SLR210 - Blockchain: bases algorithmiques

Obstruction-Free Consensus and Paxos

KANG Jiale
ZHAO Yuanjie

April 3, 2024

Contents

1 Statement of the problem **3**

2 Description of implementation **3**

3 Performance analysis **8**

List of Tables

1 Possible Values for Some Parameters 8

List of Figures

1 Consensus Latency with $\alpha = 0$ 8
2 Consensus Latency with $\alpha = 0.1$ 9
3 Consensus Latency with $\alpha = 0.5$ 9
4 Consensus Latency with $\alpha = 1$ 10

1 Statement of the problem

In this project, our goal is to design a fault-tolerant distributed system. More concretely, we try to simulate an easy version of the Synod algorithm in Paxos protocol which is obstruction-free.

Let us first have a revision of what a obstruction-free consensus (OFC) algorithm is.

In OFC algorithm, we must ensure the following properties:

- **Validity:**
  - Every decided value is a proposed value.
- **Agreement:**
  - No two processes decide differently.
- **Obstruction-free termination:**
  - If a correct process proposes, it eventually decides or aborts.
  - If a correct process decides, no correct process aborts infinitely often.
  - If there is a time after which exactly one correct process p proposes a value sufficiently many times, p eventually decides.

The Synod algorithm implements obstruction-free consensus. It exports one operation *propose(v)* with an input value in a set $v \in V$. In this problem, we set $V = \{0, 1\}$. When a process invokes *propose(v)*, the *propose* operation returns either a value $v' \in V$, where we say that the process decides v'; or it receives *abort*, where we say that the process aborts. The abstraction is long-lived: one process can invoke the *propose* operation multiple times.

2 Description of implementation

Our implementation is based on the original Synod algorithm, but we've changed a little bit. We'll describe our implementation below.

For each process p_i, we have several local variables: *proposal* with initial value *nil*, which records the content of the proposal; *ballot* with initial value $i - n$, which records the unique index of the proposal; *readballot* with initial value 0, which records the biggest ballot number seen when receive $READ$ message; *imposeballot* with initial value $i - n$, which records the biggest ballot number seen when receive $IMPOSE$ message; *estimate* with initial value 0, which records the estimated content of what is going to be decided; an array *states* with initial value $[nil, 0]^n$, which records the gather information of other processes: the first value of each tuple is the estimated proposal while the second one is the estimated ballot.

When a process invokes $propose(v)$, obviously it will first set *proposal* to v. Then we add the *ballot* value with n because there are n processes in total and we want to make the ballot number unique with each proposal. We initialize the *state* array with $[nil, 0]^n$, then send $[READ, ballot]$ to all processes.

propose(v)

```
    void propose (int v) {
        if (crashed) return;
        if (shouldCrash) {
            double r = Math.random();
            if (r < CRASH_PROBABILITY) {
                crashed = true;
            }
        }
        if (!crashed) {
            proposal = v;
            ballot += N;
            for (int i = 0; i < N; i++) {
                states[i].first = 0;
                states[i].second = 0;
            }
            ReadMessage readMessage = new ReadMessage(ballot);
            for (int i = 0; i < N; i++) {
                actors[i].tell(readMessage, getSelf());
            }
        }
    }
```

When a process p_i receives $[READ, ballot']$ from another process p_j, it will first check whether the received $READ$ has ballots bigger than the ones recorded, for both the *readballot* and the *imposeballot*. If it finds that it's not the newest, it will send back an $[ABORT, ballot']$ message. Otherwise, it will update the *readballot* because it's now the read phase. Then it sends a gather message $[GATHER, ballot', imposeballot, estimate]$ to p_j for the acceptance of the ballot number, then p_j will gather the estimated information of p_i.

upon receive READ

```java
    public void receiveReadMessage (ReadMessage m) {
        if (crashed) return;
        if (proposeResult >= 0) return;
        if (shouldCrash) {
            double r = Math.random();
            if (r < CRASH_PROBABILITY) {
                crashed = true;
            }
        }
        if (!crashed) {
            if (readBallot > m.ballot || imposeBallot > m.ballot) {
                getSender().tell(new AbortMessage(m.ballot), getSelf());
            }
            else {
                readBallot = m.ballot;
                getSender().tell(new GatherMessage(m.ballot, imposeBallot,
                    estimate), getSelf());
            }
        }
    }
```

We will then talk about the situation of receiving $ABORT$ and $GATHER$ respectively.

When a process p_i receives $ABORT$, it means that it has sent a message with a ballot to another process p_j, but p_j has newer information then the sent message. In this case, it's time for p_i to re-propose a message with a newer ballot. Seeing that it's highly possible that p_i will receive many $ABORT$ messages with the same ballot, we certainly do not want p_i to re-propose each time when it sees an $ABORT$ message. The $ABORT$s with the same ballot should only produce one re-propose operation. So we keep track of the biggest ballot received with the $ABORT$ message. If one $ABORT$ message is received, but its ballot is not bigger than the biggest ballot which we store, we do nothing because this message is kind of outdated. Otherwise, we update the biggest ballot value. When doing this, we will reduce significant time complexity so that the number of proposals does not grow exponentially.

upon receive ABORT

```java
    public void receiveAbortMessage (AbortMessage m) {
        if (crashed) return;
        if (shouldCrash) {
            double r = Math.random();
            if (r < CRASH_PROBABILITY) {
                crashed = true;
            }
        }
        if (!crashed) {
            proposeResult = -1;
            if (!hold) {
                if (!decided && m.ballot > maxAbortBallot) {
                    maxAbortBallot = m.ballot;
                    propose(proposal);
                }
            }
        }
    }
```

When a process p_i receives $GATHER$ from p_j, it will update the decision of p_j: p_j's *imposeballot* as the estimated ballot and p_j's *estimate* as the content of the estimated proposal. After the update, we will check that whether this update makes the number of tracked states has reached the majority, which means that there exists one quorum in which all processes has sent messages to p_i. In this case, we'll check that whether some process has already taken a non-initial proposal. If so, we should take the one with the maximal ballot (we note it p_k) as the estimated proposal: we set *proposal* to the estimated proposal of p_k recorded in the array *states*. Otherwise, p_i let other processes take its own proposal. In both cases, we will reset the *states* array of p_i and then send $[IMPOSE, ballot, proposal]$ to all processes: p_i now imposes the proposal.

upon receive GATHER

```java
    public void receiveGatherMessage (GatherMessage m) {
        if (crashed) return;
        if (proposeResult >= 0) return;
        if (shouldCrash) {
            double r = Math.random();
            if (r < CRASH_PROBABILITY) {
                crashed = true;
            }
        }
        if (!crashed) {
            states[(m.ballot + N) % N].first = m.estimate;
            states[(m.ballot + N) % N].second = m.imposeBallot;
            receivedStates++;
            if (receivedStates > N/2 && !biggerThanHalf) {
                biggerThanHalf = true;
                int maxidx = -1;
                for (int i = 0; i < N; i++) {
                    if (states[i].second > 0) {
                        if (maxidx == -1 || states[i].second > states[maxidx].
                            second) {
                            maxidx = i;
                        }
                    }
                }
                if (maxidx != -1) proposal = states[maxidx].first;
                for (int i = 0; i < N; i++) {
                    states[i].first = 0;
                    states[i].second = 0;
                }
                receivedStates = 0;
                biggerThanHalf = false;
                for (int i = 0; i < N; i++) {
                    actors[i].tell(new ImposeMessage(ballot, proposal), getSelf
                        ());
                }
            }
        }
    }
```

When a process p_i receives $IMPOSE$ of ballot $ballot'$ from p_j, just like what it does when it receives $READ$, it will first check that whether this message has the newest ballot. If not, it sends back $ABORT$. Otherwise, p_i set its estimated proposal *estimate* to the one p_j has sent, and update the *imposeballot*. After that, p_i should send acknowledge message. It sends $[ACK, ballot']$ to p_j.

upon receive IMPOSE

```java
    public void receiveImposeMessage (ImposeMessage m) {
        if (crashed) return;
        if (proposeResult >= 0) return;
        if (shouldCrash) {
            double r = Math.random();
            if (r < CRASH_PROBABILITY) {
                crashed = true;
            }
        }
        if (!crashed) {
            if (readBallot > m.ballot || imposeBallot > m.ballot) {
                getSender().tell(new AbortMessage(m.ballot), getSelf());
            }
            else {
                estimate = m.proposal;
                imposeBallot = m.ballot;
                getSender().tell(new ACKMessage(m.ballot), getSelf());
            }
        }
    }
```

In each process, we also keep track of the number of ACKs it receives. When p_i receives one ACK, we increase the ACK counter by one. When it reaches the majority, it means that one quorum has reached a consensus. So it's now to decide on this proposal. It decides, and sends $[DECIDE, proposal]$ to other processes to tell others about this news.

upon receive ACK

```java
    public void receiveACKMessage (ACKMessage m) {
        if (crashed) return;
        if (proposeResult >= 0) return;
        if (shouldCrash) {
            double r = Math.random();
            if (r < CRASH_PROBABILITY) {
                crashed = true;
            }
        }
        if (!crashed) {
            ACKnum++;
            if (ACKnum > N/2 && !ACKconfirmed) {
                ACKconfirmed = true;
                endTime = System.currentTimeMillis();
                decided = true;
                for (int i = 0; i < N; i++) {
                    actors[i].tell(new DecideMessage(proposal), getSelf());
                }
            }
        }
    }
```

Finally, when one process receives $DECIDE$ message with proposal v, it simply takes v as its decided proposal.

upon receive DECIDE

```java
    public void receiveDecideMessage (DecideMessage m) {
        if (crashed) return;
        if (shouldCrash) {
            double r = Math.random();
            if (r < CRASH_PROBABILITY) {
                crashed = true;
            }
        }
        if (!crashed) {
            proposeResult = m.proposal;
            this.decided = true;
        }
    }
```

The above description is about the Synod algorithm that we implemented. Then we'll talk about the overall implementation of the project.

The project uses AKKA for message passing. For each type of message, we create a message class and create a listener in the process class for each message type.

Our goal for this project is to simulate a system of N processes out of which up to f can fail ($f < N/2$).

For every process, the `main` method sends a *LAUNCH* message to it. After receiving this message, this process will generate a random value in $\{0, 1\}$ and start to propose this value. This means that it will invoke the *propose* operation with this value until a value is decided.

The `main` method will also select f processes randomly to be faulty processes and send each of them a *CRASH* message. If a process receives such message, every time it takes action, it has a fixed possibility α to be crashed. If it crashes, it becomes silent and will never respond to any event from this time on.

the main method

```java
        LaunchMessage launchMessage = new LaunchMessage();
        for (int i = 0; i < N; i++) {
            actors[i].tell(launchMessage, ActorRef.noSender());
        }

        List<Integer> crashList = new ArrayList<>();
        for (int i = 0; i < N;i++) {
            crashList.add(i);
        }
        Collections.shuffle(crashList);

        // Send CRASH messages to the actors in the crash list
        for (int i = 0; i < CRASH_NUMBER; i++) {
            actors[crashList.get(i)].tell(new CrashMessage(), ActorRef.noSender
                ());
        }

        // The list is shuffled, so we can just set the leader with the first
            actor not in the crash list

        // We now send HOLD message to every other process
        for (int i = 0; i < N; i++) {
            if (i != CRASH_NUMBER) {
                system.scheduler().scheduleOnce(Duration.create(
                    LEADER_ELECTION_TIMEOUT, TimeUnit.MILLISECONDS), actors[
                    crashList.get(i)], new HoldMessage(), system.dispatcher(),
                    null);
            }
        }
```

In this project we also emulate a leader election mechanism. After a fixed time t_le, the `main` method will randomly pick a non-faulty process to be the leader. It will send a *HOLD* message to all non-leader processes. If a process receives *HOLD* messages, it will set its hold flag to `true` and stops invoking *propose* even when it receives *ABORT*. To realize this mechanism, we use the scheduler to send the message.

3 Performance analysis

In order to find the relationship between parameters (*system size* or *process number* N, *faulty probability* α and *consensus latency* t_{le}), we set the different values as Table 1 shows below.

Table 1: Possible Values for Some Parameters

Parameters	Possible Values				
N	3	10	50	100	
f	1	4	24	49	
α	0	0.1	0.5	1	
t_{le}	10	50	100	500	1000

For each combination of the parameters, we repeated 10 times and calculated the average values to increase the precision of the experiment.

First, we set t_{le} as x-axis and *consensus latency* as y-axis, each color of curve represents the number of process N. The results shows as Figure 1 - 4.

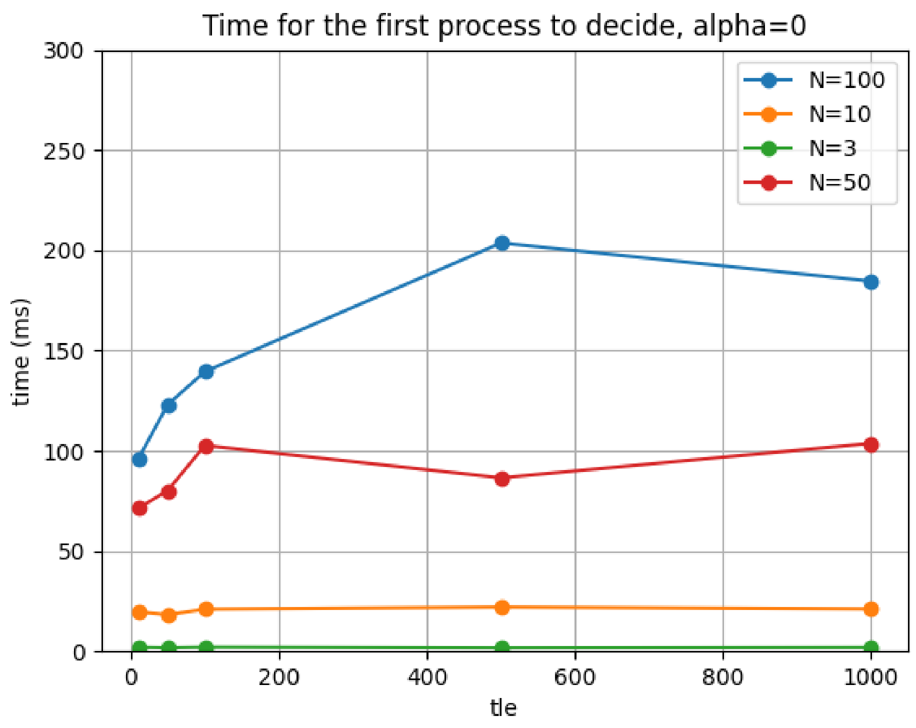

Figure 1: Consensus Latency with $\alpha = 0$

In an ideal scenario with no crash probability ($\alpha = 0$) as Figure 1, the consensus time for $N = 100$ shows a trend of increasing with the rise in t_{le}. For $N = 50$, $N = 10$ and $N = 3$, the consensus time is relatively flat, showing behavior unrelated to increases in t_{le}. This may indicate that without process crashes, the increase in the number of leader elections primarily affects the consensus time in larger systems, while smaller systems are unaffected.

For the case $\alpha = 0.1$ in Figure 2, when t_{le} keeps in low level, all the curves have insignificant changes. Until t_{le} comes to 500 ms, all the time of decision reach the highest point. Additionally, for $N = 50$, $N = 10$ and $N = 3$, the consensus time remains relatively stable. We suppose that when facing a lower crash probability, larger systems may be more sensitive in terms of consensus time, especially in environments with frequent leader elections.

Keep increasing crash probability α to $\alpha = 0.5$ as Figure 3, no significant change in the overall tendency. But there is a slight decrease for the consensus time when $N = 50$ and $t_{le} = 500$ ms. We are of the opinion that the crash probability is not the primary factor to affect consensus time.

With the crash probability at its maximum, $\alpha = 1$, in Figure 4, we see that the consensus time for $N = 100$ is still in the highest position. While for small system, the consensus time remains unchanged, irrespective of

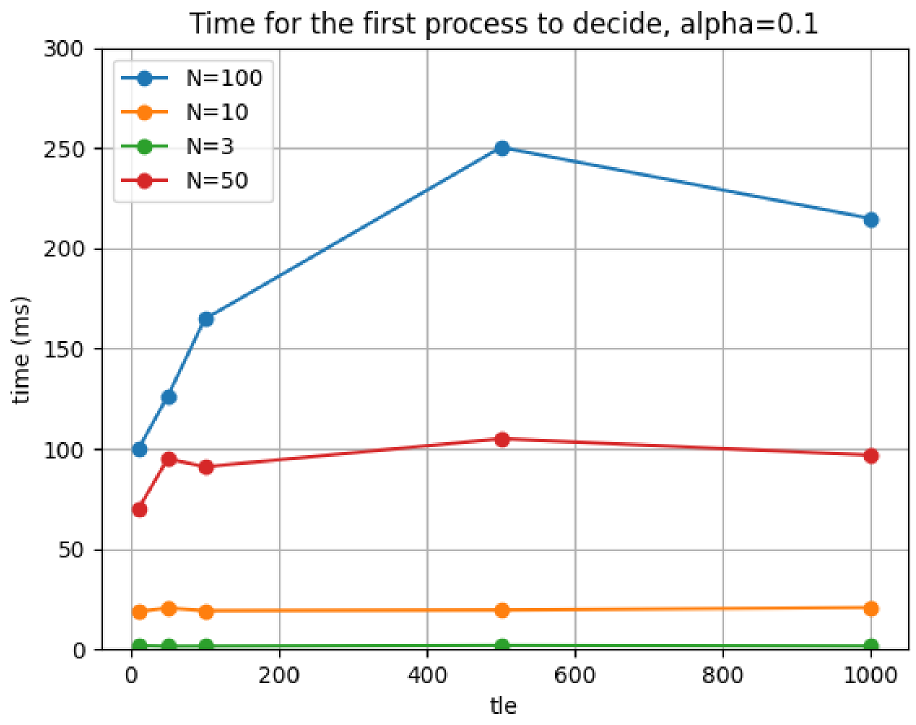

Figure 2: Consensus Latency with $\alpha = 0.1$

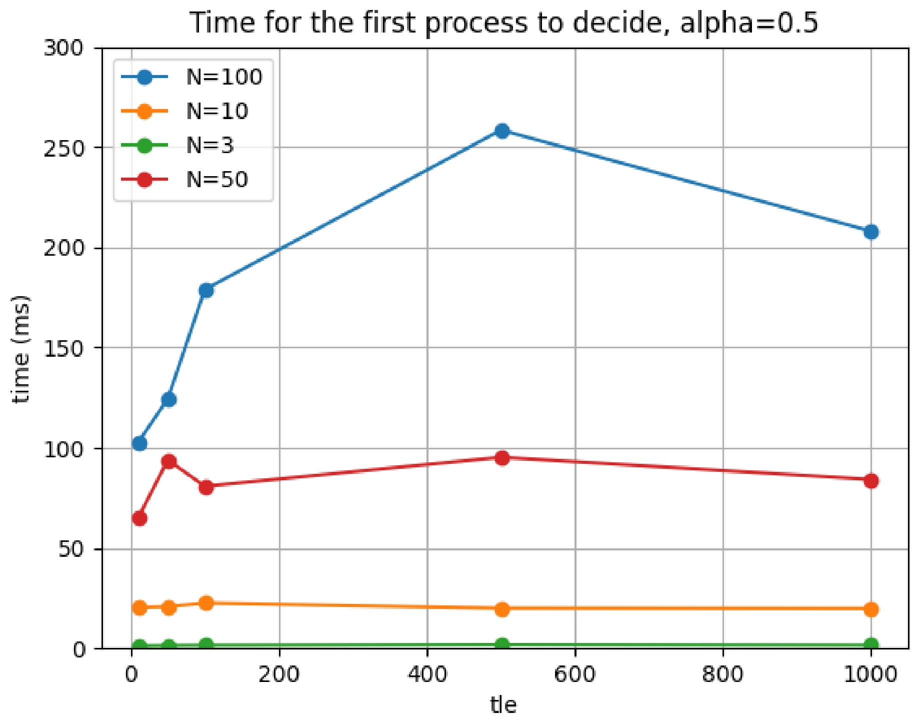

Figure 3: Consensus Latency with $\alpha = 0.5$

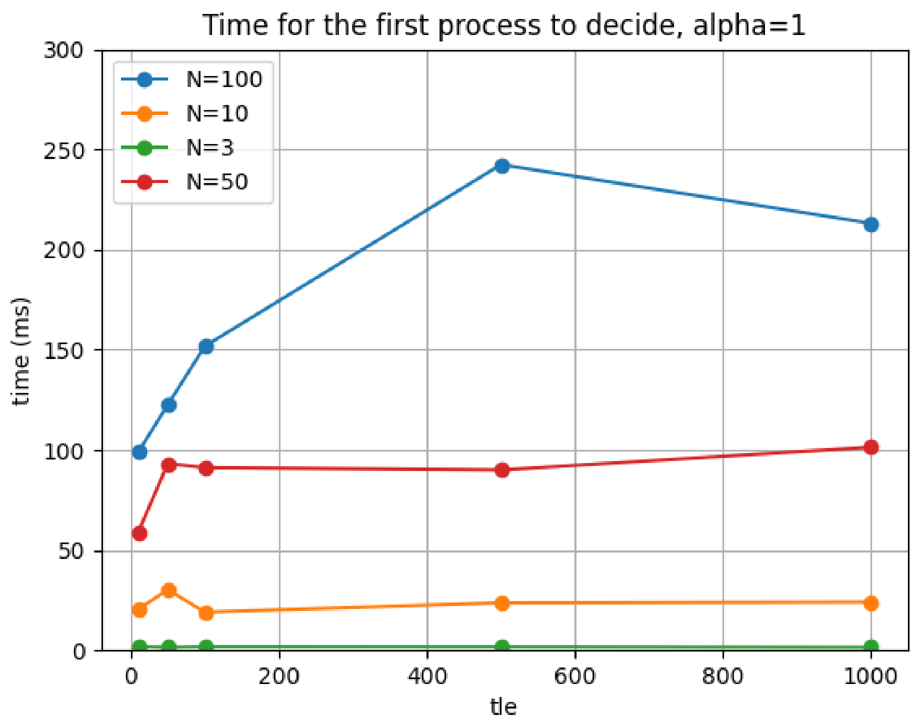

Figure 4: Consensus Latency with $\alpha = 1$

t_{le}. Interestingly, despite the highest crash probability, the consensus time does not show the significant growth seen in the previous graph (Figure 3), which might suggest that other factors may help stabilize consensus time in highly failure-prone systems.

In conclusion, there is a clear trend that:

- Larger systems ($N = 100$) use more time to complete consensus;
- Smaller systems (especially for $N = 3$ and $N = 10$) show lower and stable consensus times, indicating that they are less sensitive to the number of leader elections and crash probabilities;
- Larger systems are more sensitive to t_{le} but still keeps stable for the variations of α;
- High crash probabilities do not always lead to a significant increase in consensus time.