Programming assignment 4

Consensus

Advanced distributed systems

2013

Mahboobeh Abdal Mahmood Abadi & Mehran Nasseri

KTH

2/28/2013

Contents

[1. Exercise 1 3](#_Toc349800372)

[2. Exercise 2 5](#_Toc349800373)

[3. Exercise 3 6](#_Toc349800374)

[4. Exercise 4 6](#_Toc349800375)

[5. Exercise 5 7](#_Toc349800376)

[6. Exercise 6 9](#_Toc349800377)

[6.1. Question 1 9](#_Toc349800378)

[6.2. Question 2 9](#_Toc349800379)

[6.3. Question 3 10](#_Toc349800380)

[6.4. Question 4 10](#_Toc349800381)

[Reference 11](#_Toc349800382)

# Exercise 1

It is not possible to garbage collect the *seenid* without making some changes in the algorithm. However, by replacing the beb-broadcast with regular reliable broadcast, to distribute the Decide message, we can safely garbage collect seenid. Before garbage collection, it must be ensured that all processes decided, and then we can safely increment the id and prevent any process to propose a value for that id (otherwise, if leader crashes after *ucdecide,* the new leader must continue the current id to ensure all processes that have not decided make same decision).

**Reasons that prevent us from garbage collecting seenid in given algorithm:**

* Our algorithm assumes fail-noisy model and uses eventual leader detector so we can never ensure all processes *ucdecided*. To not violating termination property of consensus, we can only wait for receiving ACK from the majority but not from all processes.
* If we garbage collect *seenid* after *ucdecide*, the leader process can now propose and decide same or other value for the same id again. This will violate Integrity property (Every correct process decides at most once).
* Moreover, even if we ask the leader to increment the id after it *ucdecided*, this will violate Uniform agreement (No two processes decide differently) and integrity properties. Assume by contradiction to garbage collect *seenid* after *ucdecide*. We have execution in which, leader propose *v* and crashes after *ucdecide* and only process *q ucdecide v*. The next leader will not increment the *id*, since it has not *ucdecided* yet. It reads and selects highest (timestamp, value) from the majority, which is still *v*, and *ucdecide* *v*. Apparently, *q* will *ucdecide* one more time since it garbage collected *seenid* and *decided [id]* is false. This will violate integrity property. On the other hand, even if new leader increment the *id*, the previous consensus instance will be left undecided by some nodes while some *ucdecided*.
* The other execution is when leader collect the read set from the majority who *ucdecided* and garbage collected their *seenid*, so leader will propose new value. In such case if, a process has not *ucdecided* yet will decide this new value. This will violate uniform agreement on this process and causes the other processes to decide more than once, which means violating integrity.
* Alternatively, if algorithm gives leader responsibility of triggering a *RemoveId* message after receiving *ucdecide* ACK from majority; there could be an execution in which a leader proposes *v*, trigger *RemoveId* message, crashes, and all processes except *q ucdecided* and *RemoveId,* but *q* still not received the write message. Now, *q* is leader, and sends the read message. Since all processes garbage collect *seenId* there is no way to obtain previous value and timestamp. Therefore, *q* can propose and decide a new value. This will violate uniform agreement on process *q* and causes the other processes to decide more than once, which means violating integrity.

**A way to garbage collects seenid safely**

We should garbage collect seenid *x* if and only if all processes ucdecided for consensus instance id *x* AND all previous seenids has been ucdecided.

We should modify the algorithm as follow to overcome problems mentioned in previous section:

* Use regular reliable broadcast to distribute the decide message to ensure that if a correct process delivers decide message dm, all correct processes eventually deliver decide message dm.
* Variable lastDecidedId = -1 (assuming that id will be started from 0)
* After ucdeciding to ensure all previous seenids has been ucdecided do:
  + checkToGarbageCollect (id)
    - If lastDecidedId+1= Id (id of decide message) and decided[id] := true, then
      * lastDecidedId=id
      * Garbage collect seenId
      * checkToGarbageCollect (id+1)
    - else return
  + Leader must propose other value(s) with Id(s) higher than lastDecidedId to start new consensus instance.
  + Nodes must be prevented to send ACK if id <= lastDecidedId. Otherwise, since seenid has been removed the leader will have opportunity to propose different value than the one already ucdecided.

We must garbage collect seenid *x* if and only if seenid *x-1* was garbage collacted to not violate termination property i.e we must garbage collect seenid *x* if all previous seenids has been ucdecided. Assume that leader proposes values for 3 ids {0, 1, 2} and all nodes ucdecided for id 2, and still not decided for 0, and 1. If we remove seenid 2 and increment lastdecidedId to 2, so nodes will never send any ACK for id 0, and 1, since we have if id <= lastDecidedId not send ACK. This will violate termination property.

Using the reliable broadcast (RB), we can ensure that if a correct process delivers decide message dm, all correct processes eventually deliver decide message dm (agreement property).

Assume that leader ucdecided and crashed. Now we have two conditions:

* + Either none of the processes rbdeliver decide message, so the new leader will propose the same value again according to the algorithm and no one garbage collectes seen id.
  + Alternatively, even if one of the processes rbdelivered the decide message, so according to the agreement property of RB every correct process eventually rbdelivers the decide message and ucdecide.
    - In this case any process that ucdecided must not send ACK message to the leader (if lastDecidedId >= receivedId) and from the agreement property of RB we know that the leader eventually rbdelivers the decide message, ucdecides, and terminates.

In conclusion by using RB we can ensure all correct process eventually ucdecide and we can safely garbage collect the seenid from both algorithm at this time.

# Exercise 2

Since in crash recovery model a correct process can crash and recover, we should apply some changes to make the current algorithm utilizable in crash recovery model as follow:

* **Ignoring Integrity and relaxing termination property:** “only processes that never crash are required to decide and a process may decide multiple times; both modifications are inevitable in the fail-recovery model” (Christian Cachin, Rachid Guerraoui, Lu´ıs Rodrigues, 2011).

For tolerating crash-recovery, a process may propose a particular value several times and may decide several times. It is assumed that crashed process proposes the same value again upon recovery. This guarantees that the consensus abstraction eventually terminates and decides a value.

* **Logging in stable storage:** the algorithm must log following data:
  + The Timestamp, leader, consensus id, decision value from which the higher layer must retrieve it upon recovery
  + Therefore when a node crashes before handling decide event, the process can the process may retrieve the decision value/ consensus id pair from stable storage upon recovery.
  + The underlying abortable (Epoch) Consensus (AC) abstractions should deliver their outputs through variables stored in stable storage; these variables must also be retrieved by Paxos Uniform Consensus upon recovery. If these values specify that the AC delivered some output before the crash (by acReturn id/ value pair), then the algorithm takes the appropriate steps.
* **Communication**: stubborn point-to-point links abstraction and a stubborn best-effort broadcast abstraction. This ensures that the process, even if it crashes and recovers a finite number of times, will eventually process every message sent to it.

To put it briefly, the algorithm must be prepared to resume in two ways:

* In case it has missed starting a consensus instance event
* When it missed an acReturn event for a consensus instance to decide

Upon recovery, a process retrieves from stable storage the missed data, which has been recorded by the AC. The algorithm should also log its own state, consisting of the timestamp, leader, of the current consensus id, the decision value, and reclaim them upon recovery.

# Exercise 3

The decided value depends on ELD period. Since our algorithm is leader driven, the nodes will never compete to propose, except the time of having multiple leaders due to eventual agreement property of ELD. Leader is only one who can write a value. Moreover, according to our ELD implementation, node number one is the first leader accepted by all nodes and no one will try to propose at least until first timeout event in ELD.

Let us to consider following settings:

* If we set ELD period greater than link delay (1000 ms), no one will suspect node 1 falsely. Therefore for id=1 we will have value 1 decided in both scenarios.
* False suspicion may happen if we set ELD period less than link delay (1000 ms), so after first timeout event each node may trust itself as leader and each one propose its own value and finally any other value than 1 may be decided. However, in second scenario we have initial delay Dk that prevents nodes from proposing immediately. For nodes 2 and 3 Dk is at least two times more than any possible value for period (remember that we assumed a period less than 1000ms), thus these nodes will trust node 1 again before having chance of proposing their value. Therefore, value 1 would be decided again.

On the other hand, in first scenario the initial delay Dk for nodes 2 and 3 is 2000. If we set the ELD period as close as possible to link delay value, nodes 2 and 3 will have chance of proposing their value right before trusting node 1. According to the algorithm since node 3 starts with highest timestamp ‘6’, so it will receive read ACK from majority and it sends the write message with value 3. Now we can say that at least node 3 has written a value 3 in val[1]. Therefore, upon receiving read message from node 1, which is now trusted by everyone, node 3 will send ACK with value 3 and timestamp 6. Since this is a value with highest timestamp, so far, node 1 will ucdecide this value.

Moreover, this situation for scenario 1 will only happen if you set ELD period as close as to link delay value to receive the first heartbeat ***right after*** first timeout event. For any other value that does not hold this condition (e.g. period =500) value 1 would be ucdecided.

# Exercise 4

* Topology:

node(1, "127.0.0.1", 22221);

node(2, "127.0.0.1", 22222);

defaultLinks(3000, 0);

* Scenario:

command (1, "P1-1:D100:W");

command (2, "P1-2::D100:W");

* Delta = 500;
* Time delay = 2000;

Execution steps:

* + At the start of the execution, both process select node 1 as leader. Moreover, node 1 proposes value 1.
  + After a while, due to greater link delay compare to ELD period, the node 2 start to trust itself as leader and propose value 2. From this moment, we have 2 leaders trying to propose different value.
  + Now the timestamp in node one is 3 (rank (self) +N) and in node 2 is 4. Therefore, node 2 NACK the read message received from 1, but node 1 send ACK to node 2.
  + Now node 1 increments the timestamp by N and sends another read message. While node 2 sends the write message with timestamp=4 and value=2. This write message will not be acknowledged since the read timestamp in node one is 5 and it is greater than timestamp of the received write message.
  + At this time node 2 receives the heartbeat message from node 1 and start to trust it. Therefore, node 2 will never try to propose again.
  + Node 1 receives the read ACK from node 2. The value/timestamp pair from node 1 is null/0 (since it has never acknowledged any write message) and the value/timestamp pair from node 2 is 2/4 (since it acknowledged its own write message). The node 1 selects the value with highest timestamp, which is 2.
  + Finally, node 1 sends the write message, with timestamp 5 and value 2. After receiving WACK from majority, node 1 beb broadcast the decision.

Therefore, one of the concurrent consensuses by node 2 was aborted and second one terminated by sending decide message from node 1, which was the leader. However, the decided value is the value proposed by node 2.

# Exercise 5

* Abortable Consensus:
  + Proposer
    - upon event ⟨ acPropose | id, v ⟩
      * sends: ⟨ bebBroadcast | [Read, id, tstamp[id]] ⟩;
    - upon event ⟨ pp2pDeliver | pj , [Nack, id] ⟩
      * sends ⟨ acReturn | id, ⊥ ⟩;
    - upon event ⟨ pp2pDeliver | pj , [ReadAck, id, ts, v, sentts] ⟩
      * sends ⟨ bebBroadcast | [Write,id,tstamp[id],tempValue[id]] ⟩;
    - upon event ⟨ pp2pDeliver | pj , [WriteAck, id, sentts] ⟩
      * sends ⟨ acReturn | id, tempValue[id]] ⟩;
  + Acceptor
    - upon event ⟨ bebDeliver | pj , [Read, id, ts] ⟩
      * sends
        + either ⟨ pp2pSend | pj , [Nack, id] ⟩;
        + or ⟨ pp2pSend | pj , [ReadAck, id, wts[id], val[id], ts] ⟩;
    - upon event ⟨ bebDeliver | pj , [Write, id, ts, v] ⟩
      * sends
        + either trigger ⟨ pp2pSend | pj , [Nack, id] ⟩;
        + or ⟨ pp2pSend | pj , [WriteAck, id, ts] ⟩;
  + Learner
    - N/A
* Paxos Uniform Consensus
  + Proposer
    - upon event ⟨ trust | pi ⟩
      * sends ⟨ acPropose | id, proposal[id] ⟩;
    - upon event ⟨ ucPropose | id, v ⟩
      * sends ⟨ acPropose | id, proposal[id] ⟩;
    - upon event ⟨ acReturn | id, result ⟩
      * sends ⟨ bebBroadcast | [Decided, id, result] ⟩;
  + Acceptor
    - N/A
  + Learner
    - upon event ⟨ bebDeliver | pi, [Decided, id, v] ⟩
      * sends ⟨ ucDecide | id, v ⟩;

# Exercise 6

# Question 1

**Is atomic broadcast (total order broadcast) equivalent to consensus?**

Yes.

* Any solution to the atomic broadcast problem can be used to solve the consensus problem
  + initially every process pi atomically broadcasts its initial value vi
  + the decision is the first value v delivered”
* Consensus Can be used to solve Atomic broadcast
  + Messages are first disseminated using a reliable broadcast instance with identifier rb.
  + Recall that reliable broadcast imposes no particular order on delivering the messages, so every process simply stores the delivered messages in a set of unordered messages.
  + At any point in time, it may be that no two processes have the same sets of unordered messages in their sets.
  + The processes then use the consensus abstraction to decide on one set, order the messages in this set, and finally deliver them.

**What does that mean?**

While the consensus problem has attracted much attention in the theoretical distributed systems community, it has been largely ignored by systems implementers. Implementers usually consider the consensus problem to be irrelevant for real systems.

Such a claim stating that results applying to the consensus problem are irrelevant to other agreement problems is incorrect. The simplest example is atomic broadcast. The atomic broadcast problem and the consensus problem have been shown to be equivalent.

It must be noticed that the atomic broadcast problem is also subject to the FLP impossibility result and a solution for the consensus problem can be used as a building block to solve the atomic broadcast problem; therefore, the inherent difficulty of solving the consensus problem inevitably applies to the atomic broadcast problem independently of the solution that is used.

# Question 2

We can use timeout mechanism, which is based on purely local logical time. Logical time of process pi can be defined as the number of instructions that pi has executed. Most implementations assume implicitly a stronger system model. Typically, a LAN with a timeout of 30 seconds to detect crashed processes might adequately be modeled as a synchronous system.

# Question 3

There is an inevitable trade of between reducing the probability of incorrect failure suspicions, and fast reaction to process crashes. Consider atomic broadcast implemented using a timeout of 60 seconds to suspect the crash of the process. In this case, at least 60 seconds are needed to react to the crash of the sequencer process. In other word, the crash of the process will lead to a blackout period of at least 60 seconds. **This might be unacceptable for time critical applications.**

On the other hand reducing the timeout value increases the probability of incorrect failure suspicions.

# Question 4

There is no algorithm that solves consensus with a single communication step in good runs. However, such an algorithm cannot be correct. A process after having received a value *v* cannot be sure that the other processes also have received that value and decided on that as well. This means it is probable that the agreement property of the consensus problem to be violated.

However, one might be tempted to fix the problem of uncertainty about the value received and decided on by other processes, in a *time-based model*, by introducing a delay before deciding on value v. the delay is the duration needed to detect that a network failure might have occurred. After this time, if no network failure has been detected, then the process can decide on received value.

# Reference

Christian Cachin, Rachid Guerraoui, Lu´ıs Rodrigues. (2011). Introduction to Reliable and Secure Distributed Programming (Second edition). London: Springer.