Politecnico di Milano

090950 – Distributed Systems

Prof. G. Cugola and A. Margara

Projects for the A.Y. 2022-2023

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.

Quorum-based replicated datastore

You are to implement a distributed key-value store that accepts two operations from clients:

- put(k, v) inserts/updates value v for key k;
- get(k) returns the value associated with key k (or null if the key is not present).

The store is internally replicated across N nodes (processes) and offers sequential consistency using a quorum-based protocol. The read and write quorums are configuration parameters.
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 facilities (that is, sockets and RMI in the case of Java).

Assumptions

- Processes and links are reliable (use TCP to approximate reliable links and assume no network partitions can occur);

Fault-tolerant dataflow platform

You are to implement a distributed dataflow platform for processing key-value pairs where keys and values are integers.
The platform offers three operators, which are executed independently and in parallel for each key k:

- map(f: int → int): for each input tuple <k, v>, it outputs a tuple <k, f(v)>
- changeKey(f: int → int): for each input tuple <k, v>, it outputs a tuple <f(v), v>
- reduce(f: list<int> → int): takes in input the list V of all values for key k, and outputs <k, f(V)>

The platform includes a coordinator and multiple workers.
The coordinator accepts dataflow programs specified as an arbitrarily long sequence of the above operators. For instance, programs may be defined in JSON format and may be submitted to the coordinator as input files. Each operator is executed in parallel over multiple partitions of the input data, where the number of partitions is specified as part of the dataflow program.
The coordinator assigns individual tasks to workers and starts the computation.
Implement fault-tolerance mechanisms that limit the amount of work that needs to be re-executed in the case a worker fails.
The project can be implemented as a real distributed application (for example, in Java) or it can be simulated using OmNet++.

Assumptions

- Workers may fail at any time, while we assume the coordinator to be reliable.
- Links are reliable (use TCP to approximate reliable links and assume no network partitions can occur).
- You may implement either a scheduling or a pipelining approach. In both cases, intermediate results are stored only in memory and may be lost if a worker fails. On the other end, nodes can rely on some durable store (the local file system or an external store) to implement fault-tolerance mechanisms.
- You may assume input data to be stored in one or more csv files, where each line represents a <k, v> tuple.

- You may assume that a set of predefined function exists and they can be referenced by name (for instance, function ADD(5) is the function that takes in input an integer x and returns integer x+5)

Reliable broadcast library

You must implement a library for reliable broadcast communication among a set of faulty processes, plus a simple application to test it (you are free to choose the application you prefer to highlight the characteristics of the library).
The library must guarantee virtual synchrony, while ordering should be at least fifo.
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 facilities (that is, sockets and RMI in the case of Java).

Assumptions

Assume (and leverage) a LAN scenario (i.e., link-layer broadcast is available).
You may also assume no processes fail during the time required for previous failures to be recovered.

Java library for distributed snapshot

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