CS-451 – Distributed Algorithms

Fall 2017 Midterm Exam

SOLUTIONS

4th of December, 2017

Name: ____________________

Sciper number: ____________________

Time Limit: 1 hour and 45 minutes (3:15pm to 5pm).

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 (7 points)

Consider the following properties and specification of FIFO-order Uniform Broadcast:

Module:
Name: FIFOUniformBroadcast, instance fub.

Events:
Request: $\langle fubBroadcast \mid m \rangle$: Broadcasts a message m to all processes.
Indication: $\langle fubDeliver \mid p, m \rangle$: Delivers a message m broadcast by process p.

Properties:
FUB1: *Validity:* If a correct process p broadcasts a message m, then p eventually delivers m.
FUB2: *No duplication:* No message is delivered more than once.
FUB3: *No creation:* If a process delivers a message m with sender p, then m was previously broadcast by process p.
FUB4: *Uniform Agreement:* If a message m is delivered by some process, then m is eventually delivered by every correct process.
FUB5: *FIFO delivery:* If some process broadcasts message m_1 before it broadcasts message m_2, then no process delivers m_2 unless it has already delivered m_1.

Implement a non-blocking *FIFOUniformBroadcast* abstraction by using Uniform Broadcast.
Note: non-blocking algorithms never delay (block) messages from being delivered, even if their order was scrambled by the network.

Answer: Algorithm 1 shows a non-blocking FIFO-order Uniform Broadcast implementation based on the *past* vector.

Algorithm 1 Non-blocking FIFO Uniform Broadcast with the *past* vector.

```
Implements:
  FIFOUniformBroadcast (fub).

Uses:
  UniformBroadcast (ub).

Upon event ⟨Init⟩ do
1: $delivered := \varnothing$;
2: $past := \varnothing$;

Upon event ⟨$fubBroadcast \mid m$⟩ do
1: trigger ⟨$ubBroadcast \mid [past, m]$⟩;
2: $past = past \cup m$;

Upon event ⟨$ubDeliver \mid p_i, [past_m, m]$⟩ do
1: if $m \notin delivered$ then
2:     forall $n \in past_m$ do;
3:         if $n \notin delivered$ then
4:         trigger ⟨$fubDeliver \mid p_i, n$⟩;
5:         $delivered = delivered \cup n$;
6:     trigger ⟨$fubDeliver \mid p_i, m$⟩;
7:     $delivered = delivered \cup m$;
```

1.2 Question 2 (7 points)

1. What abstraction is implemented in Algorithm 2 below? (2 points)

   Answer: Algorithm 2 implements Reliable Causal Broadcast by using vector clocks.

2. If we replace Reliable Broadcast with Uniform Broadcast in Algorithm 2, do we obtain the Uniform version on the abstraction? Explain why or why not. (2 points)

   Answer: It is not possible to build Uniform Causal Broadcast by replacing the Reliable Broadcast with a Uniform Broadcast in Algorithm 2. This is because of the event $\langle xDeliver \,|\, self,\, m\rangle$ inside the event $\langle xBroadcast \,|\, m\rangle$. An execution that breaks uniformity, even when using Uniform Broadcast, is when a process $\langle xBroadcasts\rangle$ a message, it $\langle xDelivers \,|\, self,\, m\rangle$ (delivers the message from *self* to itself), and crashes before $\langle urbBroadcast\rangle$-ing it to other processes.

3. If your answer to the previous question was negative, how would you modify Algorithm 2, such that when we make the replacement stated above (at #2), we indeed get a Uniform implementation of the abstraction? (3 points)

   Answer: The necessary modifications to this algorithm are to remove line 1 from the events $\langle xBroadcast\rangle$ and $\langle rbDeliver\rangle$, and to have an additional clock for delivered messages from *self.* With the first two modifications, the Uniform Reliable Broadcast will only deliver a message if all running processes at the given time acknowledge the broadcast message. Therefore, if the process crashes while broadcasting its message, the message will either be delivered by all current processes, or none. This ensures uniformity. With the final modification we introduce a new clock that will locally count the number of delivered messages from *self.* Therefore, in the *deliver-pending* procedure, we will need to check the value of the existing vector clock for $p_k \neq self$, and of the new clock for $pk = self$. Its value is increased when a message from *self* is delivered.

Algorithm 2 Abstraction X

```
Implements:
    Abstraction X (x).

Uses:
    ReliableBroadcast (rb).

Upon event ⟨Init⟩ do
 1: forall $p_i \in S$
 2:         $VC[p_i] := 0$;
 3: $pending := \varnothing$;

Upon event ⟨xBroadcast | m⟩ do
 1: trigger ⟨xDeliver | self, m⟩;
 2: trigger ⟨rbBroadcast | [Data, VC, m]⟩;
 3: $VC[self] := VC[self] + 1$;

Upon event ⟨rbDeliver | $p_i$, [Data, $VC_m$, m]⟩ do
 1: if $p_i \neq self$ then
 2:         $pending := pending \cup (p_i, [Data, VC_m, m])$;
 3:         trigger deliver-pending;

Procedure deliver-pending is
 1: while $(p_s, [Data, VC_m, m]) \in pending$
 2:         forall $p_k$ : $(VC[p_k] \geq VC_m[p_k])$ do
 3:                 $pending := pending - (p_s, [Data, VC_m, m])$;
 4:                 trigger ⟨xDeliver | $p_s$, m⟩;
 5:                 $VC[p_s] := VC[p_s] + 1$;
```

2 Consensus (14 points)

2.1 Question 1 (9 points)

For each of the statements below, please answer with **YES** (if the statement is true) or **NO** (otherwise). Briefly motivate your answer.

1. It is possible to implement Consensus using only Best Effort Broadcast and a Perfect Failure Detector. (1.5 points)
   Answer: Yes – see Consensus algorithm I from class slides.

2. It is not possible to implement Uniform Consensus using only Best Effort Broadcast and a Perfect Failure Detector. (1.5 points)
   Answer: No – Consensus algorithm II from class slides is an example of such an implementation.

3. In the absence of a Perfect Failure Detector, any algorithm that implements Uniform Consensus requires a majority of processes to be correct. (1.5 points)
   Answer: Yes, to solve Uniform Consensus such an algorithm must assume at least a majority of correct processes. Without this assumption, the algorithm can run into situations where the set of processes are split in two equal parts A and B, and each half can consider the other half crashed (we can call this informally a "split-brain" syndrome). If processes in A are indeed crashed, then the other half B must still reach a decision so as to satisfy Termination – and that decision might be different than the decision taken by processes in A. In such a case, the processes in A disagree with processes in B. Note that this requirements does not only apply to Uniform Consensus but also to Consensus.

4. The requirement from the previous question (#3) – regarding a majority of correct processes when a Perfect Failure Detector is not available – equally applies to Consensus (non-Uniform). (1.5 points)
   Answer: Yes. The same argument as the solutions for #3 applies.

5. An execution trace of a Consensus algorithm (i.e., a per-process time-line with *propose* or *decide* events as you have seen in the class) can demonstrate how the algorithm breaks the Termination property. (1.5 points)
   Hint: Termination states that every correct process eventually decides.
   Answer: No. Termination is a liveness property. An execution trace can possibly demonstrate that Termination is satisfied, but not that it is broken.
   Alternative answer accepted: Yes, assuming that all the decision events of correct processes are captured.

6. An execution trace of a Uniform Consensus algorithm (i.e., a per-process time-line with *propose* or *decide* events as you have seen in the class) can demonstrate how such an algorithm does not satisfy the Validity property. (1.5 points)
   Hint: Validity states that any value decided is a value proposed.
   Answer: Yes, the trace can demonstrate that Validity is broken (assuming that all the decision events of processes are captured).

2.2 Question 2 (5 points)

Consider the *Suspicious Consensus* abstraction with the following interface and properties:

Events:

- Request: $\langle \textit{Propose} \mid v \rangle$: Proposes a value v for consensus.
- Indication: $\langle \textit{Decide} \mid v \rangle$: Outputs a decided value v of consensus.
- Indication: $\langle \textit{Suspect} \rangle$: Signals the suspicion of some (unknown) process in the system.

Properties:

C1. Validity Any value decided is a value proposed.

C2. Decision Agreement No two correct processes decide on different values.

C3. Termination Every correct process eventually decides.

C4. Integrity No process decides twice.

C5. Suspicion Integrity No process triggers $\langle \textit{Suspect} \rangle$ unless some process crashed.

Answer the questions below.

1. What are the differences between the Consensus abstraction and the Suspicious Consensus abstraction? (2.5 points)

   Answer: Suspicious Consensus is a naive extension to Consensus, providing an additional safety property, namely Suspicion Integrity, and the event $\langle \textit{Suspect} \rangle$.

2. Is it possible to implement Consensus as you saw it in class if the only module you are allowed to use is Suspicious Consensus? (2.5 points)
   If yes, sketch your implementation below. Otherwise, explain why this is not possible.

   Answer: See Algorithm 3 below. The solution is trivial. This is because Suspicious Consensus is a naive extension to Consensus.

Algorithm 3 Consensus using Suspicious Consensus *sc*.

```
Upon event $\langle Propose \mid v \rangle$ do
  1: trigger $\langle scPropose \mid v \rangle$
Upon event $\langle scSuspect \rangle$ do
  1: $\perp$
Upon event $\langle scDecide \mid v \rangle$ do
  1: trigger $\langle Decide \mid v \rangle$
```

3. Is it possible to implement Suspicious Consensus if the only module you are allowed to use is Consensus? Explain how this can be achieved (if yes), or why this is impossible (if not).

   Answer: Yes, albeit this implementation of Suspicious Consensus would never trigger the $\langle \textit{Suspect} \rangle$ event because there are no means to detect process failures. Note that it is OK if this event is never triggered: Suspicion Integrity is a safety property, so such an implementation would still satisfy all the properties of Suspicious Consensus. The implementation is straightforward and similar to Algorithm 3 except it has no handler for the $\langle \textit{Suspect} \rangle$ event.

3 Shared Memory, Atomic Commit (14 points)

3.1 Question 1 (3 points)

Answer the following questions:

1. What is a transaction? (1 point)
   Answer: A transaction is an atomic program describing a sequence of accesses to shared and distributed information. A transaction can be terminated either by committing or aborting.

2. What are the ACID properties? (2 point)
   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)

1. Give an execution of NBAC, in which either of the two possibilities (0 or 1) is a valid decision. (1 point)
   Answer:
   P1: propose(1)—X
   P2: propose(1)——decide(0-1)

2. Change the Commit-validity and Abort-validity of NBAC to the following:
   Commit-validity: 1 is decided if all processes propose 1.
   Abort-validity: 0 is decided if a process proposes 0.
   Is there an execution , in which the outputs of NBAC and the modified NBAC are different? Explain. (1 point)
   Answer: Yes. These changes forced the P2 in the above execution to decide 1 but the output of NBAC can be 0.

3. Is it possible to implement the modified version of NBAC with the above Commit-validity and Abort-validity using BestEffortBroadcast (beb), PerfectFailureDetector (P), and UniformConsensus (uniCons)? If so, give the implementation. If not, elaborate why. (4 points)
   Answer: No. Consider a execution in which all the processes including a faulty process propose 1. Hence, all the correct process should decide on 1. However, the proposal of the faulty process may never reach to the correct process, so they do not know if the faulty process had proposed 0 or 1 before crashing. This prevents correct processes from ever reaching a decision, which is against the termination property of NBAC.

3.3 Question 3 (5 points)

In the class, you saw an implementation of a single N-N atomic register with the following procedure:
Module:
Name: N-N-AtomicRegister, instance *r*.

Procedures:
Write(v): Writes new value v in register r.
Read(): Reads the value of register r.

The aim of this question is to implement the following three procedures for an atomic array of M registers with N writers and N readers using the abstraction of atomic registers. (N is the number of the processes and M is the number of registers.)

1. **Read(i):** Reads the value in register r_i, i.e. the register with index i. (1 point)
2. **Write(i, v):** Writes new value v in register r_i, i.e. the register with index i. (1.5 points)
   Answer: In these cases, we can have M parallel implementation of N-N atomic register. So, only we need to add index to the code in class to refer to each of these M registers. This is a valid solution as to guarantee the serialization, we can consider read and write operation in each single register separately.
3. **Write**($s, i_1, v_1, ..., i_s, v_s$)**:** Atomically writes new values $\{v_j\}_{j=1}^{j=s}$ in registers with indices $\{i_j\}_{j=1}^{j=s}$ (i.e. $\{r_{i_j}\}_{j=1}^{j=s}$), accordingly. Here, s is the number of registers which are affected by this write. For example, W(2,1,2,3,4) means writing atomically in 2 registers: write 2 in register 1 and write 4 in register 3. Hence, in Figure 1, process 3 should read the new value in register 3, i.e. 4. (2.5 points)
   Answer: In this case, the order for serialization of all reads and write for all the registers should be the same. To assure serialization, for any read or write operation, we read from all the registers and write to all the registers. So, we consider an array of integer value related to registers and assign a timestamp to that array. So, for each write, we first read the values of all the registers and then write the new values for the one which should defined in the write operation, and write the read value for the others assigned to a new timestamp.

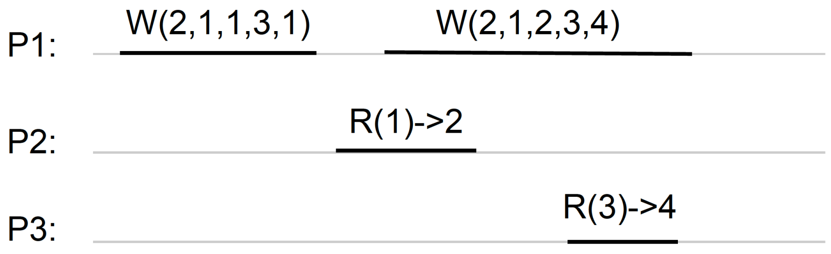

Figure 1: simultaneous read and write in an atomic m-register

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

4.1 Question 1 (7 points)

1. Give the specifications of Terminating Reliable Broadcast (*2 points*)
   Answer: See the class slides.

2. Show that the perfect failure detector is necessary to implement an algorithm for TRB. (*2 points*)
   Answer: As seen in the class, TRB can be used to implement P.

3. Explain the difference between the **Agreement** and the **Uniform Agreement** properties in the context of Terminating Reliable Broadcast (TRB). (*3 points*)

   Answer: In the non-uniform version of agreement, a process may deliver m (or φ) and then crash, without requiring correct processes to also deliver m (or φ).

4.2 Question 2 **(7 points)**

1. Give the specifications of View Synchronous Communication (VSC) (*2 points*)
   Answer: See the class slides.

2. Some properties of View Synchronous Communication (VSC) do not *allow* the joins of new processes. Briefly explain why. (*2 points*)

   Answer: Joins would break local monotonicity.

3. Some other properties of View Synchronous Communication (VSC) do not *accommodate* the joins of new processes. Briefly explain why. (*3 points*)

   Answer: Completeness and accuracy only refer to crashes, they do not impose any specific condition on the correctness of joins.