COM3026 Distributed Systems

Summative Coursework 1

Release date: Tuesday, November 14th, 2023

Submission date: Tuesday, January 23th (Week 15), 2024, 16:00

1 General Guidelines

This coursework contributes to **40%** of your final grade of this module. The coursework will require completion of the following two coding tasks:

- Task 1 (Section 2): Abortable Paxos Consensus [**70 marks**]
- Task 2 (Section 3): A fault-tolerant 3-tier service using Paxos [**30 marks**]

The solutions for the tasks above must be submitted electronically via SurreyLearn by the **deadline** as explained in Section 4. Pay attention to late submission policies of the University: late submissions will lead to 10% penalty per day and any submission after **25th January 2024, 16:00** will receive 100% penalty. You can work in teams of **up to 2** people.

The assessment will be based on the criteria specified in Section 5.

2 Task 1: Abortable Paxos Consensus [70 marks]

Write an Elixir module `Paxos` implementing an abortable variant of Paxos [3], which is similar to the Abortable Consensus protocol studied in class [1]. Your implementation will be run by a collection of Elixir processes up to a minority of which can crash. Each process participating in the protocol will run a replica of your program.

Your implementation must be able to support *multiple instances* of consensus. Each instance is associated with a unique identifier (e.g., an integer) and is started whenever some process proposes a value for this instance. Instances with distinct identifiers are independent of each other and may reach different decisions. However, all decisions associated with the same instance must be identical (*Agreement*), and every value decided by the same instance must be equal to one of the values proposed for this instance (*Validity*).

Your Paxos module must implement an API consisting of the following public functions:

- `start(name, participants)` is a function that takes an atom `name`, and a list of atoms `participants` as arguments. It spawns a Paxos process, registers it in the global registry under the name `name`, and returns the identifier of the newly spawned process. The argument `participants` must be assumed to include symbolic names of *all* replicas (including the one specified by `name`) participating in the protocol.
- `propose(pid, inst, value, t)` is a function that takes the process identifier `pid` of an Elixir process running a Paxos replica, an instance identifier `inst`, a timeout `t` in milliseconds, and proposes a value `value` for the instance of consensus associated with `inst`. The values returned by this function must comply with the following requirements:

- `{:decision, v}` must be returned if the value `v` has been decided for the instance `inst`. Note that `v ≠ value` is possible if a competing agreement attempt was able to decide `v` ahead of the attempt initiated by this propose call.
- `{:abort}` must be returned if an attempt to reach an agreement initiated by this propose call was interrupted by another concurrent attempt with a higher ballot. In this case, an application implemented on top of Paxos may choose to reissue `propose` for this instance (e.g., if the invoking process is still considered a leader).
- `{:timeout}` must be returned if the attempt to reach agreement initiated by this propose call was unable to either decide or abort before the expiration of the specified timeout `t`. This may happen e.g., if the Paxos process associated with `pid` has crashed.

You can assume that every Paxos process has *at most one* outstanding invocation of propose for the same instance at any given time: i.e., if `propose(pid1, i1, _, _)` was invoked and has not yet returned, then no `propose(pid2, i2, _, _)` such that `pid2=pid1` and `i2=i1` can be called before the first invocation returns.

- `get_decision(pid, inst, t)` is a function that takes the process identifier `pid` of an Elixir process running a Paxos replica, an instance identifier `inst`, and a timeout `t` in milliseconds. It returns `v ≠ nil` if `v` is the value decided by the consensus instance `inst`; it returns `nil` in all other cases.

The individual Paxos processes may be deployed within the same node (i.e., Erlang VM), or on different nodes (which may also be deployed on different physical machines). The Paxos clients run in their own Elixir processes and use pids of the Paxos processes to invoke the API functions above. These pids can be obtained from the symbolic names of the Paxos processes using the global registry. Further guidance on testing your Paxos implementation is available in the appendix.

3 Task 2: A fault-tolerant 3-tier service using Paxos [30 marks]

Implement a fault-tolerant application service of your choice which uses your Paxos implementation via the API specified in Section 2. The structure of your service should follow *the three-tier architecture* design pattern (https://en.wikipedia.org/wiki/Multitier_architecture) in which front-end clients interact with servers implementing the application business logic, and the latter interact with the Paxos processes at the backend to achieve fault-tolerance by consistently replicating the command log (or ledger). A replicated command log is obtained using the multi-instance functionality of the Paxos API in Section 2, i.e., the command occupying the *k*-th entry in the log is determined by the outcome of the instance *k* of consensus.

The examples of services you may consider implementing include airline seat reservation system, digital currency, shared bulletin board, key/value data store, distributing locking service, peer-to-peer game (if you are ambitious), etc.

For your guidance, a sample code of an application server (`account_server.ex`) implementing the business logic of a simple bank account (with the deposit, withdraw, and balance API) is included with the project brief. Note however that this implementation is quite basic. In particular, it does not guarantee liveness (i.e., may never terminate) in cases when multiple servers keep interrupting each other by submitting competing proposals for the same instance of consensus. It is also not particularly efficient as it assumes that at most one deposit or withdraw command can be simultaneously invoked on the same server.

To be awarded full marks for this task, you are *only* required to submit an implementation of the *application server of your choice*, which is both *safe* (in terms of its API semantics) and *live* under the assumptions of *partial synchrony* (i.e., every client operation is guaranteed to terminate provided the system behaves synchronously for sufficiently long). However, you are encouraged to implement further performance optimisations as well as client front-ends. These extra features will be evaluated on the basis of the coursework presentations and up to the **two** implementations judged most interesting by the module's staff and students will be awarded prizes.

4 What to submit

Please submit the following files:

- **paxos.ex**: the source code of your Paxos module implementation for Task 1
- **server.zip**: an archive including the source code for your application server implementation. This should include a **README** file in a readable format of your choice (e.g., plain text, PDF or Word) which provide the following details:
    - the API of your service,
    - its safety and liveness properties,
    - the usage instructions and assumptions, and
    - details of any extra features that have been implemented.

5 Assessment criteria

Task	Requirements	Marks
Paxos Layer Tolerating Process Crashes: **70 marks**	Correctly implements Uniform Consensus in the presence of process crashes in both single and multi-node deployments.	Solves Uniform Consensus correctly in failure-free scenarios: **30 marks** • No failures occur and no ballots execute concurrently: **10 marks** • No failures and 2 concurrent ballots: **10 marks** • No failures and many concurrent ballots: **10 marks** Solves Uniform Consensus correctly in the presence of crashes: **20 marks**

		• One non-leader process crashes, and no ballots execute concurrently: **5 marks** • A minority of non-leader processes crash, and no ballots execute concurrently: **5 marks** • The leader of a ballot crashes, and no ballots execute concurrently: **5 marks** • The ballot leader and some non-leader processes crash; no ballots execute concurrently: **5 marks** Solves Uniform Consensus correctly the presence of complex failure patterns and asynchrony: **20 marks** • Many concurrent ballots, cascading crashes (up to a minority) of both leaders and non-leaders: **10 marks** • Many concurrent ballots, cascading crashes (up to a minority) of both leaders and non-leaders, some processes are randomly delayed: **10 marks**
Fault-tolerant service using Paxos: **30 marks**	Correctly implements a chosen fault-tolerant service in the presence of process crashes in both single and multi-node deployments.	Correctly implements a chosen fault-tolerant service: **30 marks** • Safe in terms of its API and properties: **10 marks** • Live under the assumption of partial synchrony: **10 marks** • README file detailing the service API (4 marks), usage instructions and any assumptions that have been made (3 marks), and safety and liveness properties (3 marks): **10 marks**

Reference

[1] COM3026, Weeks 7 and 8 lectures.
[2] COM3026, Lab assignments and tutorials
[3] Lamport, L. (2001). Paxos Made Simple. ACM SIGACT News (Distributed Computing Column), 32(4), December 2001, pages 51 – 58. A copy is available on SurreyLearn

6 Appendix: Testing the Paxos module (Task 1)

6.1 Testing using IEx

To test your Paxos module using IEx, first start a desired number of Paxos processes, store their pids in a local variable, and then use these pids to exercise various functions of the API defined in Section 2. For example, a possible testing scenario may proceed as follows:

1. Start three Paxos processes with symbolic names `:p1`, `:p2`, and `:p3`

```
procs = [:p1, :p2, :p3]
pids = Enum.map(procs, fn p -> Paxos.start(p, procs) end)
```

2. Propose a value `{:hello, "world}` via process `:p3` for the instance `1` of consensus with the timeout duration of 1sec:

```
Paxos.propose(:global.whereis_name(:p3), 1, {:hello, "world"},
1000)
```

3. Retrieve the decision value of the instance `1` via process `:p2` with timeout=0.5sec:

```
Paxos.get_decision(Enum.at(pids, 2), 1, 500)
```

4. Invoke two concurrent propose attempts at two different Paxos replicas with distinct values competing for the same consensus instance `2`:

```
p_fun = fn n, i, v, t -> (fn ->
Paxos.propose(:global.whereis_name(n), i, v, t) end) end
spawn(p_fun.(:p1, 2, :hola, 500)); spawn(p_fun.(:p2, 2, :hi,
500))
```

5. Retrieve the decision value for the instance `2` via process `:p3`

```
Paxos.get_decision(Enum.at(pids, 2), 2, 1000)
```

6. Kill the three Paxos processes created above:

```
Enum.each(pids, fn p -> Process.exit(p, :kill) end)
```

6.2 Testing using the Bank Account server

You can use the provided bank account server to generate various execution scenarios for your Paxos implementation as well as to validate its compliance with the specified API. A possible testing scenario can proceed as follows:

1. Start three Paxos processes with symbolic names `:p1`, `:p2`, and `:p3` as described above.
2. Start a bank account server with name `:server1` and associate it with Paxos process `:p1`

```
a = AccountServer.start(:server1, :global.whereis_name(:p1))
```

3. Issue a series of deposit and withdrawal operations with various amounts, e.g.,

```
AccountServer.deposit(a, 10)
AccountServer.withdraw(a, 5)
```

```
AccountServer.deposit(a, 15)
AccountServer.deposit(a, 5)
AccountServer.withdraw(a, 1)
Etc.
```

4. Issue a balance operation

```
AccountServer.balance(a)
```

5. Verify that the following safety properties hold: (1) balance = (sum of all successful deposits – sum of all successful withdrawals), and (2) withdraw fails with `:insufficient_funds` if the current balance is less than the requested withdrawal amount.

6.3 Unit testing via automated testing framework

Test your implementation on a suite of unit test for the assessment criteria above using an automated framework. The framework is available in the following Dropbox folder: https://www.dropbox.com/sh/kf30swl4cq2wne0/AACkupNA5mNdNT4DdS-0cynta?dl=0. The usage instructions can be found in the README file.