Project Report

João André Pestre
Thiago Felicissimo Cesar

June 10, 2020

1 Introduction

The problem of consensus is fundamental in real-world systems. It is the basis of many distributed systems and blockchain algorithms. The Synoid Algorithm is an obstruction free algorithm that solves consensus on the message passing model. Its main use is implementing state-machine replication through the Paxos algorithm.

In this project we implement the Synoid Algorithm using the Akka framework in Java. We run some tests, for which we present the results and a discussion of them.

2 Problem Statement

Before presenting the algorithm, we should understand in a precise way the problem's statement. The classical statement of the consensus problem is captured by three requirements, as shown on the definition bellow.

Definition 1 (Consensus) *We say that an algorithm solves consensus if*

1. ***Agreement****: no two processes output different values;*
2. ***Validity****: the output value is an element of the set of proposed values;*
3. ***Termination****: every correct process outputs a value.*

For obstruction-free consensus, the Termination condition is replaced by OF Termination, whose formal definition is given below.

Definition 2 (Obstruction Free) *An algorithm is said to be obstruction-free if*

1. *If a correct process proposes, it eventually decides or aborts;*
2. *If a correct process decides, all correct process eventually decides;*
3. *If from a moment on only a single process make proposals and for a unbounded number of times, then it eventually decides.*

Note therefore that whereas a normal consensus algorithm is designed in a one-shot way — that is, the process runs it only one time —, an obstruction-free algorithm might be required to run multiple times until it can output a value without aborting.

In our case in particular, we place ourselves in a t-resilient non-anonymous asynchronous message passing system. That is, we assume that at most a minority of process may fail and that every process has an identifier which identifies its messages. The communication channel is assumed to be reliable, however message delays are unbounded.

3 The Synoid Algorithm

We now present the Synoid Algorithm on a high level. It is based on two phases: a read phase and an impose phase. Every process keeps many local variables, but to keep this discussion simple we will only mention two of them: a value which tells how recent the process request is[1] and another value that holds its estimate of the decided value.

On the read phase a process tells a majority of the process that it wants to read the "state" of the system. This in practice allows it to know if a previous process successfully imposed a value. A process that receives a read message sends it a gather message with its estimate of the decided value if it has not seen a more recent process. Otherwise, it sends an abort message to the process.

A process that receives only gather messages and no abort messages from a quorum finishes the read phase and adopts the most recent value before trying to impose it. The idea here is that if some previous process has already successfully imposed a value, we want this new process to try to impose the same value. Actually we can show formally that if a process has already imposed a value then any other process that reads from a quorum will always read the imposed value.

On the impose phase the process send his value to all the processes and waits for a quorum to respond. Every process that receives this value checks whether it has heard from another more recent process, in which case it tells the process to abort. Otherwise, it accepts the value as its new estimate to the decided value and sends an acknowledgement to the process.

An imposing process that receives only acknowledgement messages and no abort messages from a quorum finishes the algorithm and decides the value. It also tells all the other processes in which value it decided, so any process that learns it also finishes the algorithm and decide on it.

4 Implementation Description

We implemented the algorithm on the Akka framework in a vary faithful way, without making changes. Concerning technical details of the Akka framework, the main first creates all the system's processes and sends to each one of them the references to the other processes, so they are able to communicate among them. Then it sends a launch request to all the processes, and for a minority of processes randomly selected it sends on this request a boolean which tells the

[1] The notion of ordering of operations is defined using ballots. To keep this discussion on a high level we deal with it on an informal way. For a more formal description of how ballots can be used to order operations refer to the pseudo-algorithm on the slides and to the proof of correctness shown in Appendix A.

process that it is faulty. A faulty process may fail forever for each message that it reads. Each time it treats a received message it has a 50% probability of halting.

As we know, an obstruction free algorithm can run forever if at all instants two or more processes are running simultaneously. Therefore, after launching the system the main waits for a fixed amount of time before electing a process to run alone. This amount of time is called *timeout to leader election* and is represented by the variable t_{le}. After selecting this process, the main tells all but this selected process to stop proposing values. Because the algorithm is obstruction-free, once this process starts to run alone we know it will eventually decide on a value.

5 Performance analysis

To analyse the performance of the algorithm, and evaluate the importance of the *timeout to leader election*(t_{le}) variable we ran and timed some executions. For each pair of values of $n \in \{3, 10, 100\}$ and $t_{le} \in \{0.5, 1.0, 1.5, 2.0\}$ we ran 5 executions of the algorithm until the first process decides. We timed these executions from the moment before the first *launch* message was sent until the first process decides on a value. The time for each execution can be found on tables 1, 2 and 3, and the plot of average latency by *timeout to leader election* can be found in figures 1 and 2.

We can see in tables 1 and 2 that, for a small enough number of processes, consensus is achieved relatively fast. In this cases, we can not observe significant change in performance as a function of the timeout to leader election since all executions ended before the timeout was reached. This is observed as well in the plot in figure 1, where we can see the average latency varying slightly and with seemingly no correlation to t_{le}.

Latency - n=3					
Time until leader election	Exec 1	Exec 2	Exec 3	Exec 4	Exec 5
0.5s	14ms	4ms	2ms	6ms	3ms
1.0s	3ms	11ms	2ms	3ms	6ms
1.5s	1ms	8ms	8ms	13ms	2ms
2.0s	6ms	1ms	0ms	0ms	2ms

Table 1: Latency per execution for n=3

Latency - n=10					
Time until leader election	Exec 1	Exec 2	Exec 3	Exec 4	Exec 5
0.5s	2ms	5ms	9ms	2ms	1ms
1.0s	3ms	3ms	3ms	5ms	1ms
1.5s	1ms	5ms	10ms	12ms	1ms
2.0s	1ms	9ms	5ms	1ms	0ms

Table 2: Latency per execution for n=10

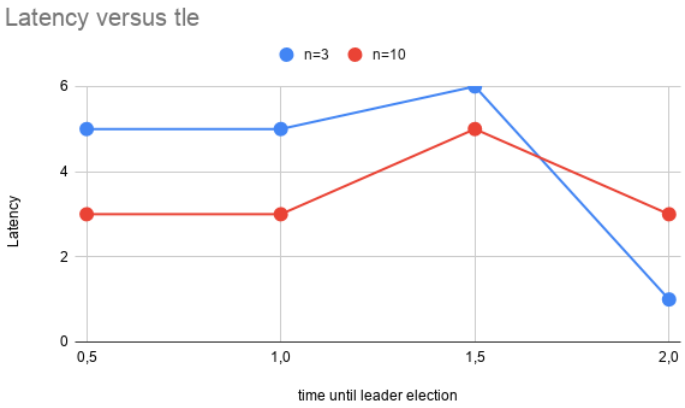

Figure 1: Latency versus tle - n=3 and n=10

For 100 processes, however, we can observe an interesting dependency on t_{le} in table 3. Even though we can still observe executions in which consensus is achieved before the timeout is reached, we see t_{le} acting as an *approximate* upper bound to the latency. We can see in executions 1 and 2 for $t_{le} = 0.5s$, execution 4 for $t_{le} = 1.0s$, executions 1 and 2 for $t_{le} = 1.5s$ and execution 1 for $t_{le} = 2.0s$ that, in runs where the latency surpasses the timeout, consensus is achieved not long after it.

Furthermore, observing the plot in figure 2 we can see the average latency clearly increasing as t_{le} increases. This shows that, on average, consensus is achieved faster when a leader is elected sooner, as expected.

Latency - n=100					
Time until leader election	Exec 1	Exec 2	Exec 3	Exec 4	Exec 5
0.5s	555ms	614ms	128ms	156ms	428ms
1.0s	342ms	346ms	335ms	1054ms	518ms
1.5s	1575ms	1517ms	189ms	976ms	747ms
2.0s	2035ms	250ms	647ms	1609ms	771ms

Table 3: Latency per execution for n=100

6 Conclusion

In conclusion, in this document we presented an implementation of the Synoid algorithm for solving obstruction free consensus using the Akka framework in Java and a proof of correctness for said algorithm.

A performance analysis of the implementation showed that, for a large enough number of processes, consensus is achieved faster the sooner a single process starts to propose alone. Furthermore, the analysis showed that, if a process starts to propose alone, the algorithm converges to consensus not long after it.

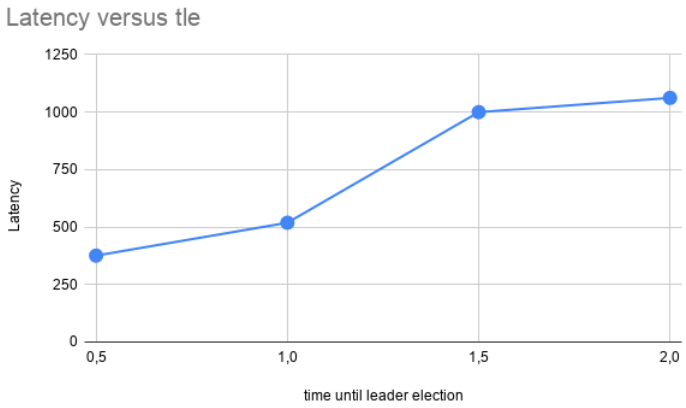

Figure 2: Latency versus tle - n=100

A Proof of Correctness of the Synoid Algorithm

We now present the proof of correctness of the algorithm. For a detailed description of all the algorithm's variables refer to the pseudo-code on the slides of the course.

Validity is trivial to show. We thus focus on OF Termination and Agreement.

Theorem 1 (OF Termination) *It holds on the algorithm that*

1. *If a correct process proposes, it eventually decides or aborts;*
2. *If a correct process decides, all correct process eventually decides;*
3. *If from a moment on only a single process make proposals and for a unbounded number of times, then it eventually decides.*

Proof. 1 comes from the assumption that there is always a majority, and thus a process can not get blocked while proposing. 2 follows from the fact once a process decides it broadcasts its value to the others, so they eventually receive it if they are correct.

To show 3 we simply remark that if only a process p proposes, its ballot will eventually be larger then all the `imposeballot` and the `readballot`, and thus it will be able to read a value and impose it, eventually deciding. Note that this argument only works because those values are constant if only p is proposing — in other words, no process is competing with p. ■

Theorem 2 (Agreement) *All the values returned by the algorithm are equal.*

Proof. Let i be the accepted impose operation of smallest ballot number. We call b_i its ballot number and v_i its returned value. Let R be the set of read operations that had success and whose ballot numbers are larger then b_i. Note that R can be ordered by the ballot numbers of the operations. Also note that any impose operation different from i is associated with a read operation in R. To show the claimed result it suffices to show that every read operation in R

returns v_i. We let elements of R be indexed according to the ballot order and we show this result by induction with respect to the index of the operation.

Base Case

Of all the read operations that had success and whose ballot number is larger then b_i, let r be the one of minimal ballot number. We call b_r its ballot number and v_r its returned value. We claim that $v_i = v_r$.

Let q be a process on the intersection of quorums that accepted i and r. Because $b_i < b_r$, we conclude that q accepted r after it accepted i. Moreover, q has not acknowledged an impose operation since i. Indeed, an impose operation is always proceeded by a successful read operation of same ballot number, and because b_r is the minimal among successful read operations such that $b_r > b_i$, there can be no impose operation i' of ballot number b' such that $b_i < b' < b_r$. This implies that, with respect to the order of operations seen by q, this i' could not happen before r. Therefore, `estimate` is still equal to v_i and thus this will be the value returned from q to r.

For the same reason as already described, v_i will be the value of largest ballot number returned by all the processes on the quorum of r — otherwise, this would also imply on the existence of an impose operation i' of ballot number b' such that $b_i < b' < b_r$, which would contradict minimality of b_r. Therefore, r must return v_i.

Induction Step

Let $r \in R_i$, and let b_r be its ballot number and v_r its returned value. The induction step says that $\forall r' \in R$ such that $b_i < b_{r'} < b_r$, $v_{r'} = v_i$. We claim that $v_i = v_r$.

Let q be a process on the intersection of the quorums that accepted i and r. Because $b_i < b_r$, q accepted i before r. Moreover, any impose i' that q accepted between i and r must be associated with a read operation $r_{i'}$ such that $b_i < b_{i'} < b_r$, and by the induction step this operation returned v_i. Therefore, `estimate` still holds v_i and this will be the value returned from q to r.

Among all the values received by r, v_i will be the one of largest ballot number — otherwise this would contradict the induction step, by a similar argument as used on the previous paragraph. Thus, $v_r = v_i$. ■