Implementation of a Leader Election Algorithm

Afonso Reis, David Ferrao, Vitor Fernandes

***Abstract*— This document corresponds to the written report of the final project of Distributed Systems course unit. Our group implemented a Leader Election Algorithm proposed by [1]. In this paper, there are two protocols proposed with weak assumptions on initial knowledge, communication reliability and synchrony. The first one is *inefficient in terms of communication*, i.e., processes that are correct will keep sending messages indefinitely. The second one is *communication efficient* meaning that it is guaranteed that, after some running time τ, there will be only one process sending messages (the elected leader). Both protocols assume that they only know their own process ID and they aren't aware of the total number of processes in the network. They also know that the rest of the processes in the network have a process ID. This implementation was done using *Java* API, since for each process we have a different Virtual Machine running. In terms of communication, all pairs of processes have two directed links that are classified as *fair lossy links*. This communication network was implemented using UDP Multicast Sockets that simulate the time and resources limitations intrinsic to the proposed type of links. Simulations were made in order to also extract different figures of merit to characterize the proposed solutions.**

I. Problem Definition

This implementation uses two different protocols in order elect a leader. Both protocols assume that:

- Each process knows their own process ID and it is not aware of the presence of the remaining processes, i.e., how many there are in the network (**K1**);
- Communication is asynchronous and unreliable (timeouts can occur);
- Not all processes are correct, i.e., some can crash. Maximum number of crashed processes in the algorithm is defined as t and the current number of crashed processes in the network is f;
- There is a local variable $silent_k$ that contains all processes that crashed or incurred in a timeout; and a local variable $members_k$ that contains all correct processes;

A. First protocol - Communication Inefficient

The first protocol also has an additional initial information. Each process is aware of a *maximum number of possible crashed processes* (α) (**K2**), described as:

$$\alpha = n - t$$

Since, initially, a process can not reach another through its directed links, each one has a *broadcast primitive* that it uses to reach the remaining processes in the network.

Processes trade sequenced $state_k$ messages that contain:

- k, contains process k identifier;
- sn_k, number of sequence of the message;
- $susp\_level_k[j]$, suspicion level of process k of all processes j that belong to $members_k$;
- $silent_k$;

In order to measure the level of suspicion of a specific process j, process k has two local variables:

- $timer_k[j]$, that has all running timers of each process known to k in this network;
- $timeout_k[j]$, that has a previous established timeout value in order to detect if process j is responding. This timeout is increased by $\frac{\eta}{10}$ every time $timer_k[j]$ expires. This is not what is suggested in the paper (it suggests timeout is increased by "1"), but we think that an increase of 1 in a 1000ms (for example) is too small and would be quite inefficient.

Besides that, each process has an additional two local variables to help him decide which process is the current leader:

- $susp\_level_k[j]$, that is the level of suspicion process k has on process j;
- $suspected\_by_k[j]$, that contains the processes g that suspect j;

The variable $susp\_level_k[j]$ is increased each time a timeout of process j occurs, it is updated according to the information received through a message $state_j$ (it sets its own $\text{susp\_level}_k[j]$ according to the $\text{susp\_level}_g[j]$, maximum of both) and when the number of processes $suspected\_by_k[j]$ reaches the value α.

When an upper layer application uses the *leader()* function call, a process returns the minimum value of $susp\_level_k[j]$ and the process identifier of j. The last one is used to untie if two or more processes have the same *susp_level*, returning the one who has the lowest value.

B. Second Protocol - Communication Efficient

The second protocol is not aware of **K2**. It contains the same local variables as protocol one ($susp\_level_k[j]$, $suspected\_by_k[j]$, $timer_k[j]$ and $timeout_k[j]$). It also contains an additional group of processes called $contenders_k \in members_k$.

Each process has a local variable called hbc_k that contains the amount of cycles the process considered himself the leader. Besides, it also has $last\_stop\_leader_k[j]$ that stores the hbc_k of the remaining processes in the network.

There are three types of messages sent:

- *heartbeat*, sent periodically to inform the remaining processes that it considers himself the leader;
- *suspicion*, sent when process k suspects of process g (when $timer_k[g]$ expires) ;

- *stop_leader*, sent only once, when process k does not consider himself the leader anymore;

A process k will remove process j from $contenders_k$ if a $timer_k[j]$ expires or it receives a *stop_leader* state message from process j.

Another difference in relation to protocol one is the structure of the transmitted $state_k$ messages:

- tag_k, type of message;
- k , process k identifier;
- sl_k, suspicion level of process k;
- $silent\_k$, sent on a suspicion level ($silent\_k = j$, process k suspects process j);
- hbc_k;

In this protocol, process k increases its $susp\_level_k[j]$ when it receives a *suspicion* message from another process g. The call of *leader()* will return in the same manner as the first protocol.

This implementation assures that, after some time, only one process will keep sending *hearthbeat* messages. Both protocols assume that, eventually, when *leader()* is called from an upper layer application, all processes k will return the same leader, process l.

II. Solution

Our implementation of both protocols was made using *Java* language with NetBeans IDE. Two separate projects were made in order to develop both protocols.

Both protocols have two different tasks (**T1**, **T2**) that have different purposes. Since they had to operate independently, specific threads were assigned to those tasks in order to facilitate operation and processing.

Concurrent variables [3] were defined in order to deal with race condition issues when both threads try to access the same variable. Since there is not a guaranteed order of execution in both threads, in a case where a local variable is not defined and a thread tries to access it (e.g. a thread wants to save a timeout value), this operation is discarded. An alternative would be to hold the thread until such variable is initialized but it would mean an execution delay.

In [1], steps of tasks **T1** and **T2** are defined as a set of instructions of pseudo-code that demonstrates the algorithmic logic. Our method of implementation implied the analysis of the proposed pseudo code and programming using each item as a reference.

Code shown is commented to facilitate a better understanding of the algorithm and execution flow. Steps were defined to better situate the implementation on the protocol.

A *leader* call answer is implemented in the same way in both protocols. It returns the variable *leader* that is the minimum value in *susp_level*, using process ID as a tie breaker:

```
public int leader(){
  int min = 999999;
  int leader = 0;
  for(int j : members){
    if(susp_level.get(j) < min ){
          min = susp_level.get(j);
            leader = j;
        }
        else if(susp_level.get(j) == min && j <
            leader) leader = j;
    }
    return leader;
}
```

A. First protocol - Communication Inefficient

The *message* class for this protocol was implemented as follows:

```
public class message implements Serializable{
    public int k;
    public int snk;
    public ConcurrentMap<Integer,Integer>
        susp_level;
    public CopyOnWriteArrayList<Integer> silent;
}
```

For **Task 1** of the first protocol:

```
public void task1(){
  //step 01 - Increase sequence number
  sn++;

  //step 02 - Put process j in suspected_by i
  for (int j : silent){
    CopyOnWriteArrayList<Integer> list =
        suspected_by.get(j);
      if(list == null) list = new
          CopyOnWriteArrayList();
    list.add(this.pid);
      suspected_by.put(j,list);
  }

  //step 03 - Compare values of suspected_by with
      alpha
  CopyOnWriteArrayList alaux;
  for(int j : members){
    alaux = suspected_by.get(j);
      if(alaux != null){
        //step 04 - Increase value of susp_level
            and reset if condition is verified
        if (alaux.size() >= this.alpha) susp_level.
            put(j,susp_level.get(j)+1);
          suspected_by.put(j,new
              CopyOnWriteArrayList());
      }
  }

  //step 05 - Update state message
  message aux = state.get(this.pid);
  aux.k = this.pid;
  aux.snk = this.sn;
  aux.susp_level.putAll(this.susp_level);
  aux.silent = (CopyOnWriteArrayList<Integer>)this.
      silent.clone();
  state.put(this.pid,aux);

  //step 06 - Broadcast state message (aux)
  this.csocket.broadcast(aux);

  //step 07 - Reset necessary timers
  for(int j : this.to_reset){
    Timer t_aux = this.timer.get(j);
      if(t_aux != null){
```

```
        t_aux.cancel();
        t_aux.purge();
      }
  if(this.timeout.get(j) != null) this.timer.put(j,
      create_timer(j));
  }
  this.to_reset = new CopyOnWriteArrayList();
  //COMPLETE
}
```

Task 2 is responsible for the handling of messages that have arrived and the verification of timeout occurrences. It updates process $state_k$ and local groups ($silent_k$ and $members_k$) according to information received. Sequence numbers guarantee that the handling of information is done in an ordered way.

Timeout occurrences were implemented as follows:

```
public void updateTimeout(int j){
  //step 08 - Increase timeout value and, if j
      doesnt belong to silent i, add it
  timeout.put(j,timeout.get(j)+1);
  if(!silent.contains(j)) silent.add(j);
}
```

For message reception (main part of **Task T2**), steps 9-19 were implemented:

```
 public int processMessage(message msg){
  //step 11 - replace current state message with
      values acquired from the received message
  if(this.members.contains(msg.k)){
      //step 09 - check if message is correctly
          sequenced
      if(msg.snk <= this.state.get(msg.k).snk){
        return 0;
      }
      this.state.put(msg.k,msg);

      //step12 - stop timers, set it to be reseted
          and remove process from silent if in it
      this.to_reset.add(msg.k);
      if(this.silent.contains(msg.k)){
          this.silent.remove((Object)msg.k);
      }
      Timer t_aux = this.timer.get(msg.k);
      if(t_aux != null)  t_aux.cancel();
  }
    else{
      //step13 - update state of current process
          with information acquired in message
        this.state.put(msg.k,msg);

        //step14/15/16 - if process that sent the
            message is not recognized by current
            process, add it to its members
        this.susp_level.put(msg.k,0);
        this.suspected_by.put(msg.k,new
            CopyOnWriteArrayList());
        this.timeout.put(msg.k,this.eta);
        this.timer.put(msg.k,create_timer(msg.k));
        this.members.add(msg.k);
        this.to_reset.add(msg.k);
  }

    //step18 - update own sups_level
    for(int l : msg.susp_level.keySet()){
      if(this.susp_level.get(l) != null){    this.
          susp_level.put(l,max(this.susp_level.get(
          l),msg.susp_level.get(l)));
        }
        this.susp_level.put(l,max(0,msg.susp_level.
            get(l)));
    }

    //step 19 - update own suspected_by with
        information it got from the message
    for(int l : msg.silent ){
      CopyOnWriteArrayList<Integer> aux = this.
          suspected_by.get(l);
        if(this.suspected_by.get(l) != null){
          aux.add(msg.k);
            this.suspected_by.put(l,aux);
        }
    }
    return 1;
}
```

B. Second protocol - Communication Efficient

```
public class message implements Serializable{
    public static final int HEARTBEAT = 1;
    public static final int STOP_LEADER = 2;
    public static final int SUSPICION = 3;
    public int tag_k;
    public int k;
    public int sl_k;
    public int silent_k;
    public int hbc_k;
}
```

Listing 1. Protocol 2 Message Structure

For **Task 1**, of the second protocol:

```
public void task1(){
  //step 02 - prepare HEARTBEAT message because he
      considers himself a leader
    if(!(this.next_period)){
      next_period=true;
    this.hbc+=1;
    }
    message msg = new message(message.HEARTBEAT,
        this.pid,this.susp_level.get(this.pid),0,
        this.hbc);

  //step 03 - broadcast state message
    csocket.broadcast(msg);
    lastsent = msg;
}
...
while(true)
{
  next_period = false;
    //step 01
    while(leader() == pid)
    {
      task1();
        Thread.sleep(eta);
    }

    //step 04

    if(next_period)
    {
      message msg = new message(message.STOP_LEADER
          ,pid,susp_level.get(pid),0,hbc);
      //step 04

      csocket.broadcast(msg);
    }
  Thread.sleep(1);
}
```

Timeout handling in this protocol is different from the first one, since it sends a *SUSPICION state*$_k$ message when it occurs and removes the timed out process from the *contenders*$_k$ group:

```
public int timerExpired(int j){
  //step 05 - increase timouts and send a SUSPICION
       state message
  timeout.put(j,timeout.get(j)+1);
    message msg = new message(message.SUSPICION,
         this.pid,this.susp_level.get(this.pid),j,0)
         ;
    csocket.broadcast(msg);
    lastsent = msg;

    //step 06 - remove timed out process from
         contenders
    contenders.remove((Object)j);
    return 1;
}
```

For **Task 2**, it implements the handling of message reception as follows:

```
//step 07 - check if process is known
if(!this.members.contains(msg.k)){
  //step 08/09/10 - allocate local variables for
       this process
    this.members.add(msg.k);
    this.susp_level.put(msg.k,0);
    this.last_stop_leader.put(msg.k,0);
    this.timeout.put(msg.k,eta);
}

//step 11 - update susp_level local variable
    according to state message received    this.
    susp_level.put(msg.k,max(this.susp_level.get(
    msg.k),msg.sl_k));

//step 12 - check if state message received is a
    HEARTBEAT and it is ordered
if(msg.tag_k == msg.HEARTBEAT && this.
    last_stop_leader.get(msg.k) <  msg.hbc_k){
  Timer t_aux = this.timer.get(msg.k);
    if(t_aux != null){
      t_aux.cancel();
      t_aux.purge();
    }
    // step 13 - reset timeouts and add process to
         contenders
    this.timer.put(msg.k,create_timer(msg.k));
    if(!contenders.contains(msg.k)) this.contenders
         .add(msg.k);
}

//step 14 - stop leader message was received,
    update last known last_stop_leader variable
else if(msg.tag_k == msg.STOP_LEADER  && this.
    last_stop_leader.get(msg.k) <  msg.hbc_k){
  this.last_stop_leader.put(msg.k,msg.hbc_k);
    Timer t_aux = this.timer.get(msg.k);
    //step 16 - remove timers
    if(t_aux != null){
      t_aux.cancel();
        t_aux.purge();
    }
    //step 15 - remove process that sent state
         message from contenders
    if(contenders.contains(msg.k)) contenders.
         remove((Object)msg.k);
}

//step 17 - if a SUSPICION message was sent and
    process belongs to silent group, increase its
    susp_level
else if(msg.tag_k == msg.SUSPICION && msg.silent_k
    == this.pid ){
  this.susp_level.put(this.pid,this.susp_level.get(
       this.pid)+1);
}
return 1;
```

III. Results & Analysis

In this section are presented results that are relevant to compare the two protocols explained before, with a brief discussion of the consequences associated to each result. Metrics used are based on article of a leader election service [2]. First thing that is measured is the delay (in ms) that takes for all processes to find a common leader, with respecting to increasing the number of processes. There are also specific variables that can be changed which are *t_unit* and η , which are important to define what is the periodicity on which a process can send messages. In order to simulate delay and losses, we have made some modifications to the Datagram socket, so that we can simulate with a more realistic environment. These parameters will be specified on the legend of each figure.

Additionally, while we did extensive tests with values for delay, loss prob and η, we did not include all of them due to space constraints. Here, only a few tests are shown, but the results were pretty much consistent with what expected regardless of testing.

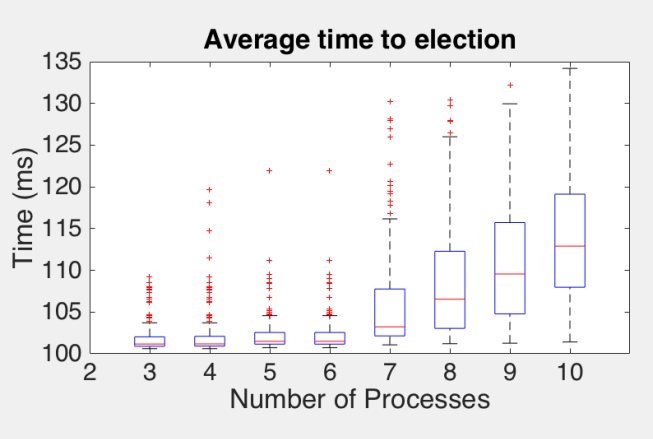

Fig. 1. Protocol 1 : Time to first election with no delay/losses, η =100ms

It is possible to see on figure 1 that, when simulating a scenario with no failures, the time required to elect a leader goes a little above the time required to send a message (100ms) and only begins to increase when number of processes reaches 7.

Comparing both protocols, it is possible to see that both will have an increase on the time to establish the leader, as the number of processes increases. This can be seen in Fig. 2. and Fig. 3. .

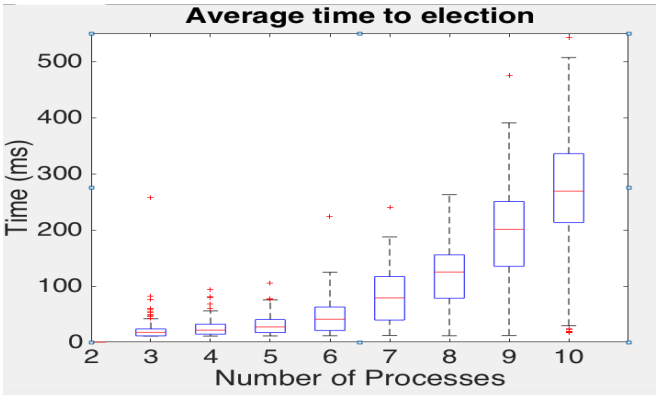

Fig. 2. Protocol 1 : Time to first election with 20% losses and random delay

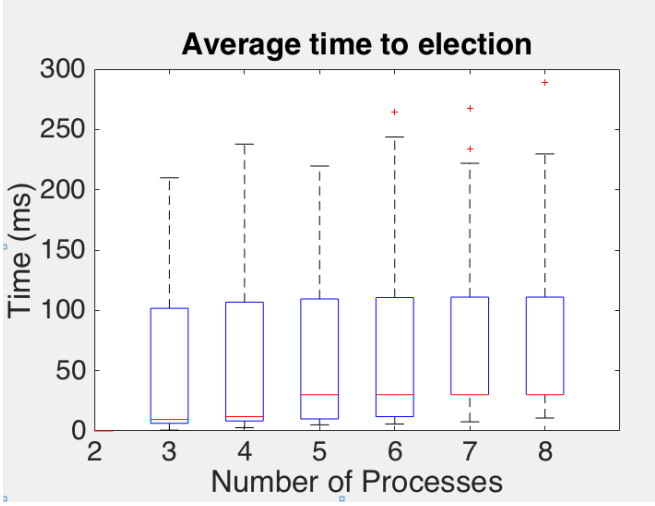

Fig. 3. Protocol 2 : Time to first election with with 20% losses and random delay

Another aspect that is considered to evaluate the efficiency of both protocols is the number of messages exchanged to establish a common leader. Again, we will be comparing this metric with the increasing number of nodes in the network

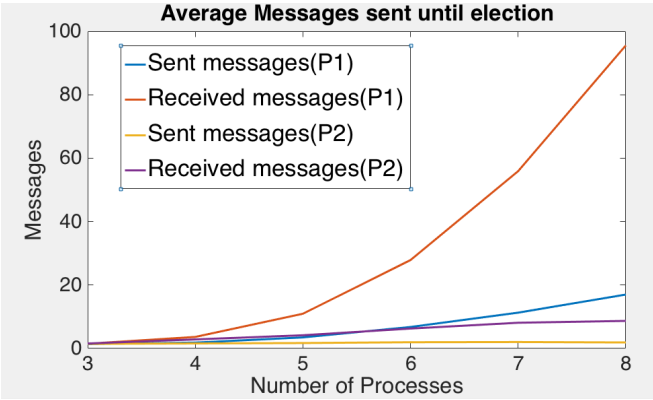

Fig. 4. Comparison between messages sent and received on P1 and P2

As expected, Protocol 2 (Communication efficient) shows the best behavior, requiring on average less messages to elect a leader. The y-scale contains messages per process,

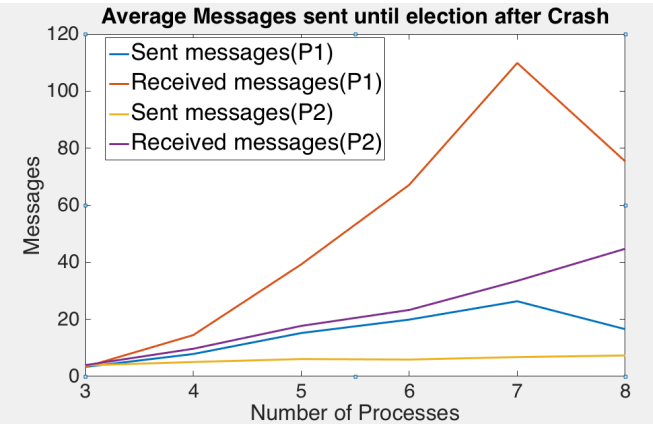

Fig. 5. Comparison between messages sent and received on P1 and P2 after leader crash

increasing with the number of processes. We present two cases: messages sent until first election and messages sent until second election (after leader crashing), which show a similar behaviour with a little difference in the number of messages.

IV. Conclusions

To sum up, it is clear that both protocols act in a different way, showing an interesting trade-off.

Protocol 2, while being faster and more efficient, requires more memory and more processing power per process.

Protocol 1, on the other hand, uses more bandwidth and is a bit slower. It can be used in an application that doesn't have a timing requisites on leader election to save CPU.

As a final note, this project gave us knowledge on socket programming and communication protocols, which will certainly be of great use in the future.

A. Work distribution

Afonso Reis - Protocol 1 and Protocol 2 implementation, graphic creation, timing and message measurements, report elaboration (sections 3 and 4). (35%)

David Ferrao - Presentation planning and elaboration, report elaboration (sections 1 and 2). (30%)

Vitor Fernandes - Protocol 1 and Protocol 2 implementation, demonstration planning, graphic creation, timing and message measurements, report elaboration (sections 3 and 4). (35%)

References

[1] Antonio Fernandez, Ernesto Jimenez, and Michel Raynal. Eventual leader election with weak assumptions on initial knowledge, communication reliability, and synchrony. In *Dependable Systems and Networks, 2006. DSN 2006. International Conference on*, pages 166–178. IEEE, 2006.

[2] Christof Fetzer and Flaviu Cristian. A highly available local leader election service. *IEEE Transactions on Software Engineering*, 25(5):603–618, 1999.

[3] Douglas Lea. *Concurrent programming in Java: design principles and patterns*. Addison-Wesley Professional, 2000.