the highest ID will always win the election. Now assume that we want to choose a leader at random such that a leader can be chosen irrespective of it's process ID. The benefits of random leader election are creation of a fair system and added security since the process leader will not be known based on IDs.

In this algorithm, every process must have a unique identifier which will be used to break ties if two processes choose identical random numbers. The algorithm shown in Figure 13.3, is a one-directional ring based algorithm that extends the Chang-Roberts algorithm. In this algorithm, every process sends messages only to its left neighbor and receives messages from its right neighbor similar to the Chang-Roberts algorithm. However, an additional field has been added to each processor - a random number between 0 and 2^31-1. Any process can initiate an election if it has not yet seen a message with a smaller random number. Similar to Chang-Roberts, it forwards any message it receives that has a smaller random number and swallows messages with higher random numbers. Once a message has traveled around the whole circle without being swallowed, the process that sent the message will receive it. Only then can it declare itself as the leader by sending a leader message that everybody will pass on. If a process receives a message with a random number equivalent to its own, it will pass on the message if the other process ID is smaller than its own.

Matt Viele, mtv364
Royce Li, rl26589
Robbie Zuazua, raz354



```
Pi::
    var
        myrandid:integer;
        myid:integer;
        awake:boolean initially false;
        leaderid:integer initially null;

    To initiate election:
        send (election, randid, myid) to Pi−1;
        awake := true;

    Upon receiving a message (election, rand, id):
        if (rand< myrandid) then
            send (election, rand, id) to Pi−1;
        else if (rand = myrandid && id < myid) then
            send (election, rand, id) to Pi−1;
        else if ( rand = myrandid && id > myid && !awake) then
            send (election,myrandid,myid) to Pi-1;
        else if (rand = myrandid && id = myid) then
            send( leader, myid) to Pi-1;
        else if ((rand> myrandid) ∧ !awake) then
            send (election, myrandid, myid) to Pi−1;
            awake := true;
        awake := true;

    Upon receiving a message (leader, id):
        leaderid := id;
        if (id != myid) then send(leader, id) to Pi−1;
```

Figure 13.3: Random leader election algorithm at Pi.