EFREI M1 Big Data: Project Description
Robust Key-Value Store

The goal of this project is to get an initial experience in designing a fault-tolerant distributed system. Here we focus on the popular key-value store application.

1 Sequential specification

The state of a key-value store is a set of key-value pairs of the form (k, v), where k is an integer and v is a value in a given *value set* (assume that values are also integers). The initial state is an empty set. The system exports two operations:

- $put(k, v)$ sets the value with key k to v (overwriting the old value if it is already in the set).
- $get(k)$ returns the value of key k (the default value $\perp$ is returned is the key is not at the system).

2 Concurrent environment

The goal of the project is to give a key-value store implementation 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.

The implemenation should ensure that in its every execution, the following conditionas are met:

Safety The corresponding history is linearizable with respect to the sequential specification above;

Liveness Every operation invoked by a correct process eventually returns.

Hint: You can treat the system as a set of multi-writer multi-reader atomic registers (indexed by keys). Just as in the ABD algoriuthm (check class04), for each key k, every process maintains a local copy of the value, equipped with a *timestamp.* To write a new value v, we need to make sure that at least a majority $(N - f)$ of the processes store v or a newer value in their local copies. To get a value, we need to contact a a majority $(N - f)$ of the processes and return the most recent value they are aware of.

Of course, the ABD algorithm only implements a *single-writer regular* register, while we want to implement a *multi-writer atomic* one. So the writer (the process executing a *put* operation) needs to be careful to choose a timestamp for the message it writes to be higher than the timestamps of preceding writes. Respectively, to prevent the new-old inversion, the reader (the process executing a *get* operation) needs to ensure that the returned value (or a newer one) is stored at a majority of the processes.

This might require one extra round of message exchange between the process performing an operation and a *quorum* (a majority) of other processes.

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/`. heck basic constructions in to see how to create an actor, and make the actors communicate.

Check `https://github.com/remisharrock/SLR210Patterns` for sample AKKA patterns which you might want to use.

4 Formalities

The project is pursued in teams of two or three students each.

The implemented system should be provided with a short report describing how the system operates and containing correctness arguments. The team should also prepare a short presentation to be given at the end of the course.

The first project meetings on October 23-25 contain a tutorial on the AKKA programming environment. The two subsequent meetings (on 5-6/11 and 26/11) will be used for discussing potential issues and problems. The final meetingson December 10-11 will be used for project presentations.

5 Implementation

The implementation should extend the basic construction creating a system of a given size and ensure all-to-all connectivity (the homework of November 7). More precisely, in the main class, create N actors (processes), and pass references of all N processes to each of them. Use the name `Process` for the process class.

In the `Process` class create methods for executing operations *put* and *get*. **For simplicity, implement just the system for just a single key. Recall that this is equivalent to implementing a single atomic register.**

To test the implementation and measure its performance, use the following procedure.

The `main` method 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 *silent* mode, not reacting to any future event.

For every remaining process, the `main` method sends a special *launch* message. Once process i receives a *launch* message, it sequentially performs:

- M *put* operations, with parameters $k = 1$ and $v = i, 2 * i, \ldots, M * i$, and
- M *get* operations, with parameters $k = 1$.

Make sure that every process performs at most one operation at a time (remember that we require every exported history to be *well-formed*).

Use the `LoggingAdapter` class to log both the invocation and the response of each operation a process performs together with it timing.

Perform the experiment for $N = 3, 10, 100$ (with $f = 1$, 4, and 49, respectively) and $M = 3, 10, 100$ (nine instances). For the instance $N = 3$ and $M = 3$, check that the resulting execution is linearizable. Also, for each instance, measure the *latency*, i.e., the total computation time.

6 Report

Prepare a short report (up to 15 pages), preferably in English (can also be written in French if English does not feel comfortable). The report should contain:

- A high level description of the system;
- A pseudocode of the implementation;
- A sketch of a proof of correctness (please argue that both safety and liveness hold);

- A report on performance analysis.

The report and the code of the implementation should be submitted by **December 6** via the Moodle service (archived in one zip file).

7 Presentation

The presentation (7 mins) should contain a very brief overview of the main features of the algorithm, its correctness arguments and performance. We envision 10 minutes per team (including 3 minutes for questions), so the time bounds are strict.