Politecnico di Milano

090950 – Distributed Systems

Prof. G. Cugola and A. Margara

Projects for the A.Y. 2023-2024

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. Projects developed in Java cannot use networking technologies other than sockets (TCP or UDP, unicast or multicast) or RMI.
7. 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.

Fault-tolerant dataflow platform

Implement a distributed dataflow platform for processing large amount (big-data) of key-value pairs, where keys and values are integers.

The platform is capable of running programs composed of a combination of four operators:

- map(f: int → int): for each input tuple <k, v>, it outputs a tuple <k, f(v)>
- filter(f: int → boolean: for each input tuple <k, v>, it outputs the same tuple <k, v> if f(v) is true, otherwise it drops the tuple
- 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 a single tuple <k, f(V)>

The reduce operator, if present in a program, is always the last one.

The platform includes a coordinator and multiple workers running on multiple nodes of a distributed system.
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 guides the computation.
Decide the architecture of the application, the channels to use (distributed file system vs network channels) and the approach to processing (batch vs stream) to optimize performance, considering all the aspects that may impact the execution time, including the need to pass data around. Implement also 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.
- Network links, when present, are reliable (use TCP to approximate reliable links and assume no network partitions can occur). The same is true for the storage of each node, if present and used.
- 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 storage (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

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++.

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.

Reliable queuing system

Implement a distributed and reliable queuing platform where a set of brokers collaborate to offer multiple queues to multiple clients.
Queues are persistent, append only, FIFO data structures.
Multiple (not necessarily every) brokers replicate the data of queues to guarantee fault tolerance and high availability in case one of them crashes.
Clients connect to brokers to create new queues, append new data (for simplicity assume that queues store integer values) on an existing queue, or read data from a queue. Each client is uniquely identified and the brokers are responsible for keeping track of the next data element each client should read.
Investigate and clarify the level of reliability offered by your system.
The project can be implemented as a real distributed application (for example, in Java) or it can be simulated using OmNet++.

Assumptions

- You may assume links to be reliable while nodes (brokers) may fail (crash failures).
- Nodes running brokers holds a stable storage (the file system) that can be assumed to be realiable.

Highly available, causally ordered group chat

Implement a distributed group chat application. Users can create and delete rooms. For each room, the set of participants is specified at creation time and is never modified. Users can post new messages for a room they are participating to. Within each room, messages should be delivered in causal order.

The application should be fully distributed, meaning that user clients should exchange messages without relying on any centralized server.

The application should be highly available, meaning that users should be able to use the chat (read and write messages) even if they are temporarily disconnected from the network.

The project can be implemented as a real distributed application (for example, in Java) or it can be simulated using OmNet++.

Assumptions

- Clients and links are reliable, but clients can join and leave the network at any time.