# ID2203 Tutorial 5

Cosmin Arad
icarad@kth.se

Handout: February 27, 2008
Due: March 5, 2008

## 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

Using the TBN framework, you will implement, as components, a Paxos uniform consensus abstraction (PUC) relying on an abortable consensus abstraction (AC) and on an eventual leader detector abstraction (ELD). These abstractions are described in Algorithm 1 (ELD), Algorithm 2 and Algorithm 3 (AC), and Algorithm 4 and Algorithm 5 (PUC). These algorithms are the ones in the textbook, adapted to support multiple instances of consensus.

You will use the same type of input as in the previous assignments, to specify your network of processes, i.e., a topology.xml file. You will reuse the communication component, the timer component, the delay component, and the beb broadcast component. You have to implement an EventualLeaderDetectorComponent (ELD), an AbortableConsensusComponent (AC), and a PaxosConsensusComponent (PUC), together with the events that are accepted and triggered by these components: TrustLeaderEvent, UcProposeEvent, UcDecideEvent, AcProposeEvent, AcReturnEvent and the messages that they exchange: HeartbeatMessage, ReadMessage, WriteMessage, ReadAckMessage, WriteAckMessage, NackMessage, and DecidedMessage. All state variables indexed by *id* in the algorithms sould be implemented with hashtables. You should use the function *min* for the *select* function in the eventual leader detector, so the lowest rank alive process becomes the leader.

You will have an `assignment5.xml` containing the architecture for testing your implementation. When launching the Assignment5.java you pass a string command line argument containing the consensus proposal operations issued by the application component:
`java Assignment5 <topology.xml> <nodeId> <ops>`.

You will slightly change the application component from the fourth assignment, to handle the UcDecideEvent and the TrustLeaderEvent, and an ApplicationEvent raised by the main program and handled by the application component, that contains the string `ops`. The `ops` string contains consensus proposal operations that are to be issued by the application component. Here is an example: `D500:P1-3:D100:P2-5:D30000`. This means that the application component will wait for 500 milliseconds, then it will propose the value 3 to PUC instance 1. Upon getting a decision from this first consensus instance, the application component waits for another 100 milliseconds and then issues a proposal with value 5, in a second consensus instance. Upon getting a decision from the second consensus instance, the application component waits for 30 seconds and then terminates the program. Note that this is the sequence of operations issued by one process. Other processes can get PUC decisions without proposing (by the beb broadcast). For example a second process with ops: `D10000:P1-7:D20000` should get a decition for value 3 (if the first process decided within the first 10 seconds).

In general, the type of values proposed to the PUC should be serializable objects, but here we are going to use just integers.

You use the same InitEvent to pass to the components the set of all processes ($\Pi$), or the number of processes, respectively. Note that the algorithms assume that each process knows and can communicate with all other processes in $\Pi$. This corresponds to a fully connected topology.

Implement the ELD, AC, and PUC components and experiment with them as instructed in the following exercises. Describe your experiments in a written report. For each exercise include the topology descriptors used, and explain the behavior that you observe.

You need to read Section 5.3 of the textbook, in order to fully understand the ideas behind the AC and PUC algorithms. Also read Section 2.5.5 of the textbook for a description of ELD.

ELD uses configuration parameters $TimeDelay$ (the initial value of period) and $\Delta$. They should be specified in an `eld.properties` configuration file that is passed to the ELD component. Here is an example `eld.properties` file:

```
period = 1000      # TimeDelay
```

```
increment = 1000   # Delta
```

The assignment is due on March 5th. You have to send your source code and written report by email before the next tutorial session. During the tutorial session you will present the assignment on a given topology description. You can work in groups of maximum 2 students. Be prepared to answer questions about your process's system architecture and explain the behavior of the algorithms. Any questions are welcome on the mailing list.

**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** We have the following known issue in Algorithm 2 and Algorithm 3. For one UC consensus instance, many AC proposals (AC instances) can be triggered with the same instance id. The AC, however, just uses the same instance id as the UC instance id, for all these AC instances. It returns successfully when receiving a majority of write acknowledgements and aborts upon receiving one negative acknowledgement. This may lead to a situation where one instance that would normally succeed be aborted by the receipt of an earlier late negative acknowledgement, or even worse, to a situation where one instance that would normally abort, because of ~~insufficient~~ write acknowledgements (followed by a negative acknowledgement), actually succeeds by the help of earlier late write acknowledgements. Improve the algorithm and your implementation to ignore such late acknowledgements[1].

**Exercise 3** Show (in your written report) that Algorithm 1 is a correct eventual leader detector, that is, it satisfies the properties of *eventual accuracy* and *eventual agreement* (see textbook page 55).

**Exercise 4** Use a topology with 3 processes and 3 bidirectional links with the same latency of 1000ms and an ELD period of 2000ms and execute PUC. Start process 0 with ops: `D500:P1-5:D40000` and process 1 with ops: `D500:P1-6:D40000`, and don't start process 2 yet. This is a special case of

[1] Hint: apply a technique that you have seen in the algorithm for implementing multiple-writers fail-silent atomic registers (RIWCM). This was left out of the algorithm for readability reasons.

failure called "initialy dead processes". When processes 0 and 1 begin to wait for 40 seconds, start process 2 with ops: `D500:P1-7:D10000`.

What is the value decided by process 2? Explain why by reasoning about the execution.

**Exercise 5** Can PUC be used in a fail-recovery model? If so, under what condition/assumption? If not, why?

**Exercise 6** Use a topology with 3 processes and 3 bidirectional links with the same latency of 1000ms and an ELD period of 2000ms and execute PUC. Start process 1 with ops: `D500:P1-6:D40000` and process 2 with ops: `D500:P1-7:D40000`, and don't start process 0 yet. When processes 1 and 2 begin to wait for 40 seconds, start process 0 with ops: `D500:P1-5:D10000`.

What is the value decided by process 0? Explain why by reasoning about the execution.

**Exercise 7** 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 parameters and the operation sequences (ops) of the two processes and explain the execution.

**Exercise 8** In the original presentation of the Paxos algorithm processes can have different roles: proposer, coordinator, acceptor, and learner. For each event handler of AC and PUC say processes with what role 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 the Fast Paxos paper available on the course web page.

**Algorithm 1** Fail-noisy Eventual Leader Detector

**Implements:**

Eventual Leader Detector ($\Omega$).

**Uses:**

PerfectPointToPointLinks (pp2p).

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

11: upon event ⟨ Timeout ⟩ do
12:     newleader = select(candidateset);
13:     if (leader ≠ newleader ∧ newleader ≠ NIL) then
14:         period := period + Δ;
15:         leader := newleader;
16:         trigger ⟨ trust | leader ⟩;
17:     end if
18:     for all p_i ∈ Π do
19:         trigger ⟨ pp2pSend | p_i, [HEARTBEAT] ⟩;
20:     end for
21:     candidateset := ∅;
22:     startTimer(period);
23: end event

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

**Algorithm 2** Abortable Consensus: Read Phase

**Implements:**

AbortableConsensus (ac).

**Uses:**

BestEffortBroadcast (beb);
PerfectPointToPointLinks (pp2p).

```
1: upon event ⟨ Init ⟩ do
2:     seenIds := ∅;
3: end event

4: procedure checkInstance(id) is
5:     if (id ∉ seenIds) then
6:         tempValue[id] := val[id] := ⊥;
7:         wAcks[id] := rts[id] := wts[id] := 0;
8:         tstamp[id] := rank(self);
9:         readSet[id] := ∅;
10:        seenIds := seenIds ∪ {id};
11:    end if
12: end procedure

13: upon event ⟨ acPropose | id, v ⟩ do
14:    checkInstance(id);
15:    tstamp[id] := tstamp[id] + N;
16:    tempValue[id] := v;
17:    trigger ⟨ bebBroadcast | [READ, id, tstamp[id]] ⟩;
18: end event

19: upon event ⟨ bebDeliver | p_j, [READ, id, ts] ⟩ do
20:    checkInstance(id);
21:    if (rts[id] ≥ ts or wts[id] ≥ ts) then
22:        trigger ⟨ pp2pSend | p_j, [NACK, id] ⟩;
23:    else
24:        rts[id] := ts;
25:        trigger ⟨ pp2pSend | p_j, [READACK, id, wts[id], val[id]] ⟩;
26:    end if
27: end event
```

**Algorithm 3** Abortable Consensus: Write Phase

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

6: upon event ⟨ pp2pDeliver | p_j, [READACK, id, ts, v] ⟩ do
7:     readSet[id] := readSet[id] ∪ {(ts, v)};
8:     if (|readSet[id]| > N/2) then
9:         (ts, v) := highest(readSet[id]);                ▷ largest timestamp
10:        if (v ≠ ⊥) then
11:            tempValue[id] := v;
12:        end if
13:        trigger ⟨ bebBroadcast | [WRITE,id,tstamp[id],tempValue[id]] ⟩;
14:    end if
15: end event

16: upon event ⟨ bebDeliver | p_j, [WRITE, id, ts, v] ⟩ do
17:    checkInstance(id);
18:    if (rts[id] > ts or wts[id] > ts) then
19:        trigger ⟨ pp2pSend | p_j, [NACK, id] ⟩;
20:    else
21:        val[id] := v;
22:        wts[id] := ts;
23:        trigger ⟨ pp2pSend | p_j, [WRITEACK, id] ⟩;
24:    end if
25: end event

26: upon event ⟨ pp2pDeliver | p_j, [WRITEACK, id] ⟩ do
27:    wAcks[id] := wAcks[id] + 1;
28:    if (wAcks[id] > N/2) then
29:        readSet[id] := ∅; wAcks[id] := 0;
30:        trigger ⟨ acReturn | id, tempValue[id]] ⟩;
31:    end if
32: end event
```

**Algorithm 4** Paxos Uniform Consensus (part 1)

**Implements:**

UniformConsensus (uc).

**Uses:**

AbortableConsensus (ac);
BestEffortBroadcast (beb);
EventualLeaderDetector ($\Omega$).

```
1: upon event ⟨ Init ⟩ do
2:     seenIds := ∅;
3:     leader := false;
4: end event

5: procedure checkInstance(id) is
6:     if (id ∉ seenIds) then
7:         proposal[id] := ⊥;
8:         proposed[id] := decided[id] := false;
9:         seenIds := seenIds ∪ {id};
10:    end if
11: end procedure

12: upon event ⟨ trust | p_i ⟩ do
13:    if (p_i = self) then
14:        leader := true;
15:        for all id ∈ seenIds do
16:            tryPropose(id);
17:        end for
18:    else
19:        leader := false;
20:    end if
21: end event

22: upon event ⟨ ucPropose | id, v ⟩ do
23:    checkInstance(id);
24:    proposal[id] := v;
25:    tryPropose(id);
26: end event
```

**Algorithm 5** Paxos Uniform Consensus (part 2)

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

7:  upon event ⟨ acReturn | id, result ⟩ do
8:      if (result ≠ ⊥) then
9:          trigger ⟨ bebBroadcast | [DECIDED, id, result] ⟩;
10:     else
11:         proposed[id]:= false;
12:         tryPropose(id);
13:     end if
14: end event

15: upon event ⟨ bebDeliver | p_i, [DECIDED, id, v] ⟩ do
16:     checkInstance(id);
17:     if (decided[id] = false) then
18:         decided[id] := true;
19:         trigger ⟨ ucDecide | id, v ⟩;
20:     end if
21: end event
```