Algorithms for Fair Leader Election Problem

Khaled Bachour, Wojciech Galuba, Ali Salehi

Abstract

The *consensus* problem in general and the *fair leader election* (FLE) problem in particular is one of the most important problem in distributed computing. In this paper we propose two algorithms for solving the FLE problem on an asynchronous shared memory system with their analysis. We tried to solve the problem with different requirements such as minimizing steps, and time complexities. Also we address the high-level definition of safety in a protocol, and we suggest an algorithm that satisfies this property.

1 Introduction

The problem of FLE is about $n \geq 1$ asynchronous processes p_i, where $1 \leq i \leq n$. Each process has a private memory of m bits. The processes can communicate only through reading and writing a shared memory of s bits. Computation proceeds in a sequence of atomic steps. In each step:

1. A scheduler chooses one of the processes.
2. The chosen process reads and modifies his private memory and the shared memory.

In addition to private memory m bits each process p_i has a read-only variable F_i and a write-only variable V_i. The variable $F_i \in \{1, \ldots, n\}$ represents the initial favorite of p_i for leader. The variable $V_i \in \{1, \ldots, n\}$ represents the final vote of p_i. In addition to the shared memory of s bits, there is a shared write-only bit T, which is initially set to 0. Setting T to 1 signals the completion of the election process.

The goal in the problem is to provide an algorithms to elect a leader by voting. The algorithm must satisfy the following conditions:

Termination T is set to 1 within a finite number of scheduler steps.

Agreement When $T = 1$, then $V_i = V_j$ for all $1 \leq i, j \leq n$.

Fairness When $T = 1$, then $|\{j|F_j = V_1\}| \geq |\{j|F_j = i\}|$ for all $1 \leq i \leq n$

We consider the following two types of schedulers. A scheduler is *fair* if at every point in the execution and for every process p_i, the is a finite number of steps until p_i is scheduled. A scheduler is *bounded fair* if at every point in execution and for every process p_i there is no more than $b \geq n$ steps until p_i is scheduled. The bound b is not assumed to be known to the processes.

Additionally we assume that each process can initialize its private memory (with the exception of F_i). The initial state of the shared memory is unknown (with the exception of T). Each process knows the number of processes n. No process can know the choices of the scheduler. In the reminder of the paper we present the *streaming* and *block message* algorithms with their analysis and comparison. We continue the paper by showing the minimal shared memory size required for solving the problem in addition to the time-optimal algorithm and then we provide our safety protocol definition. At the end we conclude with possible future works in this problem.

2 The streaming algorithm

2.1 Description

Let p_1 be the master process and let other processes be the slaves. The algorithm consists of two phases:

1. All slaves send their initial favorites F_i to the master.
2. After receiving initial favorites from all slaves, the master selects the winner of the leader election and communicates the selection to all slaves. Subsequently all slaves and the master set their V_i to the winner and the master sets T to 1.

In the above description we assume an existence of a message passing abstraction, that allows for sending data between the master and the slaves. We subsequently show how such abstraction can be implemented using shared memory.

2.2 Shared memory

The shared memory consists of a 2-bit transmission state XS and a 1-bit flag $Phase$. In the phase one of the algorithm this bit set to 0 and in phase two it is set to 1. XS can be in four possible states: *NTx*, *Tx0*, *Tx1* and *Ack*. These four states allow for transmitting arbitrary binary streams from a sender process to a receiver process in the following way.

Let S be the sender process and R be the receiver.

1. Let XS initially be set to *NTx*, which means "no transmission".
2. If XS is *NTx* or *Ack*, S sends bit 0 by setting the state to *Tx0* or sends bit 1 by setting the state to *Tx1*.
3. R confirms the reception of the bit by setting XS to *Ack* if the transmission should continue or back to *NTx* if the transmission should end.

S can only send when XS is *NTx* or *Ack*, otherwise it waits. Symmetrically, R will wait for either *Tx0* or *Tx1*. R can decide to terminate the transmission in the third step based on a termination character sent in the bit stream or when the agreed upon number of bits has been sent.

Note that in the case of one receiver and one sender there is no semantic difference between the *NTx* and *Ack* states. However, in the case of multiple senders using the same XS shared register for maintaining the transmission state, the access to the shared communication channel must be controlled. Mutual exclusion can be achieved by allowing a sender to send its first bit in the stream only when XS is set to *NTx*. Otherwise if the sender is about to send its first bit and the sender sees *Ack* then it means that another sender is currently using the channel. After sending the first bit, the sender raises a private *channelLocked* flag to signify that it is the current exclusive owner of the channel. The flag remains raised until the end of the transmission. While *channelLocked* is raised, the subsequent bits in the stream are sent upon seeing *Ack*.

2.3 Details of the algorithm

When the master (Algorithm 1) is scheduled for the first time it sets XS to *NTx* to initialize the channel and sets $Phase$ to 0. Each slave (Algorithm 2) before initiating the transfer performs a handshake with the master as follows:

1. Slave waits for the *NTx*, then it sets XS to *Ack*
2. Master upon receiving *Ack* sets XS back to *NTx*
3. Slave receives *NTx* from master and commences transmission

This handshake protocol ensures that no slave sends any bit of data before the master has been initialized. Otherwise, the bit sent by the slave would be lost after the master sets XS to *NTx* while initializing itself.

There can be at most n unique initial favorites, therefore they can be encoded in $\lceil \log_2 n \rceil$ bits. This encoding is used by slaves to send the initial favorites to the master in the first phase and by the master to send the $winner$ back to the slaves in the second phase. The length of one message is constant and therefore both the receiver and the sender know when the transmission ends.

The slaves use the *channelLocked* variable to control access to the shared communication channel (XS). There is at most one slave that has *channelLocked* set to $true$ at any given time. The *channelLocked* is set to $true$ whenever a slave has some data to send and sees $XS =$*NTx* and it is set back to $false$ after the transmission finishes. During the transmission the XS variable will never be set to *NTx* therefore no other slave can acquire the channel.

The master maintains the $slaveCount$. This variable tracks the number of slaves which have sent their initial favorites in the first phase. In the second phase $slaveCount$ holds the number of slaves that have already successfully received the winner of the election. Both phases end when $slaveCount$ reaches $n - 1$.

Index i of the $votes$ array maintained by the master stores the number of votes for candidate i. At the end of the first phase the master selects the $winner$ by calling $selectWinner$ on the $votes$ array. This function picks the candidate with the highest number of votes.

At the end of the second phase when all slaves have received the $winner$, the master sets its V_i variable to the winner and signals the termination of the algorithm by setting $T := 1$

Algorithm 1 Streaming algorithm - master

```
if process not initialized then
    $Phase := currentVote := slaveCount := 0$
    $k := 1$
    $XS := NTx$
    $winner := null$
    for i:=1 to n do
        $votes[i] := 0$
    end for
end if
if $Phase = 0$ then
    if $k = 0 \wedge XS = Ack$ then
        $XS := NTx$
    else if $XS = Tx0 \vee XS = Tx1$ then
        if $XS = Tx0$ then
            set $k$-th bit of $currentVote$ to 0
        else
            set $k$-th bit of $currentVote$ to 1
        end if
        $k := k + 1$
        if $k > \lceil \log_2 n \rceil$ then
            $votes[currentVote] := votes[currentVote] + 1$
            $XS := NTx$
            $slaveCount := slaveCount + 1$
            if $slaveCount = n - 1$ then
                $votes[1] := F_1$
                $winner := selectWinner(votes)$
                $k := 0$
                $Phase := 1$
            end if
        end if
    end if
else if $Phase = 1 \wedge XS = Ack$ then
    if $k > \lceil \log_2 n \rceil$ then
        $k := 0$
        $slaveCount := slaveCount + 1$
        if $slaveCount = n - 1$ then
            $V_1 := winner$
            $T := 1$
        end if
    end if
    if $T = 0$ then
        if $k$-th bit of $winner$ is 0 then
            $XS = Tx0$
        else
            $XS = Tx1$
        end if
        $k := k + 1$
    end if
end if
```

Algorithm 2 Streaming algorithm - slave

```
if process not initialized then
  $channelLocked := voted := voteSent := false$
  $pingSentToMaster := pongRecievedFromMaster := false$
  $k := 1$
  $winner := 0$
end if
if $Phase = 0$ then
  if $XS = NTx \wedge \neg pingSentToMaster$ then
    $XS := Ack$
    $pingSentToMaster := true$
  else if $XS = NTx \wedge pingSentToMaster \wedge \neg pongRecievedFromMaster$
  then
    $pongRecievedFromMaster := true$
  end if
end if
if $pongRecievedFromMaster \wedge Phase = 0 \wedge \neg voteSent$ then
  if $XS = NTx \wedge \neg channelLocked$ then
    $channelLocked := true$
  end if
  if $(XS = NTx \vee XS = Ack) \wedge channelLocked$ then
    if $k$-th bit of $F_i$ is 0 then
      $XS := Tx0$
    else
      $XS := Tx1$
    end if
    $k := k + 1$
    if $k > \lceil \log_2 n \rceil$ then
      $voteSent = true$
      $channelLocked = false$
    end if
  end if
else if $pongRecievedFromMaster \wedge Phase = 1 \wedge \neg voted$ then
  if $XS = NTx$ then
    $XS := Ack$
    $k := 1$
    $channelLocked := true$
  else if $(XS = Tx0 \vee XS = Tx1) \wedge channelLocked$ then
    if $XS = Tx0$ then
      set $k$-th bit of $winner$ to 0
    else
      set $k$-th bit of $winner$ to 1
    end if
    $k := k + 1$
    if $k > \lceil \log_2 n \rceil$ then
      $V_i := winner$
      $voted := true$
      $XS := NTx$
    else
      $XS := Ack$
    end if
  end if
end if
```

3 Proof of correctness

We show the correctness of the algorithm by proving the three properties defined in Section 1: *Termination*, *Agreement* and *Fairness*.

Proof. Termination. The algorithm terminates when the shared variable T is set to 1. We need to show that this event occurs within a finite number of scheduler steps from the start of the algorithm. T is set to 1 by the master in the second phase after it receives the acknowledgement of the last bit of $winner$ from the $(n-1)$th slave. There are finitely many bits sent from the master to the slave and there are finitely many slaves that receive the $winner$. Sending a single bit consists of two steps of the transmission protocol: the sender sets XS to either *Tx0* or *Tx1* and the receiver then responds with an *Ack*. The scheduler is fair, which guarantees that the receiver is scheduled within the finite number of scheduler steps after the sender, which in turn guarantees that a single bit will be sent in a finite number of scheduler steps. Both the slave and the master keep track of the number of transmitted bits in the variable k, once the variable reaches a finite value the transmission ends. No slave in the second phase receives $winner$ more than two times. After receiving the $winner$ it immediately votes and sets *voted* to true, which prevents the same slave from locking the channel again. The master keeps track of the number slaves it had sent the $winner$ to in $slaveCount$ and can terminate phase two when $slaveCount$ reaches a finite value. Hence, there are finitely many transmissions before T is set to 1. In each transmission there are finitely many bits sent. Each bit requires finitely many scheduler steps, therefore there are finitely many scheduler steps in phase two before the algorithm terminates.

By analogy, there are also finitely many transmissions in phase one. In each transmission there are finitely many bits sent with an additional handshake which takes two transmission steps and hence a finite number of scheduler steps. Therefore, both phases of the algorithm take a finite number of scheduler steps before T is set to 1.

Agreement: In the second phase of the algorithm each slave receives the same value of $winner$. Each slave then uses this value for setting its own V_i variable. The master uses the same $winner$ value for setting V_1. The master sets T to 1 only after it has received the acknowledgement of the last bit of $winner$ from the last slave. But once the slave receives the last bit of $winner$ it immediately votes. Therefore, when the acknowledgement of the last bit is received by the master, we can be sure that all slaves have already voted. Moreover, they have voted for the same $winner$ candidate, which ensures *Agreement*.

Fairness follows from *Agreement* and the fact that the master uses the $selectWinner$ function to select the candidate with the highest number of votes. That chosen candidate is then transmitted to all slaves as $winner$. □

3.1 Time complexity

For a bounded fair scheduler we find the upper bound on the number of steps t_b until the algorithm terminates. In the first phase there are $n-1$ transmissions from the slaves to the master. Each transmission involves the initial handshake and sending of $\lceil \log_2 n \rceil$ bits. In the worst case sending k bits can take $(k+1)b-1$ steps. First the sender puts the first bit on the channel then the receiver within the next $b-1$ steps receives it and acknowledges. Then the consecutive bits are sent until the last one. It is possible that the last scheduling of the receiver is delayed for the next $b-1$ steps in the case when has been scheduled in the step just before the sender sent the last bit, therefore there are $kb+b-1$ steps in total. The initial handshake adds another kb steps: the sender puts Ack on the channel until the receiver responds with Ntx. However, handshakes are only used in the first phase of the algorithm.

The $n-1$ transmissions of the first phase amount to $((\lceil \log_2 n \rceil + 2)b-1)(n-1)$ steps. And the $n-1$ transmissions without handshakes in the second phase sum up to $((\lceil \log_2 n \rceil + 1)b-1)(n-1)$. In total there can be a maximum of $t_b = ((2\lceil \log_2 n \rceil + 3)b-2)(n-1)$ steps until T is set to 0. Which gives the time complexity of $O(bn \log n)$. The value of t_b for $b = n = 115$ is 222642.

4 Block Message Algorithm

The given streaming algorithm requires information to be sent in streams of bits. But for each bit to be sent, b steps are needed (in case of a bounded scheduler). We can improve the time complexity by increasing the shared memory to $1 + \lceil \log_2(n+2) \rceil$ bits. We now use the $\lceil \log_2(n+2) \rceil$ bits to denote the initial favorite id, in addition to the two possible states NTx and Ack. The state Ack is no longer needed for message transmission, since the the message is now sent as a whole, but it is still needed for initialization. The additional bit needed to indicate the end of the first phase is retained.

This algorithm works in the same way as the streaming algorithm except the messages are sent in constant time rather than $O(log_2 n)$ time.

Correctness : By the same reasoning used for the streaming algorithm, that the block message algorithm also satisfies the three properties of correctness.

Time Complexity : During the first phase, a single vote requires 4 messages (3 for handshake, 1 for initial favorite id). This needs to be done $n-1$ times. The second phase involves sending the id of the winner to the $n-1$ slaves. Each message requires only 2 steps: Master sets winner id, one of the slaves reads it and sets memory to NTx. So total number of steps required is $6(n-1)$. For the worst case in a bounded scheduler with bound b, the number of steps becomes $6b(n-1)$, or $O(bn)$.

5 Comparing the algorithms

The two algorithms are structurally equivalent and share the same asymptotic bounds for total cost $O(bn \log n)$ but they differ in shared memory space and in time complexity.

6 Minimal shared memory size

In this section we present the modified stream algorithm that requires at most 2-bits of shared memory (compared to the original version with 3-bits)

Table 1: Time and space complexity of the proposed algorithms

Algorithm	Streaming	Block
s	3	$1 + \lceil \log_2(n+2) \rceil$
t_b	$((2\lceil \log_2 n \rceil + 3)b - 2)(n-1)$	$6b(n-1)$
$t_b\ O()$	$O(bn \log n)$	$O(bn)$
$t_b(115)$	222642	78660
$s * t_b$	$3((2\lceil \log_2 n \rceil + 3)b - 2)(n-1)$	$6b(n-1)(1 + \lceil \log_2(n+2) \rceil)$
$s * t_b\ O()$	$O(bn \log n)$	$O(bn \log n$
$s * t_b(115)$	667926	629280

6.1 Modified streaming algorithm

We use the XS variable as before (section 2) with the same set of values (*NTx*, *Tx0*, *Tx1* and *Ack*). However, we drop the *Phase* bit from the shared memory. Instead of using the *Phase* bit to signal the transition to the second phase, the master sets XS to *NTx* at the end of the first phase. This state cannot occur anytime during the first phase and slaves no longer exclusively acquire the shared communication channel, instead they all transmit their initial favorites concurrently.

The way the initial favorites are encoded is changed. Assume a slave wants to send the initial favorite i. The bit stream consists of $2^{i\lceil log_2 n \rceil + 1}$ ones and a terminating zero at the end. The master computes the sum of all ones that it receives as *sumOnes*. If the number of received zeroes reaches $n-1$ the master signals the end of the first phase by setting XS to *NTx* and the second phase proceeds identically as in the original streaming algorithm.

Given the *sumOnes* variable master computes the *votes* array by executing *computeVotes*(*sumOnes*) (Algorithm 3) and then selects the *winner* by calling *selectWinner*(*votes*) as in the original streaming algorithm.

Algorithm 3 The *computeVotes* function

```
computeVotes(integer $sumOnes$)
**if** $sumOnes$ is odd **then**
  $sumOnes := sumOnes + 1$
**end if**
**for** $i := 1$ to $n$ **do**
  $votes[i] := (sumOnes$ **div** $2^{i\lceil log_2 n \rceil + 1})$ **mod** $n$
**end for**
```

There is no handshake protocol in use in the first phase of the algorithm. We can loose at most one bit when a slave starts sending before the master is initialized. This is compensated for by multiplying the number by 2 before sending (the +1 factor in the expression $2^{i\lceil log_2 n \rceil + 1}$) and later on if *sumOnes* is odd then we add the missing bit. The algorithm has a prohibitively high time complexity on a bounded fair scheduler: $O(bn^2 2^n)$.

6.2 One-bit impossibility

The modified streaming algorithm solves the FLE problem with 2 bits of shared memory. In this section we show, it is impossible to solve the problem with with 1-bit of shared memory by providing the following theorem.

Theorem 1 (One-bit impossibility). *In an asynchronous shared memory system it is impossible for one process to communicate a single bit of information when one bit of shared memory available.*

Proof. Consider a process S that holds a bit q in its private memory. S wishes to communicate the value of that bit to process R. There is one bit t in the shared memory. The initial state of t is unknown to both R and S. Let *execution* be the sequence of system states including the private and shared memory. Subsequent elements in the execution correspond to subsequent scheduler steps. When a process is scheduled there are only two possible actions it can perform on the shared memory, it can either flip t or leave it unchanged. If S decides to leave t unchanged during a scheduling step then if R is subsequently scheduled it cannot distinguish between an execution in which S was scheduled and an execution in which it was not. Therefore S can perform only one action on shared memory, flipping t. However, q can have two possible values and at some point of the protocol two different actions on the shared memory must be performed so that R can distinguish between two possible executions in which q has different values. But given only one possible action on the shared memory it is impossible. $\square$

7 Time-Optimal solution

In this section we show an algorithm for achieving the lowest bound $3b$ for solving the problem. Assuming a two dimensional array $A[N \times N]$ of $\lceil \log_2(n+1) \rceil$ bits of of shared memory. As well as a shared array *decide*$[N]$ of bits. Each process p_i will collect votes on column i and write its vote on row i.

At the beginning all bits are uninitialized. So each process p_i, upon first being scheduled, sets *decide*$[i]$ bit to 0, sets all the values on column i to $\perp$, and sets every entry on row i to F_i. After initialization, p_i enters a normal cycle and for each time it is scheduled, it resets every entry on row i to F_i if any of these values were modified. It then updates its column from columns of other processes whose row does not contain a $\perp$. When a process finds its entire column has been filled with votes, it computes the winner and sets its *decide* bit to 1. The first process to observe all 1 values in the *decide* array sets T to 1.

An upper bound for the worst-case running time for this algorithm is $3b+3$ which includes $b+1$ steps for initialization in worst case and $b+1$ for voting. And in the worst case $b+1$ for deciding.

We surmise that the real worst-case running time for this algorithm is $3b-n+2$. We came to this conjecture by applying several known worst-case scenarios for other algorithms. None of these scenarios led to a running time greater than what proposed.

Algorithm 4 Time-Optimal Solution - Algorithm for Process i

```
if $x = 0$ then
  for $j = 1$ to $n$ do
    $A[i, j] \leftarrow \perp$
  end for
  $decide[i] \leftarrow= FALSE$
  $x \leftarrow 1$
end if
for $j = 1$ to $n$ do
  $A[j, i] \leftarrow F_i$
  if $A[i, j] \neq \perp$ then
    copy all missing votes from column $j$ to column $i$
  end if
end for
if $\forall j : A[i, j] \neq \perp$ then
  if $decide[i] \neq TRUE$ then
    $decide[i] \leftarrow TRUE$;
    select as winner, the process with highest number of votes
    (in case of tie, choose smaller id).
  end if
end if
if $\forall j : decide[j] = TRUE$ then
  $T \leftarrow 1$
end if
```

8 Protocol safety

In this section we describe a high-level definition of safety requirements for a protocol to guarantee a fair election process. A protocol is *safe* if no process can modify its original preferred leader without being detected if the decision to make this modification was based on information acquired during the election process.

8.1 A safe protocol

We present the following protocol:

- Assume an encryption algorithm $E : \{0, ..., n-1\} \times \{0, ..., 2^k - 1\} \to \{0, ..., 2^{bc}\}$.
- The Shared memory contains $x + n(bc + k)$ bits. Each process will write on $(bc + k)$ bits. The x bits will be left for “administrative” issues such as initialization and switching between phases.
- Each process i will randomly generate it's own secret key K_i of predetermined length k. It will compute $E(F_i, K_i)$ and will write that value on the bc bits designated to it. After all processes have written their encrypted vote, they write their respective keys on the remaining k designated bits. Finally, when all keys have been written, all information needed to produce the result is available and a winner can be found.

This protocol satisfies the high-level safety requirement defined above and no process can acquire any knowledge until it gives enough information about it's original vote (the encrypted vote) so as to make any later change detectable. For this argument to be valid, we make two assumptions:

1. Given $E(F_i, K_i)$ it is impossible to compute F_i without knowledge of K_i. It is also impossible to guess K_i.
2. Given a key K_i and $E(F_i, K_i)$, it is impossible to compute K_i' and F_i' such that $E(F_i, K_i) = E(F_i', K_i')$.

These two assumptions are valid for any statistically secure encryption algorithm. The former prevents a process from knowing any information before writing its encrypted vote $E(F_i, K_i)$. The latter prevents a process from changing its initial preference after it has written its encrypted vote without being detected. Hence the protocol prevents a process from changing its initial preference after it has acquired information about other processes' initial preferences.

9 Future work

9.1 Minimizing the shared memory usage

We have shown that it is impossible to solve FLE with one bit shared memory. However it is possible to solve the problem using two bits. The one-bit system can have at most two possible states of shared memory and at most two possible state transitions. For a two bit system there are 4 states possible and 12 state transitions. A question one may ask is: what is the minimal number of states and state transitions required to solve the problem?

By showing one-bit impossibility (section 6.2 we were relying on the fact that the bound b of the bounded fair scheduler is not known to any of the processes. It remains an open question, however, whether if this bound was known to the processes it would still be impossible to solve the problem by using only one bit of shared memory.

9.2 The space-time tradeoff

We have presented two algorithms that have a total cost of $O(bn \log n)$, and we could not find an algorithm with lower cost. However, we were not able to prove that the total cost of any algorithm is $\Omega(bn \log n)$ in order to show that our algorithms are optimal.

9.3 Protocol security

Protocol security needs to be presented as formal requirements rather than an abstract description. There are also many more ways in which a process can "cheat" that are not encompassed by our description.

10 Conclusions

In this paper, we show that the FLE problem can solved in two bits of shared memory and that it cannot be solved in one. We present two algorithms that solve the problem in $O(bn \log n)$, which we claim might be optimal. We also present an algorithm and conjecture that it achieves optimal time-complexity of $3b - n + 2$. Finally we describe a high-level definition of safety property in a protocol, and we suggest an algorithm that satisfies this property.