Université Paris Sorbonne Nord
Institut Galilée

Master 1.
Distributed Java Programming
Project

For this project, you are expected to apply the techniques about concurrent and distributed systems covered in the course, including distributed resource management and synchronization and mutual exclusion across processes. Working responsibly on the project will deepen your understanding of this course and better prepare you to tackle problems you will encounter in your professional career.
The final goal is to demonstrate a solid understanding about the implementation of distributed systems in Java. For that, define clear and well structured classes, communication protocols among the components of the system, and think carefully about the synchronization mechanisms needed to avoid race conditions.
Remember that learning takes practice, and making mistakes and fixing them is part of the process. Copying and pasting code from others or from generative AIs prevents you from acquiring essential skills, and it posses serious questions about your value as a future professional. Take your time, analyze the problem, and learn as much as you can with this experience.

Project Description

Overview. You will design and implement a distributed system in Java that simulates a network of *machines* sharing and transforming *resources* according to *reaction rules* concurrently launched by different *executors*.
Each machine manages a local set of resources (e.g., raw materials or data items). Certain global rules, called *reactions*, describe how resources from different machines can be combined or transformed into new resources.
Because reactions may involve resources hosted on different machines, the system must ensure that updates are performed *atomically* and *consistently* across the distributed network. To achieve this, you will implement a **Two-Phase Commit** (2PC) coordination protocol.

Machines and Reactions. The system consists of several machines, say, M_1, M_2, etc. Each machine maintains a local set of resources, represented as (name, quantity) pairs. For instance, M_1 may maintain the resources $(A, 3)$ (3 units of A available) and $(B, 5)$, while machine M_2 may maintain the resource $(C, 8)$.
Reactions describe how available resources can be consumed and produced. For instance, a rule can take the form[1]:

$$2A + 3C \longrightarrow 2B$$

That means that: if there are 2 units of A (in machine M_1) and 3 units of C (in M_2) then 2 units of B are produced (in M_1). One this rule is applied, the new state of the system is $M_1 : \{(A, 1), (B, 7)\}$, $M_2 : \{(C, 5)\}$.

Each reaction is therefore a distributed transaction that:

1. Checks availability of required resources.
2. Reserves (locks) them to prevent concurrent use.
3. Performs the transformation atomically.

Executors and Two-Phase Commit Protocol. Without coordination, simultaneous executions of different rules can:

- Attempt to consume the same resource concurrently,
- Partially apply a reaction (one machine updates, another fails),
- Leave the system in an inconsistent state (ex., a negative amount of resources).

To prevent these issues, updates must be atomic: the reaction either happens completely or not at all. This is the purpose of the Two-Phase Commit (2PC) protocol.
Let's call *executor* a process that is in charge of randomly executing, in an infinite loop, a set of reactions. For that, once a rule has been selected, the executor follows the following 2PC protocol:

- **Phase I (Prepare)**:
  1. The executor sends a `prepare` request to each participant machine involved.
  2. Each participant:
     a) Checks if the needed local resource are available.
     b) Locks those resources.
     c) Replies "YES" (ready to commit) or "NO" (cannot commit).
- **Phase II (Commit or Abort)**:

[1] You can assume that the resources on the left hand side of the reaction do not occur in the right hand side, and all the coefficients are positive.

1. If all participants replied "YES", then the coordinator sends a `commit` message and all the participant machines consume and produce the needed resources according to the rule being executed.
2. If there is a participant that replied "NO", then the coordinator sends an `abort` message and all machines release locks on the involved resources, and no changes are applied.

The above protocol ensures atomic distributed updates even if multiple machines are involved.

Some extra non-functional requirements

1. Your system must support several *Machines* (hosting the resources) and several *Executors*, each of them executing in an infinite loop a different set of reactions.
2. All the information needed to execute your programs (machines and executors) must be provided by *parameters* of your `main` classes. For instance, one should be able to launch a machine and an executor as follows:

```
java Machine --resource "(A,3)" --resource "(B,5)" ...
java Executor --reaction "A -> C" --reaction "B + C -> D" \
              --machine "127.0.0.1:12345" ...
```

3. Use Java Logging [2] so that all your programs leave a log with information about the execution of the system: when a resource has been locked, whether trying to lock a resource fails, when each phase of the 2PC ends, etc.

Deliverables and Rules

The project must be developed in groups of up to 4 students (no exceptions). Only one student per team must submit on Moodle a Zip file containing:

1. The source code, fully documented using Javadoc. Write in the header of the file a precise description of the purpose of the class and your design choices. For instance, mention that "*this class uses the protocol X in order to communicate Y and the data structure D is protected using the synchronized method M, etc*". You will loose **several points** if that information is not included.
2. A detailed README file explaining how to compile and use the system. You must provide a main class `Test` that uses a `ProcessBuilder`[3] to launch several instances of the Java virtual machine (in the same computer) to run the executor and machines with the resources. I will use that class to test your code. For instance:

[2] `https://docs.oracle.com/en/java/javase/11/core/java-logging-overview.html`
[3] `https://docs.oracle.com/javase/8/docs/api/java/lang/ProcessBuilder.html`

```
java Test --machine "..." --machine "..." --executor "..."
```

3. This project is a good opportunity to test that your code can be deployed in different machines. Hence, you have to add to your zip file the log files that resulted when you performed a test in the machines of the University[4]. I expect that you run at least 3 clients and 2 executors, all of them in different machines.

4. One of the objectives of the project is to evaluate the productivity of the system under the presence of several processes executing rules. Hence, you must test your system under different scenarios, varying the number of executors, the number of rules for each executor, the time D each executor waits before launching another reaction, etc. For each scenario, run your system at least 10 times[5] for a period of T seconds. Report the average of the resulting ratio c/t (committed transactions over the total number of transactions launched) for the 10 executions. Be aware of modifying the seeds of your random generators each time. Your zip file must contain a PDF with plots of c/t when varying the above parameters. The more experiments (and plots) you perform, the better.

5. The final score is **individual** and it heavily depends on the knowledge of the student about the code submitted. This knowledge will be assessed with technical questions during an "exam" on December 17 2025.

6. It is strictly forbidden to communicate with other groups by any means. Fragments of code slightly similar will be considered fraud and severely punished.

7. If you are planning to use extra packages, besides those included in the Java standard library, you have to send an email to `olarte@lipn.univ-paris13.fr`. Non authorized packages will not be accepted as part of the solution.

8. The deadline for sending the files on Moodle is December 11 at 12M (no exceptions).

[4]The logs output by your programs must include the IP and port used by the programs in all the requests.

[5]In fact, in order to have statistical significance, one should run the system more times in a Monte Carlo simulation (see e.g., `https://en.wikipedia.org/wiki/Monte_Carlo_method`)