SLR210 - Obstruction Free Consensus

Project Presentation

João A Pestre, Thiago F Cesar

June 18, 2020

Table of Contents

The algorithm

Problem Statement

We define the problem of Obstruction Free Consensus as it follows.

Definition (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. **OF Termination**:
   - **3.1** If a correct process proposes, it eventually decides or aborts;
   - **3.2** If a correct process decides, all correct process eventually decides;
   - **3.3** 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 an one-shot way, an obstruction-free algorithm might be required to run multiple times until it can output a value without aborting.

Model of Computation

In our case in particular, we place ourselves in a t-resilient non-anonymous asynchronous message passing system. In other words,

- we assume that at most a minority of process may fail and that every process has an identifier which identifies its message;
- the communication channel is assumed to be reliable, and thus no messages get lost;
- message delays are unbounded, and thus a message can take an arbitrarily large amount of time to arrive.

Synoid Algorithm in a high level (1)

The algorithm is based on two phases: read and impose.

Read Phase

- The process tells the other processes that it wants to read the "state" of the system and waits for a quorum to respond.
- A process that receives a read message answers it with a message containing 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.

Synoid Algorith in a high level (2)

Impose Phase

- The process send the value he wants to impose to all the processes and waits for a quorum to respond.
- A 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 of 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 on the value. It also broadcasts the result to all processes.

Remark

The notion of more recent processes and operations is presented here in an informal fashion. This ordering can be defined formally using ballot values, in a way that allows us to proof the correctness of the algorithm.

Correctness arguments

To prove correctness of the algorithm, we need to show **Validity**, **Agreement** and **OF Termination**. Validity is trivial to show, as the decided value is always one of the input values

Theorem (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, because the values of `imposeballot` and `readballot` do not grow, its ballot will eventually be larger then them, and thus it will be able to read a value and impose it, eventually deciding. □

Theorem (Agreement)

All the values returned by the algorithm are equal.

Proof (Idea).

Let i be the accepted impose operation of smallest ballot number and let R be the set of read operations that had success and whose ballot numbers are larger then that of i.

Any impose operation different from i is associated with a read operation in R, and thus to show the claimed result it suffices to show that every read operation in R returns the value of i.

If we let R be ordered by their ballot values, the proof can be done by induction on this sequence. The idea is, for some read r, to look at the quorum intersection of r and i and to note that, because $b_i < b_r$, q accepted r after it accepted i and thus it will return the value written by i.

The possibility that this value was overwritten or that some other process returns to r a value of larger `estimatedballot` (a more recent value) contradicts the induction hypothesis. We thus conclude that r returns v_i.

The same argument can be repeated for any read, assuming that all previous imposes tried to write v_i. □

Implementation

Implementation

The algorithm described was implemented using the Akka framework. The simulation of crashes and leader election was implemented as such:

1. Crashing:
   - The list of processes is shuffled
   - The first f processes receive a *launch* message with a *faulty* flag set to *true*, the rest receives the flag set to *false*
   - For processes with a *faulty* flag, every event received has a 50% chance of not being ignored
2. Leader election:
   - One of the processes from f to $n-1$ is randomly chosen as the leader
   - A *leader election* message is sent for each process at time t_{le} holding the id of the leader
   - Processes that are not the leader stop proposing

Performance Analysis

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.

- $n \in \{3, 10, 100\}$
- $t_{le} \in \{0.5, 1.0, 1.5, 2.0\}$
- 5 executions were timed from first *launch* message until first process decides.

The data is collected in the following tables.

Performance analysis: tables

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

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

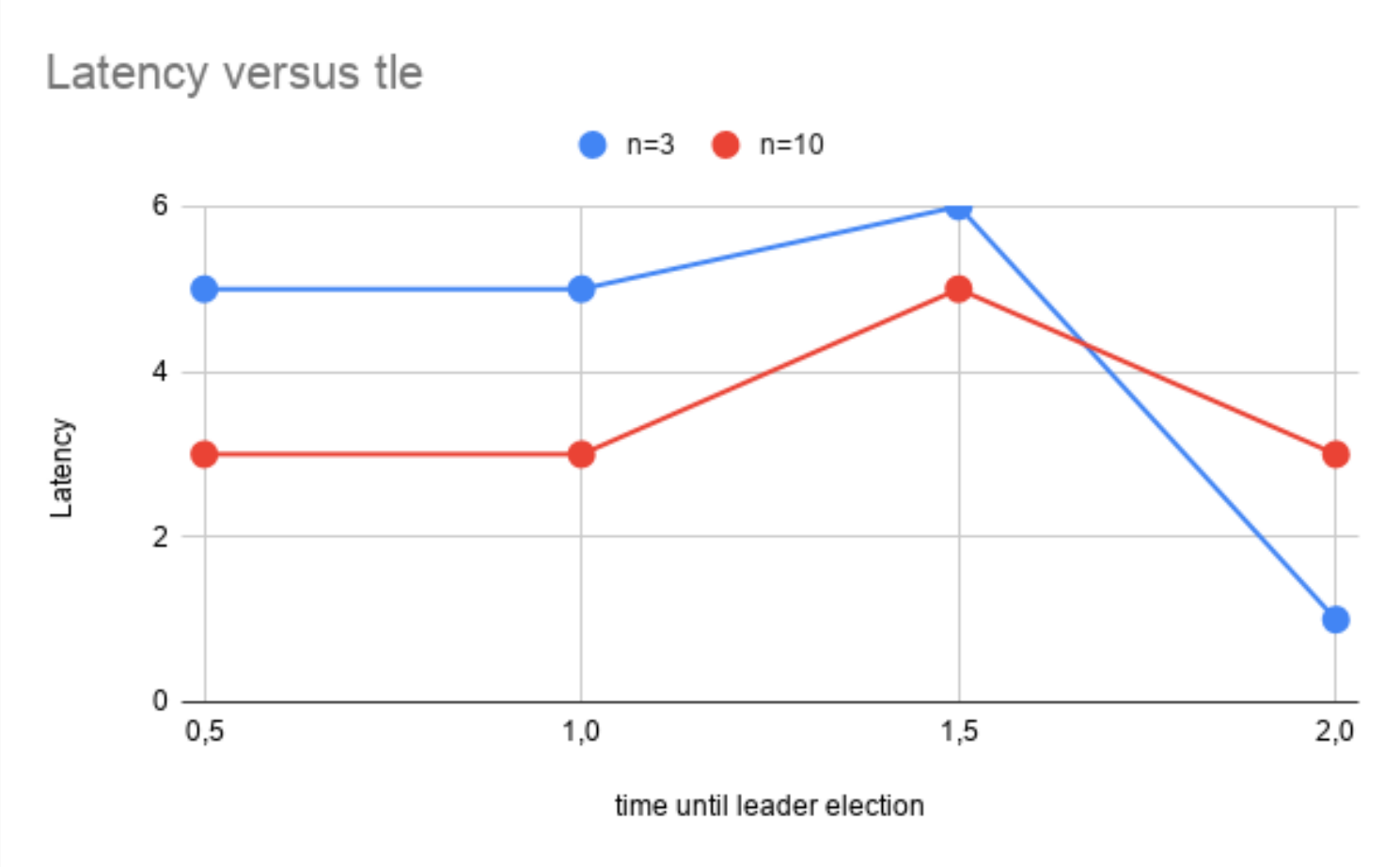
Performance analysis: tables

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

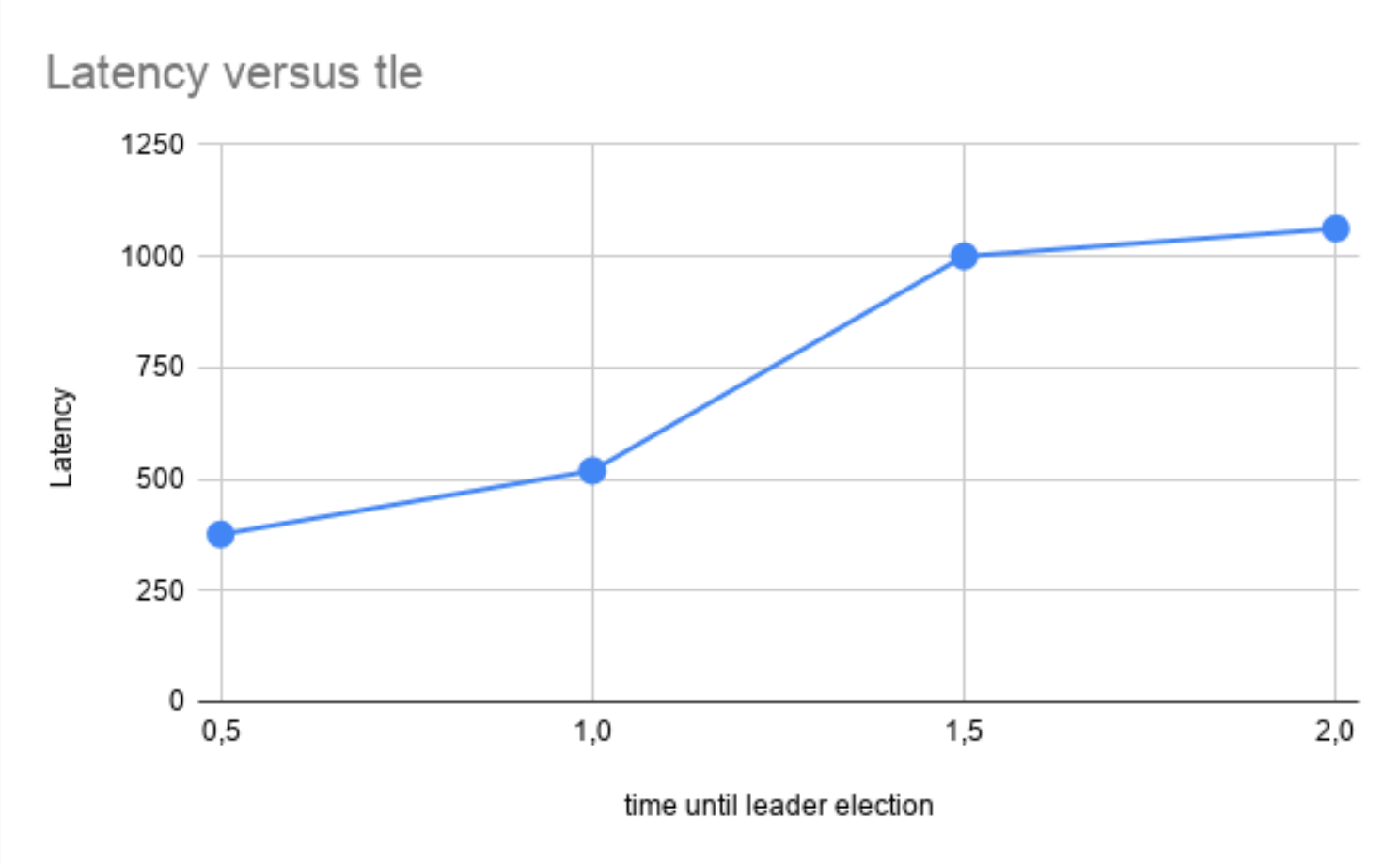
- For small n: consensus achieved before leader election
- For n large enough: t_{le} works as *approximate* upper bound

Performance analysis: plots



Average latency varies slightly with seemingly no correlation to t_{le}.

Performance analysis: plots



Average latency increases with t_{le}. Consensus is achieved faster when a leader is elected sooner.

Thank you!

Questions?