ID2203 Tutorial 4 - Consensus

Niklas Ekström neks(@)kth.se
Cosmin Arad icarad(@)kth.se
Tallat M. Shafaat tallat(@)kth.se

1 Introduction

The goal of this tutorial is to understand and implement a Paxos Uniform Consensus algorithm, as well as an Eventual Leader Election algorithm, for the partially-synchronous (fail-noisy) system model.

2 Assignment

In this assignment, you have to implement the following three algorithms/abstractions in Kompics:

- Paxos Uniform Consesus (PUC) - Algorithm 5.7 in the book. Leader-Driven Consensus
- Abortable (Epoch) Consensus (AC) - Algorithm 5.6 in the book. Read/Write Epoch Consensus
- Eventual Leader Detector (ELD) - Algorithm 2.9 in the book.

The algorithms have been re-written in this document with support for multiple consensus instances. This means that there can be multiple instances of consensus running at the same time. Each such instance is differentiated by an identifier called Paxos/consensus instance and denoted by *id* in the algorithms. Thus, when a node wants to propose a value, it should also provide the Paxos instance for which it wants to propose that value.

It is highly recommended that you read the afore-mentioned algorithms in the book before starting this assignment. Also note that PUC uses both AC and ELD.

In ELD, the function *select* can be implemented to return the *min* from the addresses passed to it. Thus, the lowest ranked alive process will become the leader.

Your implementation should provide, available via scenario and GUI, the following commands:

Pi-j This means that the node on which this command is executed should propose the value j for a consensus instance identified by i. For simplicity, both i and j should be integers.

Dk This means that the node n on which this command is executed should wait until it gets a decision to all previous (ongoing) proposals made by n, then wait for k milliseconds, and then process any further commands. If there is no previous (ongoing) proposal made by n, this command should simply wait for k milliseconds and then process any further commands.

W The node should sort all the decisions received thus far according to Paxos instance identifier and print them out. Please note that you should give enough delay before executing this command so that all the nodes have decided for all Paxos instances.

Example: `P1-7:D100:P3-5:P4-9:D20000:W`
The above command means that the node should propose value 7 in Paxos instance 1. After the node receives a decide for consensus instance 1, it should wait for 100 milliseconds and then propose value 5 in Paxos instance 3. Immediately following that (no waiting for a decision), the node should propose value 9 for Paxos instance 4. The node should then wait for Paxos instances 3 and 4 to finish. Next, the node should sleep for 20 seconds and then sort and print the decided values for all Paxos instances (even those that it didn't initiate/propose in) to the console.

You can then compare the decided values at different nodes.

3 Exercises

You have to upload your source code and report answering the exercises given below.

Exercise 1 Both Abortable Consensus (Algorithm 2 and Algorithm 3) and Paxos Uniform Consensus (Algorithm 4 and Algorithm 5) use a *seenIds* set to manage different concurrent instances of consensus. The *seenIds* set grows indefinitely. Explain in your written report, for each algorithm in part, how this set could be garbage collected.

Exercise 2 Can PUC be used in a fail-recovery model? If so, under what condition/assumption? If not, why?

Exercise 3 Construct a topology with 3 processes, use 1000ms latency between the nodes. Experiment with the following two scenarios:

```
1:  D2200:P1-1
2:  D2000:P1-2
3:  D2000:P1-3
```

and

```
1:  D2000:P1-1
2:  D2200:P1-2
3:  D2200:P1-3
```

What value is decided for Paxos instance 1 in these two scenarios? Give reasoning in your report why that particular value has been chosen.

Exercise 4 Construct a topology with 2 processes. Assign link delays and initialize ELD period and increment in such a way that the two processes that initiate concurrect PUC proposals abort at least once in AC. Present the topology, the ELD parameter (TimeDelay) and the operation sequences (ops) of the two processes and explain the execution.

Exercise 5 In the original presentation of the Paxos algorithm, processes can have different roles: *proposer*, *acceptor*, and *learner*. For each event handler of AC and PUC, tell what process role(s) are meant to execute the respective handler. In other words, what types of messages are sent and expected (and what events are triggered and expected) by each process role. Of course one process can act in more than one role. In fact, in our case, processes act in all roles, but interestingly, they need not to. Please refer to lecture 10 and the "Paxos Made Simple" paper available on the course web page.

Exercise 6 In this exercise, you will read the paper "Consensus: The Big Misunderstanding" (available on the course web page) and you should answer the following questions.

Question 6.1. According to the paper, is atomic broadcast (total order broadcast) equivalent to consensus? What does that mean?

Question 6.2. According to the paper, are all notions of time impossible in an asynchronous model?

Question 6.3. Assume we can set the timeout value very high, e.g. 1 minute, and that would guarantee that the failure detectors will never suspect a correct node inaccurately. We have then circumvented the FLP impossibility. What practical implications does this have?

Question 6.4. According to the paper, is it possible to solve consensus in one step?

Algorithm 1 Fail-noisy Eventual Leader Detector

Implements:
Eventual Leader Detector (Ω).

Uses:
PerfectPointToPointLinks (pp2p).

```
upon event ⟨ Init ⟩ do
    leader := select(Π);                      ▷ deterministic function known in advance
    trigger ⟨ trust | leader ⟩;               ▷ by all processes, e.g. min
    period := TimeDelay;
    for all p_i ∈ Π do
        trigger ⟨ pp2pSend | p_i, [HEARTBEAT] ⟩;
    end for
    candidateset := ∅;
    startTimer(period);
end event

upon event ⟨ Timeout ⟩ do
    newleader = select(candidateset);
    if (leader ≠ newleader ∧ newleader ≠ NIL) then
        period := period + Δ;
        leader := newleader;
        trigger ⟨ trust | leader ⟩;
    end if
    for all p_i ∈ Π do
        trigger ⟨ pp2pSend | p_i, [HEARTBEAT] ⟩;
    end for
    candidateset := ∅;
    startTimer(period);
end event

upon event ⟨ pp2pDeliver | p_j, [HEARTBEAT] ⟩ do
    candidateset := candidateset ∪ {p_j};
end event
```

Algorithm 2 Abortable Consensus: Read Phase

Implements:

AbortableConsensus (ac).

Uses:

BestEffortBroadcast (beb);
PerfectPointToPointLinks (pp2p).

```
upon event ⟨ Init ⟩ do
    seenIds := ∅;
    majority := ⌊N/2⌋+1;
end event

procedure initInstance(id) is
    if (id ∉ seenIds) then
        tempValue[id] := val[id] := ⊥;
        wAcks[id] := rts[id] := wts[id] := 0;
        tstamp[id] := rank(self);
        readSet[id] := ∅;
        seenIds := seenIds ∪ {id};
    end if
end procedure

upon event ⟨ acPropose | id, v ⟩ do
    initInstance(id);
    tstamp[id] := tstamp[id] + N;
    tempValue[id] := v;
    trigger ⟨ bebBroadcast | [READ, id, tstamp[id]] ⟩;
end event

upon event ⟨ bebDeliver | p_j, [READ, id, ts] ⟩ do
    initInstance(id);
    if (rts[id] ≥ ts or wts[id] ≥ ts) then
        trigger ⟨ pp2pSend | p_j, [NACK, id] ⟩;
    else
        rts[id] := ts;
        trigger ⟨ pp2pSend | p_j, [READACK, id, wts[id], val[id], ts] ⟩;
    end if
end event
```

Algorithm 3 Abortable Consensus: Write Phase

```
upon event ⟨ pp2pDeliver | p_j, [NACK, id] ⟩ do
    readSet[id] := ∅;
    wAcks[id] := 0;
    trigger ⟨ acReturn | id, ⊥ ⟩;
end event

upon event ⟨ pp2pDeliver | p_j, [READACK, id, ts, v, sentts] ⟩ do
    if (sentts = tstamp[id]) then
        readSet[id] := readSet[id] ∪ {(ts, v)};
        if (|readSet[id]| = majority) then
            (ts, v) := highest(readSet[id]);                    ▷ largest timestamp
            if (v ≠ ⊥) then
                tempValue[id] := v;
            end if
            trigger ⟨ bebBroadcast | [WRITE,id,tstamp[id],tempValue[id]] ⟩;
        end if
    end if
end event

upon event ⟨ bebDeliver | p_j, [WRITE, id, ts, v] ⟩ do
    initInstance(id);
    if (rts[id] > ts or wts[id] > ts) then
        trigger ⟨ pp2pSend | p_j, [NACK, id] ⟩;
    else
        val[id] := v;
        wts[id] := ts;
        trigger ⟨ pp2pSend | p_j, [WRITEACK, id, ts] ⟩;
    end if
end event

upon event ⟨ pp2pDeliver | p_j, [WRITEACK, id, sentts] ⟩ do
    if (sentts = tstamp[id]) then
        wAcks[id] := wAcks[id] + 1;
        if (wAcks[id] = majority) then
            readSet[id] := ∅; wAcks[id] := 0;
            trigger ⟨ acReturn | id, tempValue[id]] ⟩;
        end if
    end if
end event
```

Algorithm 4 Paxos Uniform Consensus (part 1)

Implements:

UniformConsensus (uc).

Uses:

AbortableConsensus (ac);
BestEffortBroadcast (beb);
EventualLeaderDetector (Ω).

```
upon event ⟨ Init ⟩ do
    seenIds := ∅;
    leader := false;
end event

procedure initInstance(id) is
    if (id ∉ seenIds) then
        proposal[id] := ⊥;
        proposed[id] := decided[id] := false;
        seenIds := seenIds ∪ {id};
    end if
end procedure

upon event ⟨ trust | p_i ⟩ do
    if (p_i = self) then
        leader := true;
        for all id ∈ seenIds do
            tryPropose(id);
        end for
    else
        leader := false;
    end if
end event

upon event ⟨ ucPropose | id, v ⟩ do
    initInstance(id);
    proposal[id] := v;
    tryPropose(id);
end event
```

Algorithm 5 Paxos Uniform Consensus (part 2)

```
procedure tryPropose(id) is
    if (leader = true ∧ proposed[id] = false ∧ proposal[id] ≠ ⊥) then
        proposed[id] := true;
        trigger ⟨ acPropose | id, proposal[id] ⟩;
    end if
end procedure

upon event ⟨ acReturn | id, result ⟩ do
    if (result ≠ ⊥) then
        trigger ⟨ bebBroadcast | [DECIDED, id, result] ⟩;
    else
        proposed[id]:= false;
        tryPropose(id);
    end if
end event

upon event ⟨ bebDeliver | p_i, [DECIDED, id, v] ⟩ do
    initInstance(id);
    if (decided[id] = false) then
        decided[id] := true;
        trigger ⟨ ucDecide | id, v ⟩;
    end if
end event
```