SLR210 Project

Obstruction-Free consensus and Paxos

by
Guittard Adrien
Wew Emile
Gélébart Mael

Télécom Paris
15/04/2024

Index

Problem and Context.....2

Implementation.....6

Performance Analysis.....7

Conclusion.....11

Problem and context

Ditributed computing and consensus:

In this project, we are in a distributed system environment. This means we have computer systems, which we will call processes, who are not in the same place. They do not share any data, nor variables and have different memory spaces. They communicate with each other through messages. We are also in an asynchronous system, which means that other the processes have no way of knowing what the other processes are doing (they are not synchronized). A classic (and very important) problem to resolve in distributed systems is called consensus. Basically, we want all processes to do the same thing, to agree with each other, to reach a **consensus.** Formally, we say that each process can propose an **input** value v (***propose(v)***) and then they can decide (***decide(v)***) on an **output** value v. We want all processes to decide on the same value. For context, a **correct** process, is a process that doesn't crash, doesn't lie, and eventually decides on an output value. We would like to have what is called **"wait-freedom",** all correct processes can finish their operation in a bounded number of steps no matter what all then other processes do. In order to resolve wait-free consensus, the following conditions should be met:

Agreement:

- No two processes decide on different values.

Validity:

- Every decided value is a proposed value.

Termination

- No process takes infinitely many steps without deciding

Allowing faulty processes:

Of course, in the real world computers can crash or fail. In this project, if we say **n** is our number of processes, we would like to allow up to **f<n/2 faulty** processes. These f faulty processes are not correct, at any point they can simply stop responding to messages and do nothing. If we allow this, it is impossible to solve consensus. This is because of the FLP theorem (Fischer, Lynch, Paterson), the proof is beyond the scope of this project. So, what now? Well if with faulty processes, the properties listed above cannot be met, maybe we can relax those properties and solve that problem...

Obstruction-Freedom:

The idea here is to find properties that are a bit less restrictive than the 3 we listed and that this will allow us to solve consensus in our system with faulty processes. For example, we allow processes to **"abort"**, meaning that they refuse a proposition. We keep Agreement and validity and we will relax termination for 3 new properties:

Obstruction-free termination:

- If a correct process p proposes, it eventually decides or aborts.
- If a correct process decides, no correct process aborts infinitely often.
- If there is a time after which a single correct process p proposes a value sufficiently many times, p eventually decides.

With these relaxation of the traditional consensus properties, it is possible to solve what we will now call **obstruction-free (OF) consensus.** Obstruction-freedom means any process can finish in a bounded number of steps if all of the other processes stop. Basically, if run in isolation, a correct process will be able to decide in a reasonable amount of time. However, if we read the Obstruction-free properties we notice that guarantees that we will decide in a reasonable amount of time. The **"sufficiently many times"** of the third point only describes out the behavior as time goes to infinity. It is totally possible that then processes will proposing and aborting over and over for a long time before a decision made. Since we are interested in a more real world scenario (with failures), we would like to have a guarantee of some sort that the processes will decide in a certain amount of time. This is where we have use for an **oracle**. An oracle is an external module that will give us extra information. In our case, this oracle will provide us, after a certain fixed time, a process that is **correct** and it will be considered the **leader.** With a leader selected (by **Ω**) , this simple piece of code will make sure that the leader and only the leader decides, and that all the correct processes will decide the same value as the leader:

Oracle:

This is where we have use for an ***oracle(1)***. An oracle is an external module that will give us extra information. In our case, this oracle will provide us, after a certain fixed time, a process that is correct and it will be considered the leader. With a leader selected (by **Ω**) , and this simple piece of code will make sure that the leader decides first, and that all the correct processes will decide the same value as the leader:

```
upon propose(v)
  while not(decided)
     if Ω  outputs self then
    result = ofcons.propose(v)
    if result=(decide,v') then
       send (decide,v') to all
       return v'

upon received (decide, v')
    return v'
```

For a process pi, if **Ω** doesn't output pi, pi will not do an OF.propose and keep looping in "while **not(decided)**" until it has decided. All the processes that are not chosen by **Ω** are not making any progress until they receive a **(decide, v')** message that makes them decide on v'.

If **Ω** outputs pi, then pi does an **ofcons.propose(v),** since no other process is proposing a value, it is as if all the other processes have stopped, nothing is obstructing pi's execution. The Obstruction-freedom guarantees us that process pi will decide in a bounded amount of steps. Since pi is the only one to have proposed a value and every decided value is a proposed value (validity), pi decides on its proposal v. Pi sends **decide(v)**. All the other correct processes decide v.

In our project, we assume we have access to such an oracle, but this means we still need an algorithm that solves OFconsensus in order to start testing.

Note: (1) It is not possible to have access to such an oracle in a fully asynchronous system. We are making the assumption that the system is **eventually synchronous**. This is a suitable assumption to make in a real world application.

Synod

Synod is an algorithm that solves OF consensus. A formal proof was seen in class, but it would take too long to detail it. What is important is how it functions. It is a 2 phase algorithm, first, processes propose values, secondly, they gather responses. Majorities of processes are needed to agree on values so that you can't have 2 majorities that decide differently. A ballot system is in place, where each proposition is attached to a ballot number and only the proposition with the highest ballot number is considered, the others are aborted. A process p decides on its proposal only if it has received acknowledgement from a majority of processes that p can safely decide. It then sends its decision to the other process who will decides on that same value.

Initially:
ballot:=i-n; proposal:=nil; readballot:=0;
imposeballot:=i-n; estimate:= nil; states:=[nil,0]n

upon propose(v)
proposal := v; ballot:=ballot + n; states:=[nil,0]n
send [READ, ballot] to all

upon receive [READ,ballot'] from pj
if readballot > ballot' or imposeballot > ballot' then
send [ABORT, ballot'] to pj
else
readballot:=ballot'
send [GATHER, ballot', imposeballot, estimate] to pj

upon receive [ABORT, ballot] from some process
return abort

upon receive [GATHER, ballot, estballot, est] from pj
states[pj]:=[est,estballot]

upon #states ≥ majority // majority of responses
if ∃ states[pk]=[est,estballot] with estballot>0 then
select states[pk]=[est,estballot] with highest estballot
proposal:= est //choose a potentially decided value
states:=[nil,0]n
send [IMPOSE, ballot, proposal] to all

upon receive [IMPOSE,ballot',v] from pj
if readballot > ballot' or imposeballot > ballot' then
send [ABORT, ballot'] to pj
else
estimate := v; imposeballot:=ballot'
send [ACK, ballot'] to pj

upon received [ACK, ballot] from majority
send [DECIDE, proposal] to all

upon receive [DECIDE, v]
send [DECIDE, v] to all
return [decide, v]

Implementation

Objectives of the project

Our goal for this project is to implement the synod algorithm and evaluate its performances under different constraints. The main parameters evaluated here are:

- **N** - The number of process in the system
- **f** - The number of process prone to failure (always < N/2)
- **Alpha** - The chance for a process to crash (stop responding) after making a step in the algorithm.
- **TLE** - The time before electing a leader (time before the system become synchronous, see note (1))

To determine the execution time, we mesure the first process to decide a value.
We aim to confirm the theoretical assumption made during the course, namely, determine what are the limitation of the obstruction free consensus, why an oracle is mandatory when scaling the number of processes and the correlations between alpha, f (the crashing of processes) and the performances.

Environment of the project

For this project, we are using the AKKA Actor model implemented in Java. It allows us to accurately model an asynchronous message passing system to implement the synod algorithm.
For the data gathering and its visualisation, we made a python script and a Jupyter notebook using the Pyplot package of Matplotlib.
All of the implementation including the python script and notebook are included in the archive of the project.

Performance analysis

Variables and measures

In the following, **N** is the number of processes, **TLE** is the time to leader election which the time we allow until a leader is forcibly chosen by the oracle. **α** is the probability of failure of the faulty processes. We always have **f= (N/2)-1** faulty processes. Each time we measured execution time for a given set of inputs, we did it 5 times and took the average of the 5 executions (the colored areas around the curves is the standard deviation of the 5 executions).

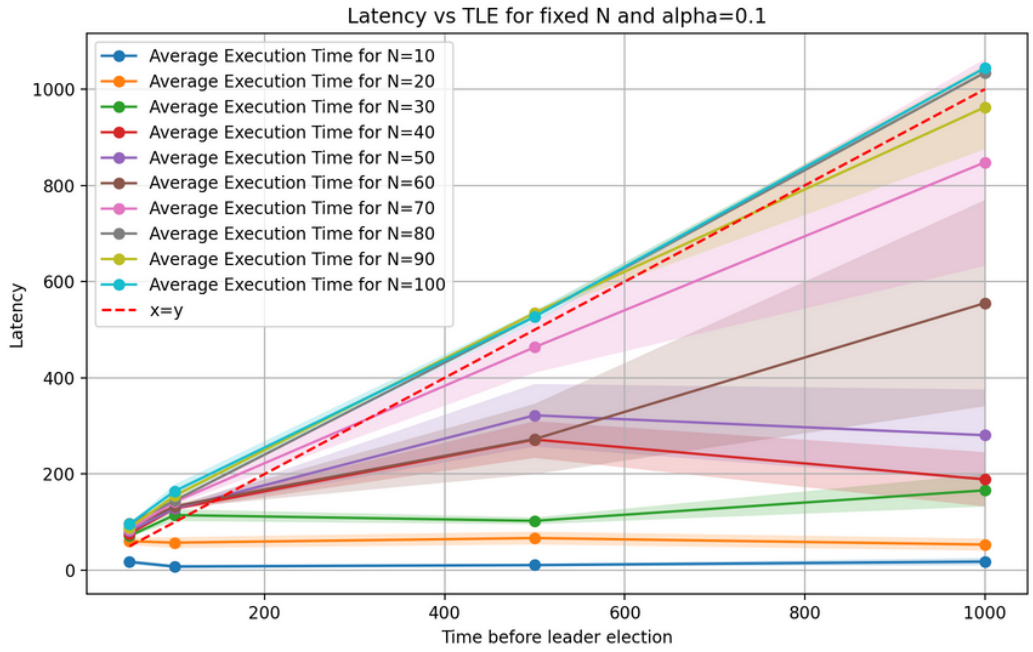

Figure 1

Global impact of TLE

The fist thing to notice in this performance analysis is the impact of TLE. Either they manage to decide before the leader is elected, in that case the execution time is lower than TLE (below the dotted red line x=y), which is the case for N=10-50 as seen on figure 1. Or the processes don't manage to decide before the leader, in that case a leader gets elected and shortly thereafter, he decides. This is the case for N=80,90,100. We can easily see on figure 1 that these 3 lines on or slightly above x=y meaning that they decide slightly after a leader is elected. Finally, we have the edge cases in between for n=60,70. Due to the randomness of the executions (since we have concurrent execution of 100 processes), sometimes they decide before TLE and sometimes after. This also leads to our data having a very large standard deviation in these cases as can be seen on figure 1.

The influence of TLE is so important that if it is low enough, the number of processes doesn't matter. They can't decide before TLE so they decide just after TLE. This can easily be seen on on figure 2, the first row is the same color for all N. Conversely, if the number of processes is high enough, all that matters is TLE, for N=80,90,100, the columns are basically identical. The execution time is directly corrolated to TLE. Like said previously, in the middle, we have a more erratic performance because the average execution time is right below TLE, this means that the processes will sometimes decide without a leader (<TLE), and sometime they will need the leader and decide in TLE + ~100-200ms. This can be best seen for N=40,50,60 and TLE = 500,1000ms

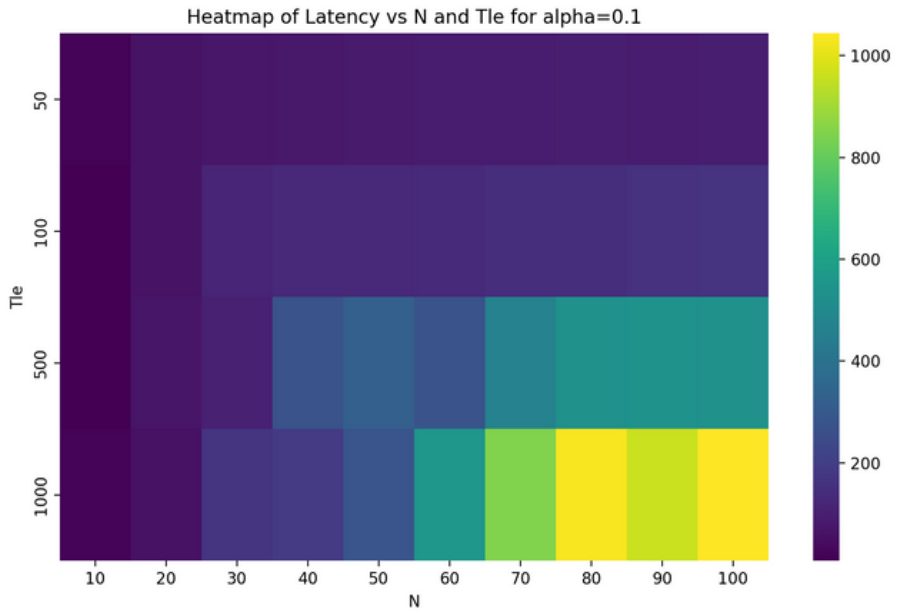

Figure 2

Impact of α

For the first 2 graphs, we had α set to 0,1. But does α influence the outcome? First off, we can see that as soon as α!=0, The outcome is the same if the other variable a re fixed. We can see on figure 3 that as soon as α!= the execution stays the same no matter the value α takes (identical columns). This can be explained by the fact that the faulty processes have α chance of crashing each time they process an event. However, the number of steps to reach an agreement is very high compered to alpha. In compraison, they will all crash during the begning. Hence the system rapidly becoms similar to the setting where alpha = 1.

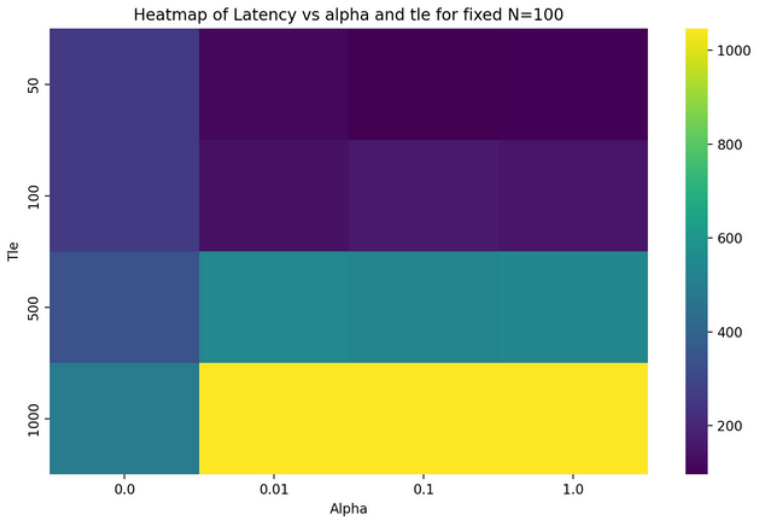

Figure 3

α = 0

When α=0, the executions times have a high standard deviation. The increased number of processes that are active means that there is more of a chance that a given process gets its proposition accepted by a majority, this process can therefore decide before TLE. The execution times vary a lot with an average at ~500ms and are capped by TLE. For TLE=500,1000 a few executions finished at TLE and most before this gets us an average below the TLE and below what we get for α>0. For TLE = 50,100, something else happens, the performance is worse for α=0. At first this doesn't make sense, why would we we have more performance when less processes fail. However this is more a result of the limitations of AKKA, with no faulty processes, exponentially more messages are exchanged and it simply takes a lot of time to process them. Therefore it takes more time than with processes that crash.

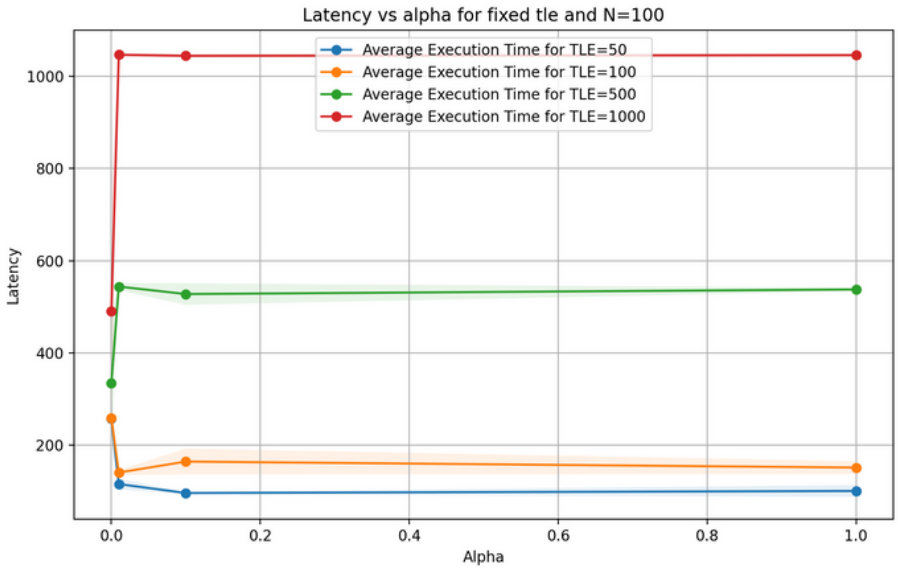

Figure 4

On figure 5, the effect of α is even more pronounced and easily observable thanks to TLE = 100ms. For α !=0 (the last 3 columns), the behavior is the same no matter N, except for for very low values of N, where the decision is made before TLE.
For α = 0, as we said before for a given number of processes, they can make, on average, a decision in a certain amount of time (above it was ~500ms for N=100) without a leader. This average amount of time depends solely on the number of processes. If we have a large amount of processes, enough to flood the system with message, the performance for α=0 will be worse than for α>0. Here it is the case for n>50. If we have a low enough amount of processes, the decision is done before TLE (N=10,20). In between, we have execution that will reach TLE and then the rest of the execution time depends of the number of messages that need to be processed before the leader election message.

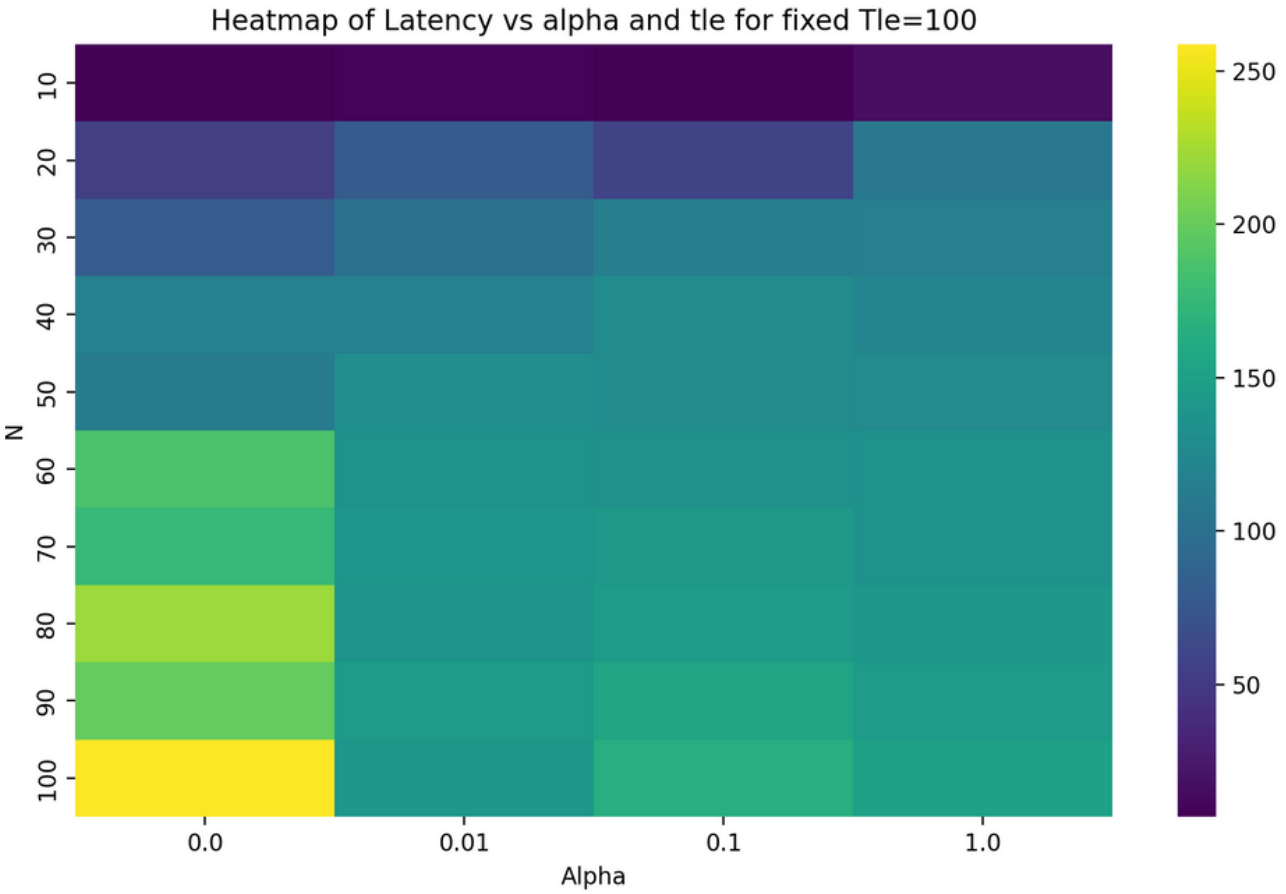

Figure 5

Standard deviation

As a last remark, we would like to point out if the processes manage to decide before TLE, the executions times can be all over the place. We can that easily on figure 6. The standard deviation for TLE = 500 and 1000 is quite large for N=50,60. The execution times are unpredictable. As soon as the decision is made with the leader, and thus linked to TLE, the standard deviation shrinks significantly as seen on the green line for N>=80.

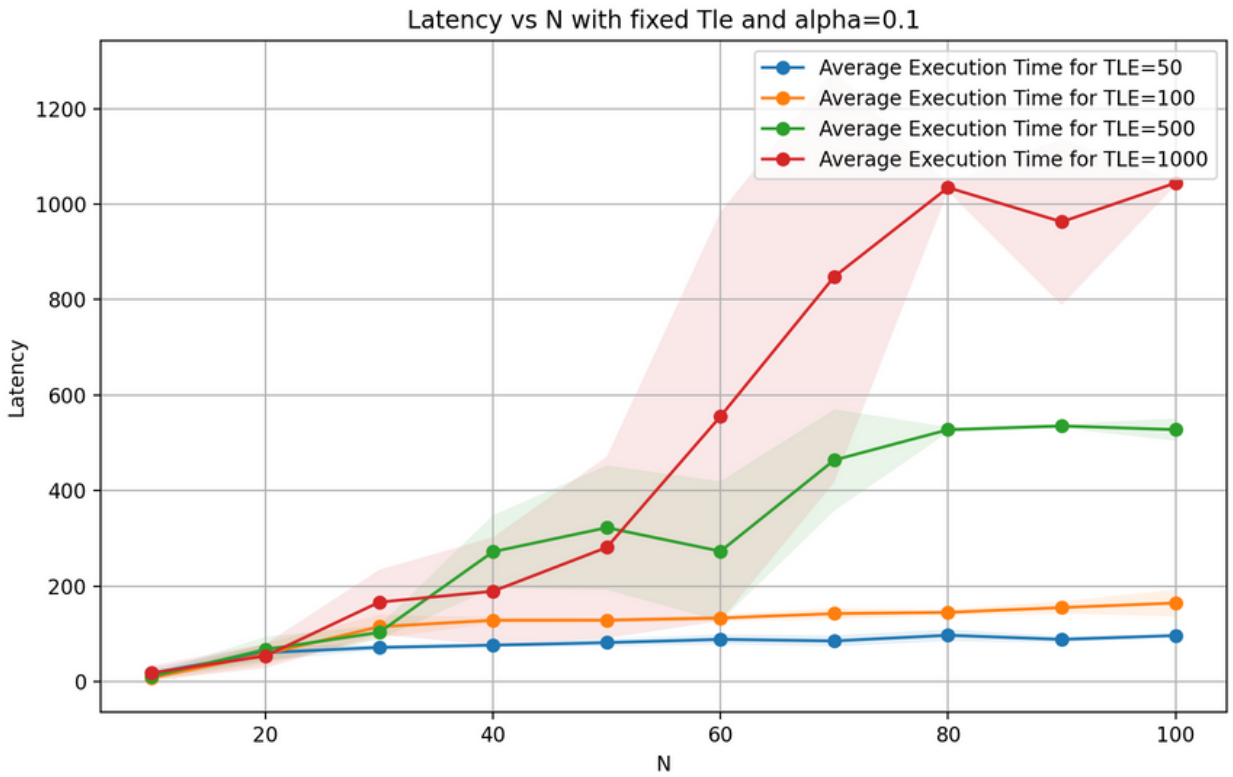

Figure 6

Conclusion

In a real world setting, we have to assume that alpha > 0. We cannot consider that processes may never crash. We can conclude different things from this analysis:

- if N is large, without oracle, the execution time scales with the number of processes and we have no guarantees in a suitable execution time. Therefore, we need to make the system eventually synchronous in order to implement an oracle in the system in order to get a bounded execution time.
- If N is small enough, we could consider a totally asynchronous system because the probability of not deciding in a satisfactory amount of time is negligible.