Politecnico di Milano

090950 – Distributed Systems

Prof. G. Cugola and A. Margara

Projects for the A.Y. 2020-2021

Rules

1. The project is optional and, if correctly developed, contributes by increasing the final score.
2. Projects must be developed in groups composed of a minimum of two and a maximum of three students.
3. The set of projects described below are valid for this academic year, only. This means that they have to be presented before the last official exam session of this academic year.
4. Students are expected to demonstrate their projects using their own notebooks (at least two) connected in a LAN (wired or wireless) to show that everything works in a really distributed scenario.
5. To present their work, students are expected to use a few slides describing the software and run-time architecture of their solution.
6. Students interested in doing their thesis in the area of distributed systems should contact Prof. Cugola for research projects that will substitute the course project.

Replicated, high-performance key-value store

Implement a byzantine fault tolerant replicated key-value store, following the approach suggested in http://www.pmg.lcs.mit.edu/papers/osdi99.pdf

Requirements:

The store consists of N nodes, each holding a copy of all keys and related values. The store offers two functionalities to clients:

- put(k, v) inserts/updates value v for key k,
- get(k) returns the value associated to key k (or null of the key is not present)

Clients can interact with the store by contacting any of its nodes; nodes internally coordinate to offer the service.

The project can be implemented as a real distributed application (for example, in Java) or it can be simulated using OmNet++. In the first case you are allowed to use only basic communication failities (i.e., sokects and RMI, in case of Java).

Assumptions:

Assume unreliable (byzantine) processes and reliable (TCP) links, see the paper for details.

Java library for distributed snapshot

Implement a library that offers the capability of storing a distributed snapshot on disk. The library should be state and message agnostic.

Implement an application that uses the library to cope with node failures (restarting from the last valid snapshot).

Assumptions:

- Nodes do not crash in the middle of the snapshot.
- The topology of the network (including the set of nodes) does not change during a snapshot.
- Multiple snapshots may run in parallel.

Replicated data store

Implement a replicated key-value store that offers causal consistency.

Requirements

- Implement causal consistency with limited (coordination) overhead.
- New replicas can be added or removed at runtime.
- The store can be implemented as a real distributed application (for example, in Java) together with some client code to use / test the implementation, or it can be simulated using OmNet++. In the first case you are allowed to use only basic communication facilities (i.e., sokects and RMI, in case of Java).

Assumptions

- Processes are reliable.

- Channels are point-to-point (no broadcast) and you may assume the same fault model of the Internet (congestions or partition).
- Clients are "sticky": they always interact with the same replica.

Distributed job scheduling

Implement an infrastructure to manage jobs submitted to a cluster of Executors. Each client may submit a job to any of the executors receiving a job id as a return value. Through such job id, clients may check (contacting the same executor they submitted the job to) if the job has been executed and may retrieve back the results produced by the job.

Executors communicate and coordinate among themselves in order to share load such that at each time every Executor is running the same number of jobs (or a number as close as possible to that). Assume links are reliable but processes (i.e., Executors) may fail (and resume back, re-joining the system immediately after).

Choose the strategy you find more appropriate to organize communication and coordination. Use stable storage to cope with failures of Executors.

Implement the system in Java (or any other language you choose) only using basic communication facilities (i.e., sockets and RMI, in case of Java). Alternatively, implement the system in OMNeT++, using an appropriate, abstract model for the system (including the jobs themselves).