Course Project: Optimistic Concurrency Control for Distributed Transactions

Distributed Systems 1, 2020 – 2021

In this project, you are requested to implement a protocol for the management of distributed transactions. The system allows multiple clients to make concurrent transactions consisting of sequences of read and write operations on items of a distributed data store. The protocol must ensure *strict serializability* for transactions that are successfully committed, aborting any transaction that would result in an inconsistency. The decision on whether to commit or abort is taken at the end of the transaction, based on the version of data items at each server of the store. This approach falls under the "optimistic concurrency control" paradigm: we assume that conflicts are rare and avoid the overhead of pessimistic locking approaches. Each transaction is handled by one of the many available coordinators, using a two-phase commit protocol to ensure that all involved servers agree on the decision.

1 General description

The project should be implemented in Akka with *clients*, *coordinators* and *servers* of the key-value data store being Akka actors, identified by a unique ID. For simplicity, we assume that the data held by servers consists of integer values. Each of these items is attached to a globally-unique key and another integer representing the data version. We assume that the nodes of the system do not change over the execution of the protocol: when the program starts, a configurable number of servers, coordinators, and clients are created and initialized. However, servers and coordinators may crash at any time. Whenever a node crashes, it also recovers after a given delay.

Client requests. A client contacts a random coordinator to begin a transaction (TXN). Then, it waits for confirmation from the coordinator that the TXN will be handled by it. If the client times out while waiting for confirmation, it will retry after some time. Once the TXN start is confirmed, the client has the guarantee that it will eventually commit or abort (i.e., the TXN should survive a coordinator crash). The client sends all read and write commands through the same coordinator for the entirety of the TXN. After read operations, it expects a reply from the coordinator carrying the current value for the requested key. The client can unilaterally abort the transaction, or ask the coordinator to commit it. Each client is involved in a single TXN at a time, starting a new one when the previous completes.

The client actor is provided. It reads item values two-by-two, and then writes subtracting a given amount from the first item and adding the same amount to the second one. This approach should help you test that the protocol works: the sum of visible values must be constant.

Private workspaces. Whenever a client requests a read or write for a given key, the coordinator forwards the requests to the server holding that item. The server cannot immediately satisfy requests, e.g., changing the values of visible items: it must wait for the coordinator to commit the transaction. Therefore, all operations of a TXN must be kept in a dedicated private workspace, recording the version of the items when they were accessed. When the coordinator aborts, the workspace can simply be deleted. When the coordinator commits, the changes in the workspace are applied to the visible items updating their version.

Validation and commit. The system uses optimistic concurrency control to ensure that committed TXNs do not conflict, which would lead to an inconsistent state. Private workspaces can be modified freely since they do not affect visible items, but they must be validated before changes are applied. The commit process is initiated by the coordinator of the TXN when requested by the client. The "prepare" message is sent to all servers taking part in the distributed TXN. Server reply with either "yes" (ready to commit) or "no" (cannot commit) depending on two conditions: 1. the version of the visible items did not change, and 2. no other conflicting transaction can be committed at the same time. For the latter condition, the servers should have a way to lock items during the commitment phase. You must only lock items if needed, and only for the time needed. If all servers are ready to commit (voted "yes"), the coordinator multicasts "commit", otherwise "abort".

Crash and recovery. Coordinators and servers may crash at key points of the algorithm. The system should implement a simple crash detection algorithm based on timeouts, as seen in the labs. Nodes should also recover after some time and resume operations, ensuring that the system remains consistent across failures.

2 Implementation-related assumptions and requirements

- We assume links are FIFO and reliable.
- To emulate network propagation delays, you are requested to insert small random intervals for unicast transmissions.
- Ensure actor encapsulation. Avoid sharing objects unless they are immutable.
- To emulate crashes, a participant should be able to enter the "crashed mode" in which it ignores all incoming messages and stops sending anything.
- During the evaluation it should be easy, with a simple instrumentation of the code, to emulate a crash at key points of the protocol, e.g., during the commit phase.
- Clients do not crash.
- Your implementation must include a mechanism to assess whether the state of the system is still consistent after a number of transactions have taken place.

3 Grading

You are responsible to show that your project works. The project will be evaluated for its technical content (algorithm correctness) and your ability to discuss implementation choices. *Do not* spend time implementing features other than the ones requested — focus on doing the core functionality, and doing it well.

A correct implementation of the whole requested functionality is worth 6 points. It is possible to submit programs implementing a subset of the requested features or systems requiring stronger assumptions. In these cases, lower marks will be awarded.

You are expected to implement this project with exactly one other student, however the marks will be individual, based on your understanding of the program and the concepts used.

4 Presenting the project

- You MUST contact through e-mail the instructor (gianpietro.picco@unitn.it) AND the teaching assistant (davide.vecchia@unitn.it), well in advance, i.e., at least a couple of weeks before the presentation.
- You can present the project at any time, also outside of exam sessions. In the latter case, the mark will be "frozen" until you can enrol in the next exam session.
- The code must be properly formatted otherwise it will not be accepted.
- Provide a short document (typically 2-4 pages) in English explaining the main architectural choices and discussing how your protocol satisfies the requirements of the project.
- Both the code and documentation must be submitted in electronic format via email at least one day before the meeting. The documentation must be a single self-contained pdf/txt. All code must be sent in a single tarball consisting of a single folder (named after your surnames) containing all your source files. For instance, if your surnames are Rossi and Russo, put your source files and the documentation in a directory called `RossiRusso`, compress it with zip or tar ("`tar -czvf RossiRusso.tgz RossiRusso`") and submit the resulting archive.
- The project is demonstrated in front of the instructor and/or assistant.

Plagiarism is not tolerated. Do not hesitate to ask questions if you run into troubles with your implementation. We are here to help.