BFT&Blockchains 2023
Project: Obstruction-Free Consensus and Paxos

The goal of this project is to get an initial experience in designing a fault-tolerant distributed system. Here we focus on a state-machine replicated system build atop a consensus abstraction.

1 Specification

An obstruction-free consensus (OFC) algorithm exports one operation *propose*(v) with an input value in a set $v \in V = \{0, 1\}$. When a process invokes *propose*(v), we say that the process *proposes* v. The operation returns either a value $v' \in V$ (in which case we say that the process *decides* v') or a special value *abort* $\notin V$ (in which case we say that the invocation *aborts*). A process can invoke the *propose* operation multiple times.

The following properties must be met:

- Validity: every decided value is a proposed value.
- Agreeement: 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.

2 Concurrent environment

The goal of the project is to implement OFC for the following environment:

- We have N asynchronous processes. Every process has a distinct *identifier*. The identifiers are publicly known.
- Every two processes can communicate via a reliable asynchronous point-to-point channel.
- Up to $f < N/2$ of the processes are subject to crash failures: a faulty process prematurely stops taking steps of its algorithm. A process that never crashes is called correct.

3 Prerequisites

The project assumes a basic knowledge of Java. Get familiarized with the Java version of AKKA, an actor-based programming model `https://akka.io/docs/`. Check basic constructions in to see how to create an actor, and make the actors communicate.

4 Implementation

The implementation should extend the basic construction creating a system of a given size and ensure all-to-all connectivity. Create N actors (processes), and pass references of all N processes to each of them.

Use the name `Process` for the process class. For the *Process* class, create methods for invoking the operation *propose*, processing received messages, and returning response indications.

Our goal is to simulate runs of the algorithm in a system of N processes out of which up to f can fail. You can follow the following procedure.

For every process, the `main` method then sends a special *launch* message. Once process i receives a launch message, it picks an input value, randomly chosen in $\{0, 1\}$ and invokes instances of *propose* operation with this value until a value is decided. As a basis, one can use the OFC pseudocode discussed in the lecture (adjusted to be used within AKKA).

The `main` method also selects f processes at random (e.g., using the shuffle method from `java.collections`) and sends each of them a special *crash* message. If a process receives a crash message it enters the *fault-prone* mode: for any processed *event* in the algorithm, the process decides, with a fixed probability α, if it going to *crash*. If it crashes, it enters the *silent* mode, not reacting to any future event.

Use the `LoggingAdapter` class to log both the timing of the invocation and the response of every operation each process performs.

- Emulate a leader election mechanism: after a fixed timeout t_{le}, the `main` method randomly picks up a process that is not *fault-prone* and sends a *hold* message to every other process. After receiving a *hold* message, a process stops invoking *propose* operations.

  For example, by invoking `Thread.sleep(50)`, the `main` method "freezes" for 5ms.

  An alternative method consists in using the scheduler. For example, the following command:

  ```
  system.scheduler().scheduleOnce(Duration.create(50, TimeUnit.MILLISECONDS),
  testActor, "foo", system.dispatcher(), null);
  ```

  results in a message ``foo'' sent by the scheduler to `testActor` in 50ms.

- Perform the experiment for different system sizes N and leader election times t_{le}.

  For each configuration, measure the *consensus latency*: how long it takes for the first process decides.

  Your goal is to find out how the latency depends on N (for a fixed t_{le}) and t_{le} (for a fixed N).

  Also try to see if the probability of failures α (for fixed N and t_{le}) affects the latency.

- As a baseline, you can consider $N = 3, 10, 100$ (with $f = 1, 4, 49$, respectively), $t_{le} = 0.5s, 1s, 1.5s, 2s$, $\alpha = 0, 0.1, 1$.

Each experiments should be repeated 5 times and the average latency should be evaluated.

5 Report

Prepare a short report (up to 15 pages). The report should contain:

- The statement of the problem that your implementation solves;
- A high level description of the implementation;
- A report on performance analysis, with plots relating the latency with N, t_{le} and α.

The report should appear comprehensible for a non-expert in consensus protocols. Try to be pedagogical, imagine your peers as the audience. The grades depend on the clarity and completeness.

The report and the resulting code should be submitted by **15/06/23 (to be agreed on)**.