CS-451 – Distributed Algorithms

Fall 2017 Final Exam

15.01.2018

Name: _____

Sciper number: _____

Time Limit: 3 hours (8:15am to 11:15am).

Instructions:

- This exam is closed book: no notes, electronics, nor cheat sheets allowed.
- Write your name and SCIPER on *each page* of the exam.
- If you need additional paper, please ask one of the TAs.
- Read through each problem before starting to solve it.
- When solving a problem, do not assume any known result from the lectures, unless it is explicitly stated that you might use some known result.

Good Luck!

Part	Max Points	Score
1	14	
2	14	
3	14	
4	14	
Total	56	

1 Broadcast (14 points)

1.1 Question 1 (9 points)

State the interface (properties and specification in terms of indications and requests), and implementation of a Causal Total Order Broadcast (CTOB) algorithm, i.e., a broadcast algorithm that satisfies both causality and total-order. Furthermore, explain why your implementation solves CTOB.
Hint: You can assume that you have access to a procedure that sorts messages deterministically and which respects the causal order among messages. You can assume to have access to any abstraction, except for Total Order Broadcast or CTOB itself.

Answer:
Properties: (2 points)

- **CTOB1**: *Validity:* If a correct process p broadcasts a message m, then p eventually delivers m.
- **CTOB2**: *No duplication:* No message is delivered more than once.
- **CTOB3**: *No creation:* If a process delivers a message m with sender p, then m was previously broadcast by process p.
- **CTOB4**: *Uniform Agreement:* If a message m is delivered by some process, then m is eventually delivered by every correct process.
- **CTOB5**: *Causal order:* If any process p_i delivers a message m', then p_i must have delivered every message m such that $m \rightarrow m'$.
- **CTOB6**: *Total order:* Let m and m' be any two messages. Let p_i be any process that delivers m without having delivered m'. Then no process delivers m' before m.

Specification: (1 point)
Module:

- **Name:** CausalTotalOrderBroadcast, instance ctob.

Events:

- **Request:** $\langle ctobBroadcast \mid m \rangle$: Broadcasts a message m to all processes.
- **Indication:** $\langle ctobDeliver \mid p, m \rangle$: Delivers a message m broadcast by process p.

Reasoning: (2 points)
If we replace URB with Causal URB in TOB, the local deliveries in *cubDeliver* will respect the causal order of the messages, i.e., if message $m \rightarrow m'$, m' is not delivered unless m has already been delivered. Therefore, m' cannot be proposed to Consensus (added to *pending*), unless m is proposed at the same time or m has already been *ctobDelivered*. Assuming that the sorting procedure is deterministic and respects the causal order of the messages, the messages are delivered in both a Causal and a Total Order manner.

Algorithm 1 Causal Total Order Broadcast. (4 points)

Implements:
CausalTotalOrderBroadcast (ctob).

Uses:
CausalUniformBroadcast (cub).
UniformConsensus (ucons).

Upon event $\langle Init \rangle$ **do**
1: $delivered := \varnothing$;
2: $pending := \varnothing$;
3: $wait := false$;
4: $sn := 1$;

Upon event $\langle ctobBroadcast \mid m \rangle$ **do**
1: **trigger** $\langle cubBroadcast \mid m \rangle$;

Upon event $\langle cubDeliver \mid p_i, m \rangle$ **do**
1: **if** $m \notin delivered$ **then**
2: $pending := pending \cup (p_i, m)$;

Upon $(pending \neq \varnothing)$ and $wait = false$ **do**
1: $wait := true$;
2: **trigger** $\langle uconsPropose \mid pending \rangle_{sn}$;

Upon event $\langle uconsDecide \mid decided \rangle_{sn}$ **do**
1: $pending := pending \setminus decided$;
2: $ordered := deterministicSort(decided)$; ▷ the sorting procedure respects the causal order of messages
3: **forall** $(p_i, m) \in ordered$ **do**
4: **trigger** $\langle ctobDeliver \mid p_i, m \rangle$;
5: $delivered := delivered \cup m$;
6: $sn := sn + 1$;
7: $wait := false$;

1.2 Question 2 (5 points)

Algorithm 2 implements a FIFO Uniform Broadcast (FUB) abstraction by using vector clocks. Algorithm 3, in turn, implements a Causal Order Uniform Broadcast (CUB) abstraction, using only FUB as an underlying module. Please answer the following questions:

1. Does the implementation presented in Algorithm 3 work correctly? Explain your reasoning for or against. If you think that the implementation is incorrect, then give modifications to make it correct. (3 points)

   Answer: Algorithm 3 does not work correctly, because the local message number *lsn* in the FIFO module is only incremented when a message is sent. Imagine an execution where a process p_i sends a message m, process p_j receives m, and immediately broadcasts m'. If for some reason m is delayed by the network, and a different process p_k receives m' before receiving m, p_k will *fubDeliver* and subsequently *cubDeliver* m', because the serial number of message m' is 1 and the vector clock value at process p_k for process p_j is also 1.

   A necessary modification to Algorithm 3 is to insert a line in the procedure *fubDeliver* that will resend the fubDelivered messages upon receipt: **trigger** $\langle fubBroadcast \mid m \rangle$. Resending all received messages increases the *lsn* of the current process, getting it ready for the next causal message. A receiving process will therefore not deliver a message, unless it has already delivered all messages that were sent (including resent) by the same process, since those will have lower serial numbers. This ensures causality.

2. Can the *delivered* vector be removed from Algorithm 3? (2 points)

   Answer: The *delivered* vector can safely be removed from Algorithm 3 in its current implementation, i.e., before the proposed modification. That is true because the FUB will only pass the messages to CUB whose serial number is the one that it expects (vector *next*). Furthermore, because of the *no duplication* property of Uniform Reliable Broadcast (UB), the same message will only be transmitted once from UB to FUB (we rely on UB's *delivered* vector). (Even if the *no duplication* property did not exist, FUB will continue to work properly, however, its *pending* vector will constantly increase in size if messages are resend.)

   After the proposed modification is made, the *delivered* vector cannot be removed from CUB, because each message will then be resent by all correct processes, thus there will need to be a mechanism of filtering out the *N-1* copies of each message, where N is the number of correct processes at a given moment.

Algorithm 2 FIFO Uniform Broadcast using vector clocks.

Implements:
FIFOUniformBroadcast (fub).

Uses:
UniformBroadcast (ub).

Upon event $\langle Init \rangle$ **do**
1: $pending := \varnothing$;
2: $lsn := 0$;
3: $next := [1]^N$;

Upon event $\langle fubBroadcast \mid m \rangle$ **do**
1: $lsn = lsn + 1$;
2: **trigger** $\langle ubBroadcast \mid (m, lsn) \rangle$;

Upon event $\langle ubDeliver \mid p_i, (m, sn) \rangle$ **do**
1: $pending := pending \cup (p_i, m, sn)$;
2: **while** $(p_i, m, sn) \in pending$ **such that** $sn = next[p_i]$ **do**
3: $next[p_i] := next[p_i] + 1$;
4: $pending := pending \setminus (p_i, m, sn)$;
5: **trigger** $\langle fubDeliver \mid p_i, m \rangle$;

Algorithm 3 Causal Order Uniform Broadcast based on FIFO Uniform Broadcast.

Implements:
CausalOrderUniformBroadcast (cub).

Uses:
FIFOUniformBroadcast (fub).

Upon event $\langle Init \rangle$ **do**
1: $delivered := \varnothing$;

Upon event $\langle cubBroadcast \mid m \rangle$ **do**
1: **trigger** $\langle fubBroadcast \mid m \rangle$;

Upon event $\langle fubDeliver \mid p_i, m \rangle$ **do**
1: **if** $m \notin delivered$ **then**
2: **trigger** $\langle fubBroadcast \mid m \rangle$;
3: **trigger** $\langle cubDeliver \mid p_i, m \rangle$;
4: $delivered = delivered \cup m$;

2 Consensus (14 points)

2.1 Question 1 (4 points)

Write the properties of Consensus with their description. *Answer.*

- Validity: Any value decided is a value proposed
- Agreement: No two correct processes decide differently
- Termination: Every correct process eventually decides
- Integrity: No process decides twice

2.2 Question 2 (5 points)

We assume a crash-free model where processes aim to solve Consensus by proposing and deciding on real values (from $\mathbb{R}$). We define malicious processes to be processes which can lie about what they propose. For instance, suppose process p is malicious, then p can propose value v. If Consensus decides on v, p can refuse its proposal of v; as a result, the validity property of consensus is broken. Robust Consensus fulfills the same properties of Consensus except validity, which is replaced by Robust Validity as described below:

Robust Validity: Any value decided is proposed by a non-malicious process, or the value is bounded between two values proposed by non-malicious processes.

Given a system with N processes, none of which may crash, out of which f are malicious, such that $f < \frac{N}{2} - 1$, devise an algorithm that implements Robust Consensus. You are allowed to use Best Effort Broadcast or any additional modules.

Answer. Take any algorithm from the course, replace ordering the values and taking the smallest/largest, by ordering the values and taking the median among all proposition.

2.3 Question 3 (5 points)

Consider any Consensus algorithm that is implemented using the Eventually Perfect Failure detector $\diamond P$ and relies on a majority of correct processes. Does this algorithm solve the Uniform variant or the Non-Uniform variant of Consensus? Explain your answer.

3 Shared Memory, Atomic Commit (14 points)

3.1 Question 2 (4 points)

State the properties of the following with a short description of each property.

- Non-Blocking Atomic Commit (2 points)
  Answer. Agreement: No two processes decide differently.
  Termination: Every correct process eventually decides.
  Commit-Validity: 1 can only be decided if all processes propose 1.
  Abort-Validity: 0 can only be decided if some process crashes or votes 0.

- ACID properties (2 points)
  Answer. Atomicity: a transaction either performs entirely or none at all.
  Consistency: a transaction transforms a consistent state into another consistent state.
  Isolation: a transaction appears to be executed in isolation.
  Durability: the effects of a transaction that commits are permanent.

3.2 Question 2 (6 points)

Answer the following questions:

- Is it possible to implement Atomic Commit with an Eventually Perfect Failure Detector if a process can crash? Briefly motivate your answer. (4 points)

  Answer. No.

- Is it possible to implement Atomic Commit with only a Uniform Consensus module if a process can crash? Briefly motivate your answer. (2 points)

  Answer. Yes.

3.3 Question 3 (4 points)

Implement an 1-N atomic register using an arbitrary number of 1-1 atomic registers.

Answer: Suppose p_0 is the only writer among the N processes to the 1-N atomic register. We consider a N by N matrix of 1-1 atomic registers. In row i and column j, p_i is the writer and p_j is the reader. For writing v in the 1-N atomic register, p_0 writes v with the time-stamp in all 1-1 atomic registers in row 0. For reading from the 1-N atomic register by p_i, p_i finds the value with the highest time-stamp in column i, named $v*$, and write $v*$ with the corresponding time-stamp to all the registers in row i. After all these writing finished, p_i reads $v*$.

4 Group Membership, TRB, View Synchronous Communication (14 points)

4.1 Question 1 (6 points)

Consider a system where processes which crash can transmit a "farewell" message to the remaining processes. An Augmented Failure Detector (AP) is a failure detection module that has this feature, and with the following interface:

Events:

- Indication: $\langle Crash \,|\, p,m \rangle$: Signals that process p crashed with farewell message m.
- Request: $\langle Farewell \,|\, m \rangle$: A process updates the farewell message which will be later transmitted (upon a crash) to other processes.

Properties:

AP1. Strong Completeness Eventually, every process that crashes is permanently suspected by every correct process.

AP2. Strong Accuracy No process is suspected before it crashes.

AP3. Validity If a process p is suspected, then the suspicion indication includes the last farewell message set by process p.

Answer the three questions below.

1. What is the difference between the Eventually Perfect Failure Detector and AP? (1 points)
   There are multiple differences: the $\langle Farewell \,|\, m \rangle$ request, the farewell message m in $\langle Crash \,|\, p,m \rangle$, the Accuracy property in AP is Strong (not Eventually Strong), as well as the Validity property.

2. If the system model is synchronous, can we implement the AP abstraction? If yes, briefly describe your implementation; is any module necessary? (2 points)
   An implementation could be using uniform reliable broadcast to ensure that the farewell message m is propagated to all processes when $\langle Farewell \,|\, m \rangle$ is called. Then a traditional failure detector is implemented, and when a process p is suspected, the locally stored farewell message for p is included in the $\langle Crash \,|\, p,m \rangle$ indication.

   The most important part of the implementation is that the request $\langle Farewell \,|\, m \rangle$ should not return to the caller (higher-level module) until message m has finished uniformly broadcasting to the other processes.

3. What if the system model is asynchronous? Briefly describe your implementation if it can be implemented. If any module is necessary in your implementation, give a proof that it is indeed necessary. (3 points)

   If the system is asynchronous, then properties AP1 and AP2 cannot be implemented, because there are no assumptions on how much time it takes for a process to timeout and become suspected. See the implementation details of failure detectors in the Introduction slides. For this reason, the AP abstraction cannot be implemented.

4.2 Question 2 (4 points)

The classic problem of Terminating Reliable Broadcast (TRB) is defined for a specific broadcaster process $pi = src$. In this classic model, all processes have knowledge of who src is. We redefine this problem to remove this assumption: what if the broadcaster is unique but unknown? Specifically, at initialization, any single process pj can be nominated as the broadcaster, e.g., through a parameter passed to the $\langle Init \rangle$ procedure. The identity of pi is initially unknown to all other processes except the nominated process.

We call this abstraction *AnyTRB*, having the same properties and interface as classic TRB, except that the request $\langle trbBroadcast \,|\, m \rangle$ can only be invoked by the nominated broadcaster. Below we show the *Init* procedure where a process checks if it was nominated as the broadcaster and stores this information in a local variable.

Algorithm 4 *Init* procedure for *AnyTRB*.

Upon event $\langle Init \,|\, src \rangle$ **do**

```
1: bcast = false;          ▷ Local variable to remember if we are the nominated source.
2: if src = true then      ▷ src can be true or false.
3:     bcast = true;       ▷ We are nominated as source.
```

Answer the two questions below.

1. Is it possible to implement *AnyTRB* using only Best Effort Broadcast, Consensus, and P? Describe your implementation if yes, or otherwise explain why is it not possible. (2 points)

   An implementation is possible as follows, informally. Each process broadcasts whether they are the nominated process or not. All processes wait until they hear from all the other processes (either their nomination status or a suspicion). If none of the alive processes was nominated, then all processes can deliver ϕ (this means that the nominated process *pi* crashed and was suspected, hence the other processes will never be able to deliver a message from *pi*); otherwise, the nominated process simply broadcasts its message and processes agree using Consensus to either deliver the message or ϕ.

2. Assume we are allowed to use the AP abstraction we defined at Question 1 earlier. Can we implement *AnyTRB*? Describe your implementation if yes and state if any additional modules are necessary besides AP; otherwise, explain why is it not possible. (2 points)

   This implementation is significantly simpler: processes set as their farewell message their nomination status as well as the message they wish to broadcast, and then also attempt to broadcast this information. Regardless of whether the nominated process *pi* crashes or not, all processes will deliver this information and consequently decide on delivering *pi*'s message. This implementation assumes that processes do not crash before setting their farewell message.

4.3 Question 3 (4 points)

Answer the two questions below.

1. Explain the purpose of the *block* events in the View Synchronous Broadcast abstraction. (2 points)

   In VS, the block events allow the abstraction to signal to the application that a new view is ready, such that the application stops initiating new broadcast messages, allowing the installation of a new view.

2. State the Accuracy property of Group Membership. Is this a safety or a liveness property? (2 points)

   Accuracy states that if some process installs a view (i,M) and $p \notin M$, then p has crashed. This is a safety property.