Distributed Systems + Middleware Technologies for Distributed Systems

Year 2017-2018

Rules

1. Projects must be developed in groups composed of a minimum of two and a maximum of three students.
2. For the "Distributed systems" course the project is optional and, if correctly developed, contributes by increasing the final score. For the "Middleware technologies for distributed systems" course three projects have to be chosen.
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**

Distributed Systems

Distributed stream processing

Implement a stream processing system for in-memory processing of streaming elements. The stream processing system exposes primitives to build a directed acyclic graph of operators. Each operator processes incoming elements and forwards the results to the downstream operators using a "pipeline" approach.

A special "supervisor" process exists. The supervisor exposes the primitives to build the operator graph. After the graph is built, the supervisor "deploys" it by instantiating a different process for each operator. Operator processes can be launched on different machines (to improve performance).

The following assumptions hold:

- The supervisor is reliable (but operators may fail).
- Input elements consist of numbers.
- A single operator type exists, which computes an aggregate over a count-based window. The user can specify the type of aggregation to perform (average, sum, min, max), the size, and the slide of the window.
- Operators are connected through FIFO (TCP) links.
- The output of an operator can be connected to multiple downstream operators (split), which receive a copy of the data produced upstream. Similarly, an operator may consume data from multiple upstream operators (join).
- There is a single source of input elements and a single consumer of output elements.
- The source can replay the input elements from a specified point.

The system should implement fault tolerance (operators crashing). To do so, operators must periodically store their state into a reliable storage (you can assume it is the local disk and you can assume it does not fails). The supervisor monitors the state of the operators. If an operator fails, all operators must be brought back to the last reliable state and the source must be asked to replay all the elements from that state on.

Additional notes:

- Operators are not partitioned (there is no data parallelism, only task/operator parallelism).
- The system is not required to guarantee that each result is delivered to the consumer exactly once in presence of failures.

Implement the project in Java or simulate it in OmNet++.

Optional

To get additional points you may choose to enable one of the following extensions:

1. Consider input elements composed of key-value pairs and enable data parallelism.
2. Guarantee that each result is delivered to the consumer exactly once.

Replicated data storage

Implement a replicated data storage. For simplicity assume that the data to be replicated are integer numbers, each identified by a unique id. N servers keep a copy of the data shared by clients, offering two primitives:

- int read(dataId);
- void write(dataId, newValue);

The client may connect to any of these servers (a single one for the entire session). Servers cooperate to keep a consistent, replicated copy of the shared data. In particular, the system must provide a *sequential consistency model*.

The following assumptions hold:

- Servers are reliable and known to each other.
- Severs run on the same subnet (IP multicast is available among them).
- Channels are unreliable.

Suggestion: may use a totally ordered multicast primitive (possibly built on top of IP multicast) to guarantee the consistency requirements.

Implement the project in Java or simulate it in OmNet++.

Optional

Relax the assumptions about servers being known and reliable.

Middleware Technologies for Distributed Systems