April 19, 2005

Consensus with Unitialized Shared Memory

Mini-project report for the Problem Solving in Computer Science course

by

Eda Baykan

Abhishek Garg

Alex Şuşu

Abstract

In this paper we present several distributed algorithms for the consensus problem. We propose different approaches to solve this problem with minimum shared memory, with minimum run time and with an optimal memory VS time tradeoff cost. In addition to this, we give a taxonomy of cheating types for this problem and propose two algorithms that are resilient to specific types of cheating.

1 Introduction

A consensus algorithm is one in which processes agree on the same value from the values initially proposed independently by each one of them.

Before the consensus algorithm starts, the processes are assigned IDs, from 0 to n-1 - where n is the number of processes involved in the algorithm.

The consensus problem consists of two phases. In the first phase each process proposes a value - in our particular setting of the problem, his favorite among the others. In the second phase, each process decides with the same deterministic function what is the value they agree on - in our particular case, we determine by using majority voting (in the case several proceses get the majority number of votes, we choose the one with the smallest ID), who is the winner of the elections. When all processes decide, they reach consensus.

Processes act in an asynchronous way and communicate through shared memory which is uninitialized, excluding a bit that indicates whether the leader is chosen or not.

Processes get access to the shared memory when the scheduler gives turn to them. In the algorithms we propose, we assume a bounded scheduler which does not ignore each process more than b steps, where b is a given natural number.

We focus our work on four main problems:

- Minimizing the number of bits in the shared memory
- Minimizing the number of steps until termination
- Optimizing the total cost
- Designing a cheat resilient algorithm

The body of this paper is organized as follows. In section 2, we give algorithms for minimum shared space. In section 3, we propose an algorithm that solves the leader selection problem with minimum number of steps. In section 4, we present a solution that has the optimal cost. Furthermore, we prove that our optimal cost algorithm eventually terminates and is safe. Finally in section 5, we introduce the notion of cheating and we show that it is not possible to design an algorithm that is resilient to all kinds of cheating. At the end of section 4, we propose an algorithm that is cheat resilient to restricted types of cheating.

2 The Algorithm with Minimum Shared Memory

In the beginning of this section we propose a 3 bit shared memory algorithm for solving the consensus problem. Then we build on this a 2 bit consensus protocol. The algorithm requires $t(b) = 2n^2b\lceil log_2(n)\rceil + nb$ steps (in the worst case).

2.1 The 3 Bit Algorithm

The basic idea of the algorithm is that we want to transform our asynchronous problem into a synchronous one. We attain this by using the *barrier paradigm* (a barrier is an IPC "artifact" that allows a group of asynchronous tasks to synchronize by waiting all the processes to arrive at the barrier before proceeding further) with the prisoner consensus [1]. We assume a specialization of the processes: one process is the "leader" and all the others are "followers". This specialization is agreed before the consensus protocol starts - process P_0 is the leader, P_1, P_2, ..., P_{n-1} are followers). The algorithm runs in the following phases:

- **Phase 1:** In the first at most (2*n-1)*b steps - prisoner consensus with uninitialized SM takes at most (2*n-1)*b steps - all processes read the Announce bit and process P_1 writes in Data the least significant bit of his proposal.
- **Phase 2:** In the next n*b steps - prisoner consensus with initialized SM takes at most n*b steps - the leader process flips the Announce bit and, within this round every process reads from bit Data the least significant bit of the proposal of process P_1
- **Phase 3:** In the next n*b steps, the leader process flips the Announce bit and the process P_1 writes in Data the second least significant bit of his preference
- **Phase 4:** in the next n*b steps, the leader process flips the Announce bit and, within this round every process reads from bit Data the second least significant bit of process P_1
- **Phase 5 to Phase** $n * \lceil log_2(n)\rceil * 2 - 1$ **:** We repeat Phase 4. The only thing different in all these phases, is the actual bit that is sent (and by whom it is sent).
- **Phase** $n * \lceil log_2(n)\rceil * 2$ In the next n*b steps, the leader process flips the Announce bit and, within this round every process reads the most significant bit of the last process, and decides on the result of the election
- **Phase** $n * \lceil log_2(n)\rceil * 2 + 1$**:** In the next n*b steps, the leader process flips the Announce bit and, sets T=1. OBS: in this last b-round all followers must not do anything

The pseudocode of the leader and followers is presented in 1,2. In the leader and follower pseudocode, we are writing "read" or "write" whenever we are accessing the SM.

Algorithm 1 Algorithm for Consensus with Uninitialized Memory - Follower Routine

```
1: ProcessFollower()
2: read AnnounceNewI := Announce;
3: read SwI := Sw;
4: if vi = 0 and SwI = 0 then
5:    v_i := v_i + 1;
6:    AnnounceOldI := AnnounceNewI;
7:    write Sw := 1;
8:    if PID==1 then
9:       write Data:=LSB(F_i)
10:   end if
11: else if v_i = 1 and SwI = 0 and AnnounceOldI = AnnounceNewI then
12:   v_i := v_i + 1
13:   write Sw := 1;
14: else if (vi = 1 or v_i = 2) and SwI = 0 and AnnounceNewI != AnnounceOldI
    then
15:   write Sw := 1;
16:   AnnounceOldI := AnnounceNewI;
17:   read LSB(pref[1]) := Data;
18: else if v_i = 3 and SwI = 0 and AnnounceNewI != AnnounceOldI then
19:   v_i := v_i + 1;
20:   write Sw := 1;
21:   AnnounceOldI := AnnounceNewI;
22: else if v_i = 4 and SwI = 0 then
23:   v_i := v_i + 1;
24:   write Sw := 1;
25:   if PID = 1 then
26:      write Data:= SecondLSB(Fi)
27:   end if
28: else if v_i = 5 and SwI = 0 and AnnounceNewI != AnnounceOldI then
29:   {"discharge" round}
30:   vi := vi + 1;
31:   write Sw := 1;
32:   AnnounceOldI := AnnounceNewI;
33: else if vi = 6 and SwI = 0 then
34:   {"charge" round}
35:   vi := vi + 1;
36:   write Sw := 1;
37:   AnnounceOldI := AnnounceNewI;
38:   read SecondLSB(pref[1]) := Data;
39:   {After this we repeat the cases for v_i = 4, 5, 6 until v_i = 2*n*⌈log_2(n)⌉+1}
40: else if  v_i = 2 * n * ⌈log_2(n)⌉ + 1 then
41:   V_i = Majority();
42: end if
```

Algorithm 2 Algorithm for Consensus with Uninitialized Memory - Leader Routine

```
 1:
 2: ProcessLeader()
 3: loop
 4:    read AnnounceNewI := Announce;
 5:    read SwI := Sw;
 6:    if SwI = 1 then
 7:       write Sw := 0;
 8:       c := c + 1;
 9:       if (c = 2*(n-1) and state = 0) or (c=n-1 and state != 0) then
10:          AnnounceNewI := ! AnnounceNewI;
11:          write Announce := AnnounceNewI;
12:          if state=0 then
13:             read LSB(pref[1]) := Data;
14:             state := state + 1;
15:          else if state=1 then
16:             read SecondLSB(pref[1]) := Data;
17:             state := state + 1;
18:          else if  then
19:             ...
20:          else if state = n * ⌈log2(n)⌉ then
21:             {The leader starts sending his proposal}
22:             write Data := LSB(pref[0]);
23:             state := state + 1;
24:          else if state=n * ⌈log2(n)⌉ + 1 then
25:             Vi := Majority();
26:             T:=1;
27:          end if
28:       end if
29:    end if
30: end loop
```

2.2 2 Bit Algorithm

First, we tried to adapt the "channel" paradigm presented by Wojtek during one of the lectures. In Appendix A we present our results. Following, we present a sketch of a 2 bit solution, built on the previous 3 bit algorithm. Basically this solution transmits the data through the Announce bit. Also, the solution is centralized - the leader (and not the followers) gathers the proposals of all processes with the algorithm presented below. The problem is that it is less obvious how to make every follower to get the proposals as well. To perform this we do the following:

- in the first phase, the leader is process P_0 and he collects all proposals - we use the Announce bit to signal another barrier in order to go to the next phase
- in the second phase, the leader is process P_1 and he collects all proposals ...
- in the nth phase, the leader is process P_{n-1} and he collects all proposals. At the end he sets T=1, since all other processes have been leaders and know all proposals s.t. they can vote.

SubAlgorithm LeaderCollectsAllProposals
The process that has to send the data does the following:

- when his vi is 0 (and Sw=0) it sets the Announce bit to the value of the data it has to send and makes, as in the normal prisoner consensus, makes Sw=1
- If Announce has:
  - the same value, then the leader will see at the end of this current barrier that nobody has changed its value so he knows that the current process that has to transmit data transmits a data bit equal to the current value of Announce.
  - the negated value, then the leader will see at the end of this current barrier that somebody has changed its value (he keeps in his private memory the current value of the Announce) so he knows that the current process that has to transmit data transmits a data bit equal to the current value of Announce. In this case it will be a problem with the other processes that might interpret wrong the fact the flipping of Announce. We keep in mind that Sw is 1. We have two cases:
    * either the process has vi=0 (and Sw is 1), (so he knows that he has not made Sw=1 once during this barrier so the leader could not count him), so this has to be a violation of the prisoner consensus, so it can be only the fact that another process wants to transmit as data the current value of Announce. Therefore he will not do anything.
    * either the process has vi=1, (so he has made already Sw=1 once during this barrier and he is actually waiting that the Announce bit flips), so he will do the following: he will set vi=2 (we can

say vi=0, but we say vi=2 to distinguish between the cases), and he will not write anything into Sw, since Sw is already 1.

* while in vi=2, whenever he will be invoked he will either see
  - Sw=1 so he cannot do a thing,
  - either he will see Sw=0 (in which case we know the leader was invoked BEFORE, and then the leader settled the Announce bit to the right value). In this case he knows that he has not voted for the current barrier, yet the Announce bit was flipped. So he knows that actually the previous Announce bit flip was made for transmitting data not for announcing. So he will change his state to the previous state (he will set vi=0) and he will not do anything else.

3 The Minimal Time Algorithm

In this section we present the algorithm which has $O(b)$ run time and $O(n^2 logn)$ space complexity where n is the number of processes and b is the bound of scheduler.We specify space complexity in terms of shared memory. Furthermore, each process has $O(n^2 logn)$ private memory.

3.1 High-level description of our solution

Our approach is to write the schedule of scheduler,ScheduleSnapshot(SS),into shared memory in order to minimize run time. When the scheduler gives turn to a process, the process writes its process id at the end of ScheduleSnapshot if the conditions Writing into SS are satisfied. Processes learn the end of SS by reading SnapshotIndex from the SharedMemory. Each process keeps track of the ScheduleSnapshot since it is first activated by the scheduler. With this approach,processes handle uninitialized memory problem since they remember the memory locations they have started writing. In other words, this algorithm does not spend additional time on initializing shared memory. A process understands whether other processes proposed or decided by looking at the ScheduleSnapshot from the index it is first activated to the end of the it. Since one of the processes has to make T= 1, processes must understand whether the rest has voted or not.

In this algorithm when a process votes, it writes $\overline{P_{id}}$ at the end of SS if the conditions for Writing into SS are satisfied. Each entry in SS has $\lceil log_2(n) \rceil + 1$ bits where $\lceil log_2(n) \rceil$ bits are used to convey process id information and 1 bit is used to specify whether process has voted or not. We name a process that has voted as $\overline{P_{id}}$ or a process with hat interchangeably.

The steps of the algorithm can be summarized as follows. If the scheduler gives turn to a process for the first time, process writes P_{id} at the end of SS and writes its favorite into F[n] array into Shared Memory. This array holds the favorite of each process. If the process is given turn again, it remembers the memory location where it has written when it is given turn for the first time.We name this memory location as myLifeStartedAt $_{id}$. Process reads entries from ScheduleSnapsot from myLifeStartedAt $_{id}$ to the end of the SS and stores these values into MyHistory $_{id}$.Process decides writing into ScheduleSnapsot its pro-

cess id by looking at MyHistory $_{id}$ and considering the conditions for writing into SS.

As we show in Section *Space Analysis*, ScheduleSnapsot has a predetermined size,which we name as QueueSize. In order to use memory efficiently, we have designed ScheduleSnapshot as circular. If the process decides to write, it updates SnapShotIndex with SnapShotIndex++ MOD QueueSize. When a process analyzes its history and sees no process with hat, it searches whether it can see all processes except itself. If it sees, this means every process has announced its favorite. Then the process votes. The process that votes for the first time, initializes the counter named as TrustLifeStartedAt to its process id. With this strategy other processes who sees at least one process with hat in their history, can read this counter and may have a larger history. Each process that votes, updates TrustLifeStartedAt with their MyLifeStartedAt$_{id}$ if they can have a larger history. The process who sees all processes except itself with hat, can announce the end of consensus. In Algorithm 3, we give the pseudo-code for the algorithm minimum number of steps.

3.2 Conditions for Writing into ScheduleSnapshot(SS)

In order to use memory efficiently, we urge following conditions on the processes while writing to ScheduleSnapshot(SS) in Shared Memory.

- If activated for the first time, Write P$_{id}$ at the end of SS
- If Voting has not been completed
  - If at the end of SS there is P$_{id}$ or $\overline{P_{id}}$, DO NOT Write
  - If all processes in MyHistory$_{id}$ has seen P$_{id}$, DO NOT Write
  - If there is at least one process who has not seen P$_{id}$ before, Write

3.3 Example Iteration

In this section we give an example iteration of our algorithm. Lets take assume that there are 3 processes and b is 8. Since there are 3 processes, we need 7 entries in ScheduleSnapshot. We describe how we calculate the size of in the next section. Before the scheduler starts
SnapshotScedule is: @@@@@@@
The entries are indexed from 0 to 6
SnapshotIndex(SI) = 3
The schedule is given as below
$P_1P_2P_1P_1P_2P_1P_3P_1P_2P_3$

- When the scheduler gives turn to P_1 , it reads SI and writes its process id at 3. It then updates SI to 4 and MyLifeStartedAt$_1$ to 3.
  - @@@P_1@@@
- When the scheduler gives turn to P_2 , it reads SI and writes its process id at 4. It then updates SI to 5 and MyLifeStartedAt$_1$ to 4.
  - @@@P_1P_2@@

- When the scheduler gives turn to P_1 again,it reads ScheduleSnapshot vector from MyLifeStartedAt$_1$ to SI-1(i.e from 3 to 4). It discovers that P_2 has not seen P_1, so it writes its process id at index 5. Finally it updates SI to 6.
  - $\boxed{@@@P_1P_2P_1@}$
- When the scheduler gives turn to P_1 ,it reads ScheduleSnapshot vector from MyLifeStartedAt$_1$ to SI-1. It sees that the last entry in SS is itself, so it does not write into SS(to avoid duplication)
- When the scheduler gives turn to P_2 , it reads ScheduleSnapshot from MyLifeStartedAt$_2$ to SI -1 (4 to 5). It discovers that P_1 has not seen P_2, so it writes its process id at index 6, makes SI = 7. Since the QueueSize is 7 and we store SS as circular, sets SI to 0.
  - $\boxed{@@@P_1P_2P_1P_2}$
- When the scheduler gives turn to P_3, it reads SI, writes its process id at index 0, makes SI = 1 and sets MyLifeStartedAt$_3$ = 0
  - $\boxed{P_3@@P_1P_2P_1P_2}$
- When the scheduler gives turn to P_1, reads ScheduleSnapshot vector and discovers that all the processes have been inside the memory at least once since his last access. It writes itself with hat at index 1, updates SI to 2 and sets TrustLifeStarted to MyLifeStartedAt$_1$(= 3)
  - $\boxed{P_3\overline{P_1}@P_1P_2P_1P_2}$
- When the scheduler gives turn to P_2, it reads ScheduleSnapshot vector and discovers that it has seen a process with hat. So it reads TrustLifeStarted. It does not update TrustLifeStarted because it can not have a larger history. It reads ScheduleSnapshot from TrustLifeStarted to SI-1 and discovers that it has seen all processes, so writes itself with hat at location 2 and updates SI to 3
  - $\boxed{P_3\overline{P_1P_2}P_1P_2P_1P_2}$
- When P_3 comes, it sees a processes with hat in his history. So, it reads TrustLifeStartedAt (which is 3), starts reading the SS vector from this location. It discovers that everybody has voted except itself, so it votes and set T to 1.

3.4 Space Analysis

In this section we show how we calculate the Size of ScheduleSnaphot(SS). ScheduleSnapshot is a circular queue that has $n^2 - n + 1$ entries where each entry is $\lceil log_2(n) \rceil + 1$ bits. In our proof we take b bigger than $n^2 - n + 1$. For different values of n, we give the worst case for memory allocation.Within the boxes we write ScheduleSnapshot and the schedule is same as SS.

- For $n = 2$

  $\boxed{P_1 P_2\ \overline{P_1}}$

  The number of entries in SS: $n + (n-1)$

- For $n = 3$

  $\boxed{P_1 P_2\ P_1 P_2\ P_3\ \overline{P_1 P_2}}$

  The number of entries in SS: $n + (n-1) + 2(n-2)$

- For $n = 4$

  $\boxed{P_1 P_2\ P_1 P_2\ P_3\ P_1 P_3\ P_2 P_3\ P_4\ \overline{P_1 P_2 P_3}}$

  The number of entries in SS: $n + (n-1) + 2(n-2) + 2(n-3)$

- For $n = 5$

  $P_1 P_2\ P_1 P_2\ P_3\ P_1 P_3\ P_2 P_3\ P_4\ P_1 P_4\ P_2 P_4\ P_3 P_4\ P_5\ \overline{P_1 P_2 P_3 P_4}$

  The number of entries in SS: $n + (n-1) + 2(n-2) + 2(n-3) + 2(n-4)$

- To generalize the number of entries in ScheduleSnapsot for n processes is:

$$n + (n-1) + 2\sum_{i=2}^{n-1}(n-i) = n^2 - n + 1$$

  We call QueueSize as the size of ScheduleSnapshot and it contains $(n^2 - n + 1)(\lceil log_2(n) \rceil + 1)$ bits.

3.5 Run Time Analysis

Claim1: The upper bound for Run Time of Algorithm is $3b - n + 1$ where b is the bound of bounded scheduler and n is the number of processes.
Proof1: Lets assume that the schedule is given in the box below.
$\boxed{P_1 P_2 \ldots P_n \uparrow\ P_n P_n P_n P_n \uparrow P_1 P_2 \ldots P_{n-1} \ldots\ P_n \uparrow}$
In the first b steps(to the first arrow),all processes take turn at least once.The worst case scenario is, P_n to take $(b - n + 1)$ turns consecutively after the first b steps(in figure to the second arrow). From the second arrow, other processes take turn since scheduler does not ignore a process more than b steps. That's why all processes except P_n votes. P_n can be given turn at any point between the second arrow and third arrow. In the worst case, P_n is given turn at step $3b - n + 1$. It understands that every process has voted and makes T= 1

3.6 Proof of Safety

Claim2: When one of the processes makes T = 1,our algorithm ensures that every process has voted.
Proof2: The process that sets T = 1, checks its MyHistory $_{id}$. If it sees all processes except itself with hat,it votes and makes T = 1. Therefore when the algorithm terminates,it is ensured that every process has voted.

3.7 Proof of Termination

Claim3: Eventually one of the processes makes T = 1 in order to end the consensus protocol.
Proof3: As we show in section *Run Time Analysis*, upper bound of our algorithm is $3b - n + 1$ steps. In other words, the winner of the elections is selected in $3b - n + 1$ steps in the worst case - so, in a finite amount a time.

4 Minimum Total Cost Algorithm

In this section, we propose a $O(bnlog\ n)$ total cost algorithm for consensus. The algorithm requires $O(nlog\ n)$ shared memory and takes $O(b)$ rounds to reach consensus. There are no assumptions on the initialization of memory. The working of algorithm is independent of the initial state of memory. We give the complete pseudocode in the algorithm 1.

Let us first look at the different components of the system.

Shared Memory In the shared memory we maintain the following four data structures.

1. **Data Queue** A queue, called *data*, of size $2n$ is maintained in the shared memory. In this queue each cell is of size $\lceil log_2(n) \rceil$ and stores the favourite of the cardinals.

2. **Hat Vector** we need a vector of $2n$ bits, called *hat* for storing the flags which state if the cardinal has expressed his favourite, and if he has also voted.

3. **Counter** There is also a $2\lceil log_2(n) \rceil$ bit counter, called *index*, which specifies the location where the current cardinal is allowed to write his data.

4. **Min Counter** Finally we maintain a $2\lceil log_2(n) \rceil$ bits minCounter, which stores the index of last reliable memory access, such that after this location we can find the favourites of n cardinals in the n consecutive locations of the queue.

Private Memory Each cardinal has $3\lceil log_2(n) \rceil + 2$ bits of private memory. Cardinal uses $\lceil log_2(n) \rceil$ bits, called *favor* to store the ID of his favorite cardinal. A register of $2\lceil log_2(n) \rceil$ bits are required to store the memory location or the index of queue in shared memory where the cardinal has written the ID of his favorite. We call this register *memLoc*. 2 bits are required to differentiate between two rounds of algorithm, namely *proposal* and *voting*. We need 2 bits for 2 rounds because once a cardinal has voted for the winner, we need to have a 3^{rd} value to ensure that the cardinal doesn't vote twice. We call this set of two bits *activated*

The algorithm works as follows. The scheduler selects a cardinal on some scheduling protocol and evokes the cardinal procedure (*cardinalProcess*) with data structure of the selected cardinal as the paramemter. The selected cardinal checks the value of activated flag in his private memory. Based on the value of activated flag the cardinal either makes the proposal or votes.

Algorithm 3 Algorithm with minimum number of steps

```
if activated_i=0 then
  activated _i=1
  F[i] = F_i
  Temp = SnapshotIndex MOD QueueSize
  CurrentSnapshotIndex = Temp + QueueStartAdress
  SS[CurrentSnapshotIndex] = P_i
  myLifeStartedAt_i= = CurrentSnapshotIndex
  SnapshotIndex = SnapshotIndex++ MOD QueueSize
else
  CurrentSnapshotIndex = SnapshotIndex + QueueStartAdress -1
  if SnapshotIndex < myLifeStartedAt_i then
    Read the SS in a circular manner
  end if
  MyHistory_i[ ] =SS[myLifeStartedAt_i] ... SS[CurrentSnapshotIndex]
  if T = 0 then
    if If last element of MyHistory_i = (P_i or P_i-bar) then
      DO NOT Write into SS
    else if there is a process with hat in MyHistory then
      if SnapshotIndex < TrustLifeStartedAt then
        Read the SS in a circular manner
      end if
      TempHistory_i[ ]=SS[TrustLifeStartedAt] .. SS[CurrentSnapshotIndex]
      if (TempHistory_i[ ] < MyHistory_i[ ]) then
        TrustLifeStartedAt = myLifeStartedAt_i
      end if
      MyHistory_i[ ] =SS[TrustLifeStartedAt] ... SS[CurrentSnapshotIndex]
      if V_i = 0 then
        Read F[n] and decide for the leader in favor of majority
        V_i = 1
        if all processes except P_i have hat  then
          T = 1
        else
          SS[CurrentSnapshotIndex] = P_i-bar
          SnapshotIndex = SnapshotIndex++ MOD QueueSize
        end if
      else
        DO NOT Write into SS
      end if
    else
      if P_i sees all process ids in MyHistory_i then
        TrustLifeStartedAt = myLifeStartedAt_i
        SS[CurrentSnapshotIndex] = P_i-bar
        V_i = 1
        SnapshotIndex = SnapshotIndex++ MOD QueueSize
      else if All Processes in MyHistory_i has seen P_i then
        DO NOT Write into SS
      else
        SS[CurrentSnapshotIndex] = P_i
        SnapshotIndex = SnapshotIndex++ MOD QueueSize
      end if
    end if
  end if
end if
```

Algorithm 4 Algorithm with minimum number of rounds and Total Cost

```
1:  cardinalProcess(cardinal)
2:  if cardinal.activated = 0 then
3:     cardinal.activated := 1
4:     index++
5:     data[index] := cardinal.favor
6:     hat[index] := 0
7:     cardinal.memLoc := index
8:  else if cardinal.activated = 1 then
9:     state = 0
10:    state = chkQueue(cardinal,hat,index)
11:    if state = 1 then
12:       index++
13:       hat[index] := 1
14:       minCounter := cardinal.memLoc
15:       putVote(cardinal,data,minCounter)
16:       cardinal.activated := 2
17:    else if state = 2 then
18:       index++
19:       hat[index] := 1
20:       putVote(cardinal,data,minCounter)
21:       cardinal.activated := 2
22:    end if
23: end if
24:
25: chkQueue(cardinal,hat,index)
26: count := 1
27: while mod(cardinal.memLoc+count,2n) ! = index+1 do
28:    if hat[mod(cardinal.memLoc+count,2n)] = 0 then
29:       count++
30:    else
31:       return 2
32:    end if
33: end while
34: if count = n then
35:    return 1
36: else
37:    return 0
38: end if
```

Proposal round If the value of activated is 1 that means the cardinal is entering the shared memory for the first time and is yet to copy his proposal in the shared memory. The cardinal increments the *index* counter by one and copies his favorite in the memory location pointed by the index. the *index* is maintained as a vector of $2n$ bits. So, incrementing beyond this point just makes the counter start from the beginningof the queue. The cardinal also sets the *hat* flag corresponding to value in *index* to 0 and copies the *index* to *memLoc* in his private memory.

Voting Procedure If the value of activated flag is 2, that means the cardinal is already through the proposal round and is ready to vote, once he gets the complete information about the other cardinal's favorite. The cardinial decided if he should vote depending upon the value returned by the procedure *chkQueue()*. If the value returned is 1, that means the cardinal is the first one to express his proposal in the queue. The cardinal computes the cardinal with maximum proposals and votes for him. He sets the value of *minCounter* to the value of *memLoc*, so that the cardinals who come to vote in the future rounds know the starting address of the queue. In addition to this, the cardinal sets the *hat* flag to 1 corresponding to location pointed by index. If the value returned by *chkQueue()* routine is 2, then the cardinal just looks at the value stored in *minCounter*, counts the maximum number of votes from this starting point in the queue, votes, sets the *hat* to 1 and exits.

ChkQueue Procedure This procedure checks, if there exists a *hat* flag with value set to 1 in the hat vector since the value of *memLoc*to current value of *index*. If yes, then the procedure return 2, else the procedure counts the number of *hat* flags set to 0 in the hat vector betwwen the value of *memLoc*to current value of *index* and returns 1 if this value is equal to n else returns 0.

4.1 Complexity Analysis

In this section we give the run time and space analysis of the algorithm and give a proof of their exactness.

Claim 1: The number of memory access between the starting of Proposal Round and Voting round can be atmost $2b - 2$.
Proof of claim 1: The proposal round starts with the first process, entering the shared memory and writing his favorite in the location specified by the *index*. If $S = P_1P_2....P_m$ is the sequence of processes that enter the shared memory, with $P_i \neq P_j$ not necessarily, then the latest P_1 can repeat itself in the Proposal round is at $i = b - 1$. If $i = b$ then, that means all the processes are already in once and at the worst this i specifies the start of Voting round. The voting round will start when P_1 appears again in S after $i = b - 1$. Since the scheduler is bounded, this can be atmost $j = i + b$ (because bound on scheduler is b). At this point (at $j = 2b - 1$), the voting round starts. So, number of memory access between start of Proposal round and Voting round can be atmost $2b - 2$.

Claim 2: The voting round needs atmost b memory accesses to terminate.
Proof of Claim 2: Since, the scheduler is bounded, if we look at a window of size b, we should have all the processes existing inside the window atleast

once. For Voting round to terminate we need all the processes to go through the shared memory atleast once. In the worst case, this may require b accesses (because size of window can be atmost b).

Theorem 1: The algorithm takes $3b-2$ memory accesses to terminate.
Proof of Theorem 1 The proof of algorithm directly follows from Claim 1 and 2. The algorithm goes through 2 different rounds, namely Proposal and Voting and terminates when the voting round terminates. From clain 1 we see that number of memory access before the voting round starts are $2b-2$ and from claim 2 it follows that it takes atmost another b rounds for voting to finish. So, total number of accesses for the program to terminate are atmost $3b-2$.

The algorithm takes $3b-2$ rounds in the worst case to arrive at the consensus. Every cardinal needs to go through the complete process twice. Once for expressing the favourite and 2^{nd} time to make their vote. If we consider a bounded scheduler with a bound b, then two rounds of scheduling (i.e. $2b$ memory accesses) should be enough for algorithm to terminate. But, the algorithm is designed so that, the first cardinal which expresses his favourite has to be the first one to cast vote and set the *minCounter*. The other cardinals then look at this minCounter, compute the cardinal with maximum liking and vote for him. In, the worst case, as given in Fig, the first cardinal can be scheduled at the latest $2b-1$ rounds from his first access. Now, for all the other cardinals to vote, it may take atmost another $b-1$ rounds to get access to the shared memory. So, the maximum number of memory access are $3b-2$ which is linear ($O(b)$) in bound b.

Since, each process makes entry in the queue only twice, and the queue being circular, we need queue of exactly $2b$ size. Each entry in the queue is of size $\lceil log_2(n) \rceil$, because we can encode ID of n cardinals using $\lceil log_2(n) \rceil$ bits. So, as explained in section 4, we need only $2(n+2)\lceil log_2(n) \rceil + 2n$ bits of memory.

If we define the *Total Cost* as memory times the number of memory access, then total cost of the algorithm is $O(bnlog\ n)$.

5 Cheating Analysis

We make an assumption that the shared memory is error-free and the cardinals agree on the same protocol - so, every cardinal knows the correct code of every other cardinal. Cheating means that at least one cardinal runs a modified protocol - dependent or not of him - than the one it should. Therefore, all the bad things can happen at the level of the cardinals:

- their program is altered s.t. they do not respect the protocol
- their program can be corrupted by insertion of faults in their private memory, or in their code.

We have two goals: cheat detection and cheat correction. The latter objective is definitely harder. We list here some types of cheating that might be involved in the consensus and the leader selection problem.

Forging The cardinal can overwrite the data written in the shared memory by some other cardinal, leading to false selection of leader and may completely

jeopardize the consensus protocol. The only way one can avoid such cheating is by using some hardware protection. In other words, a cardinal is given write permission only in a specific part of the memory, which is decided by the scheduler. Otherwise, the cardinals have the read-access for complete shared memory.

Proposal Cheating The cardinal cheats by proposing a cardinal different from his favorite. We cannot propose any countermeasure to the general case, as long as the value that he is proposing is proposed consistently. This can be sub-categorized into the following types of cheating :

1. **Selfish Cheating** The cardinal proposes himself. This is not a very intelligent cheating because the cardinal can set his favorite to his own ID at the beginning of protocol.

2. **Intelligent Cheating** The cardinal proposes a different cardinal than the one nominated by his Fi variable, in order to vote with the cardinal that will win (or has big chances of winning), given the fact that he knows some of the other cardinals' proposals (we are not considering the cases $n_¡=2$), s.t. he will be one of the new pope's favorits

Anarchist The cheating cardinal votes with a different cardinal than the one that should get ellected in a correct implementation, based on the public proposal information that has been transfered into the SM (not based on the private F_i)

Crazy Cheating Any combination of the above mentioned proposal cheatings, at different moments or simultaneously, can be categorized under this heading.

Cheat Resistance of Algorithm 4 Our algorithm is resistant to the *Intelligent Cheating* as defined above. This is because, when the cardinal expresses his proposal, he doesn't know if he is the first one to enter the shared memory and can't rely on the data written in other memory location. So, he expresses his vote independent of the favorit's of other cardinals. Only data he can rely on in the memory, is the one that corresponds to value of *memLoc* and the values stored in the queue after this location (which is not reliable unless the cardinal sees some *hat* flag set to 1, but by that time all the cardinals have already expressed their favorite). We can avoid the *forging* kind of cheating by using the hardware protection. We say that the cardinal is only allowed to write in the memory locations pointed by the *index* counter but can read the complete memory. By giving this feature we can avoid cheatings which are most likely to occur in normal circumstances.

We would like to underline that our solutions with 3-bit Shared Memory and minimum number of steps do not rely on the leader to centralize the data. The minimum time solution does not use actually any leader. The 3-bit solution uses the leader only to synchronize the communication of the cardinals.

Cryptographic Approach We concentrate our attention on the forging and intelligent cheating, and propose a protocol that will remove the possibility that any cardinal can read the proposals of the other cardinals before expressing his own proposal. We employ a cryptographic scheme (that itself is imperfect, but makes the probability of breaking it infinitesimal - for example finding the actual private key when you know the public key, or finding another private key that corresponds to the private one). Our system uses:

- a public (or asymmetric) key infrastructure - we can use RSA with keys of 1024 bits or more. We use it to perform the digital signature of the proposal values, s.t. we remove the possibility of forging.
- a private (or symmetric) key infrastructure - we can use AES with a key of 128 bits or more, or 3DES. We use it in order to disallow a cardinal to find out the other cardinals' proposals before expressing his proposal, thus disallowing intelligent cheating.

Each cardinal has from the initialization, in his private memory:

- every cardinals' RSA public key
- his RSA private key
- his AES key

The space complexity of the algorithm is $O(nlog\ n)$ and time complexity is $O(n^2b)$. A sketch of the protocol is given below:

- Phase 1: every cardinal writes in the SM AESKey(proposal) followed by the digital signature: RSAPrivateKey(AESKey(proposal)). We notice that we are not digitally signing a digest / hash (e.g. obtained with MD5, etc) of the proposal because we assume that the proposal can be expressed on a number of bits between 1 and 200 normally (we assume $n < 2^200$), so we do not need to apply a hashing algorithm to obtain a message of about 100 bits. In the case $n > 2^200$, we can apply the hashing techniques, but we discard these details from our presentation.
- Phase 2: every cardinal reads from the SM all the proposals written in the previous step, and makes a copy of the values in his private memory. Each cardinal i checks for each cardinal j that his proposal is correctly signed by checking if the following equality holds:
  $RSAPublicKey_j(RSAPrivateKey_j(x)) = x$
  where $x = AESKey_j(proposal_j)$
  If, for one of the cardinals j, this equality does not hold then his proposal is corrupted - someone else has over-written (or it might be that cardinal j has signed wrong). In this case, if we want to attain correction cheating (instead of just only detection) all the cardinals that respect the (correct) protocol can continue to loop through phases 1 and 2 until the equality holds for every cardinal.
- Phase 3: every cardinal writes in the SM, AESKey followed by the digital signature: RSAPrivateKey(AESKey)

- Phase 4: every cardinal reads the AESKey and starts decoding everybody's else AESKey(proposal) and decides correctly. More exactly, each cardinal i:
  - checks that for each cardinal j, the $AESKey_j$ is signed correctly by checking that the following equality holds: $RSAPublicKey_j(RSAPrivateKey_j(AESKey_j)) = AESKey_j$
  - decodes for each cardinal j, his proposal, that is given by the following cryptographic function applications: $AESKey_j(AESKey_j(proposal_j))$
  - decides by picking deterministically (as all the others) one of the candidates that has the majority of votes (e.g, picks the cardinal with majority of votes, and, in the case of ties, the one with the lowest id).

During each phase we perform synchronization with prisoner consensus s.t. the processes know that each other process has written/read the values in the SM. We can see that this last protocol addresses proposal cheating in the case of intelligent cheating, but not in the more general case like the anarchist cheating. This protocol helps in detecting cheating but cannot easily address the problem of correcting the election process.

6 Comparison of Algorithms

In this section we give a comparison of the algorithms that we propose. In table 1 and 2, we present costs for time and space. In table 1, we give the exact cost. In table 2, are given the cost with BigO notation. Finally in table 3 we present the total cost with BigO notation.

Table1

Algorithm	**Time**	**Shared Memory**	**Private Memory**
Minimum Run Time	$3b - n + 1$	$2logn^2 + (n^2 + 1)logn + 1$	$2logn^2 + (n^2 - n + 5)logn + 2$
Minimum Memory	$nb + 2n^2blogn$	3	$nlogn + 2$
Optimum Cost	$3b - 1$	$2(n + 2)logn + 2n$	$n(3logn + 2)$

Table2

Algorithm	**Time**	**Shared Memory**	**Private Memory**
Minimum Run Time	$O(b)$	$O(n^2logn)$	$O(n^2logn)$
Minimum Memory	$O(bn^2logn)$	$O(1)$	$O(nlogn)$
Optimum Cost	$O(b)$	$O(nlogn)$	$O(nlogn)$

Table3

Algorithm	**Cost**
Minimum Run Time	$O(bn^2logn)$
Minimum Memory	$O(bn^2logn)$
Optimum Cost	$O(bnlogn)$

7 Conclusion

In this paper, we give algorithms for the consensus problem in the case of uninitialized memory. We present three algorithms with different tradeoffs between

the size of the shared memory and the number of steps. Finally, we show that our optimal cost algorithm is cheat free with some constraints on the definition of cheating.

A Appendix A. Another Minimum Space Algorithm

We present a protocol for our problem that uses 2 bits. This protocol has an order of time complexity of $O(n * b + n * log(n) * b)$ steps. This protocol is inspired from our solution from the "channel" paradigm with the four states:

- NT (Not Transmit - there is no information transmitted on the channel)
- T0 (some process is transmitting a 0 bit)
- T1 (some process is transmitting a 1 bit)
- ACK (the recipient process of the 0 or 1 bit has received the bit).

We can encode the states of the channel on 2 bits that we call bit1 and bit2.

If the SM is initialized with a known value - the processes know apriori that bit1=v1 and bit2=v2 - we can specify in the protocol that the encoding of the states of the channel are:

- NT = bit1 = v1, bit2 = v2
- T0 = the binary value of NT + 1
- T1 = the binary value of T0 + 1
- ACK = the binary value of T1 + 1

The algorithm uses once again the leader-follower paradigm. The leader is the one who is centralizing all the other processes proposals, and then can judge who is winning the elections and communicate the results to all the other processes.

- a follower when it is invoked - if it finds the channel in the NT state, if the follower did not send his proposal yet, it starts transmitting his proposal to the leader. It will set the state of the channel into T0 or T1 (depending on the value he wants to transmit). A very important remark is that this follower has to send all of its data of the proposal to the leader, bit by bit. IMPORTANT: This is the way that allows the leader to keep track of from whom the bits are coming.
  When the follower sends all the bits of his proposal, then he will get into a state where he will not send any more data, and will wait from the leader the result of the elections, and then he will update his V_i variable.

- when the leader is invoked it reads the state of the channel. If it is T0 or T1, he stores into his private memory the value of the bit on the channel. Then:

- if the leader has not received another full word of ceil(log2(n)) bits, it sets the state of the channel to ACK. By setting the state to ACK (and not NT), we are making sure that only the follower that is in the process of sending his proposal will write on the channel.
- if the leader has received another full word of ceil(log2(n)) bits,
  * if the leader has received all the proposals it starts sending the result of the election to every follower. For this a good solution is to broadcast the result (of ceil(log2(n) bits) to all the followers. For this we use prisoner consensus to make sure every follower reads every bit of data (just like in our Minimal space solution in Section 2). When the leader is sure of that he sets T=1.
  * if the leader did not receive all the proposals he sets the channel to the NT state, s.t. another follower can send his proposal.

Initially, we thought to implement the communication of the result of the elections, by the leader, to each follower by using unicast with the channel solution. But the channel solution is good in the case that we have a fixed destination of the messages and not various destinations for the message. If we have the following example:

- n=3, b is considerably bigger than n
- the SM is initialized with NT
- P0 is the leader that gathers the data
- by Tx we designate a transmit state of the channel where we don't care about the value of the transmitted data
- We have the following scheduling: P1 (P1 does nothing) — P0 (P0 writes Tx) — P1 (P1 reads from the channel Tx and write ACK) — P0 (P0 reads the ACK and sets channel to Tx) — P2 (P2 does not know that P0 has communicated to P1 2 bits, so he can think that this Tx is meant for him)

As we can see in this example, the processes cannot know the state of the system. We claim that in the case of the leader sending the message to every follower, one cannot know the state of the system without performing synchronization (e.g., through a barrier implemented with prisoner consensus).

Now, in the case that the SM is uninitialized, what we propose is to use the prisoner consensus algorithm s.t. we are able to agree on the encoding of the state of the channels and initialize the channel to the state NT, and then use the "channel" algorithm for the initialized SM. The values of the 2 shared bits (that are NOT known apriori by the processes) are bit1=v1 and bit2=v2. We use bit1 as the Sw bit of the prisoner consensus algorithm, and bit2 as the Announce bit. When bit2 has value v2 then the leader has not announced that the prisoner consensus has finished. When bit2 has value !v2 then the leader has announced that the prisoner consensus has finished. This prisoner consensus with (the typical) initialized Sw bit and the addition that the Announce bit is uninitialized will work because every process will see the uninitialized (v2) value

of the Announce bit at least once before the leader will flip the bit. At the end of the prisoner consensus, the bit1 will be set to 0 and bit2 will be set to !v2. Then, we specify in the protocol the following encoding of the states of the channel:

- NT = bit1 = 0, bit2=!v2
- T0 = the binary value of NT + 1
- T1 = the binary value of T0 + 1
- ACK = the binary value of T1 + 1

So we have "initialized" the shared memory by making all processes agree on the encoding of the states of the channel, depending on the initial uninitialized values of the SM, and having the first state of the channel NT.

References

[1] Jayadev Misra. A consensus protocol in a prison.