Università della Svizzera italiana

Facoltà di scienze informatiche

Distributed Algorithms **Autumn 2022**

Prof. Fernando Pedone, Eliã Batista, Nenad Milosevic

Project

Description

Paxos is a protocol used to solve consensus in asynchronous systems. Simply put, consensus can be used by a set of processes that need to agree on a single value. More commonly though, processes need to agree on a sequence of totally ordered values - a problem known as *atomic broadcast*. In this project, you'll use the Paxos protocol to implement atomic broadcast.

In your implementation, four *roles* need to be provided:

- *clients:* submit values to proposers
- *proposers:* coordinate Paxos rounds to propose values to be decided
- *acceptors:* Paxos acceptors
- *learners:* learn about the sequence of values as they are decided

Your protocol should always guarantee *safety*. To guarantee liveness, in a complete Paxos implementation, one of the proposers should be elected as the leader. For simplicity, in this project, you will not be asked to implement a leader election oracle. However, you should not make any assumptions about which proposer is the leader, and you should support more than one proposer. Therefore, if two proposers are proposing in parallel and preventing each other from executing Phase 2 of the protocol, your implementation should ensure that they will keep trying until one of them hopefully succeeds.

Assumptions

You may assume that the number of acceptors is 3, that is, the failure of 1 acceptor can be tolerated. Note that more than 1 acceptor failing should not violate *safety*, only *liveness*. In general, proposing and learning values should be possible as long as a majority of acceptors and at least 1 of each client, proposer and learner are alive.

You may assume crash failures. That is, processes fail by halting and do not recover. This allows the following simplifications:

- no need to implement a recovery procedure for acceptors or learners
- all state can be kept in memory - no need to use stable storage

Implementation

Your Paxos implementation must be based on *IP multicast* only. You will support four different multicast groups, one for each role of the protocol. To ease the deployment of your implementation in a cluster environment, you are asked to support the configuration of nodes via a configuration file. This file must allow you to specify the multicast addresses for each Paxos role as follows:

```
<role> <ip address> <port>
```

An example configuration file would be:

We can assume asynchronous communication since the OS process scheduler renders the system approximately asynchronous to outside observers.

```
clients  239.0.0.1 5000
proposers 239.0.0.1 6000
acceptors 239.0.0.1 7000
learners   239.0.0.1 8000
```

Your implementation should also provide bash scripts to easily start the different roles:

- `acceptor.sh <id> <path to config file>` - starts an acceptor with the given id.
- `proposer.sh <id> <path to config file>` - starts a proposer with the given id.
- `learn.sh <id> <path to config file>` - starts a learner with the given id - the learner prints learned values to standard output (one value per line).
- `client.sh <id> <path to config file>` - starts a client with the given id and submits values it reads from standard input (one value per line).

An example execution might look like this:

```
## start the acceptors in background
$ ./acceptor.sh 1 config &
$ ./acceptor.sh 2 config &
$ ./acceptor.sh 3 config &

## start a proposer in background
$ ./proposer.sh 1 config &

## start a client in background that submits values read from a file
$ ./client.sh 1 config < test_input &

## start a learner printing to stdout
$ ./learn.sh 1 config
value 1
value 2
value 3
...
```

Deliverables

At the end of the project we expect a working system, that can be easily tested on a cluster of machines. Your submission should include the following:

- The source code and build scripts.
- A `README` file containing build instructions. If your language is not Java, Go, C, C++ and/or Python, provide clear instructions on how to obtain any requirements (compilers, interpreters and such). If there are any known limitations/bugs in your implementation, describe them here.
- The scripts for starting acceptors, proposers, learners and clients. The scripts will be executed from the root of your source folder.

Grading

Your grade will be based on correctness, completeness, readability and performance. The grade will be affected if your scripts/configuration/application differ from the specification given above.

Your implementation should guarantee the following:

- message loss or processes crashing should never violate safety (total order, agreement or integrity)
- if a majority of acceptors are killed, no progress should be made (asynchronous consensus assumption)
- learning values must be possible if there are a majority of acceptors and 1 of each other role (no crashes or message loss)

Some of the tests you can expect:

1. Proposing 100 values per client (2 clients, 2 proposers, 3 acceptors, 2 learners). Check that learners learn values in total order. Check that values that were proposed were learned. Repeat with 1000 and 10000 values per client.
2. Repeat test 1 with only 2 acceptors.
3. Repeat test 1 with only 1 acceptor.
4. Repeat test 1 with some % of message loss.
5. Repeat test 1 and kill 1 or 2 acceptors while values are being proposed
6. Learners catch up. Start 3 acceptors, 1 proposer, and 1 learner. A client proposes 100 values. Once finished, start an additional learner and a client that proposes another 100 values. Check that learners learn values in total order. Specifically, the newer learner needs to learn previous values before the new ones.

Algorithm:
1. Configuration file defines how many clients, acceptors, proposers, learners
2. Bash script generates N docker containers with each having a specified [agent type, IP, Port]
3. Each agent (corresponding to a docker container) initializes according to it's type
4. Bash script inputs test data and outputs learner(s)
5. Unit tests for edge cases

Architecture:
1. Agent class
-Acceptor
-Client
-Proposer
-Learner
2. Message class (for storing each type of message in the protocol) (drop message with % probability)
3. Communication Channel class??? (for simulating lossy channels according to theory)