Distributed Algorithms, Final Exam

January 14, 2013

Solution

1 Multiple Choice Questions (15 points)

Question 1. (2 points) 1, 3

Question 2. (2 points) 1, 2, 3

Question 3. (2 points)

Question 4. (2 points)

Question 5. (2 points)

Question 6. (5 points)

1. L
2. S
3. L
4. S
5. S

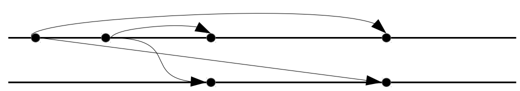
2 Reliable Broadcast (13 points)

Question 1. (1 point) Slides or book.

Question 2. (6 points)

1. Uniform reliable broadcast – No

2. Causal broadcast – No



3. Terminating reliable broadcast – No

Question 3. (6 points) This is a best-effort causal broadcast abstraction. Accordingly, on a crash-free execution (all processes are correct) agreement is guaranteed due to validity.

Of course, the crash-free case is not that interesting, so lets discuss what happens in case there are crashes. What happens when a message is broadcast? A broadcast of message m by process p enforces all other processes to receive all the messages that belong to the causal past of m[1]. This, of course, includes the messages that were delivered and the messages that were broadcast by p before message m. It should be clear that this is a direct consequence of the definition of causality:

A message m_1 causally precedes a message m_2 ($m_1 \rightarrow m_2$) when:

1. both are broadcasts of the same process and m_1 was broadcast before m_2

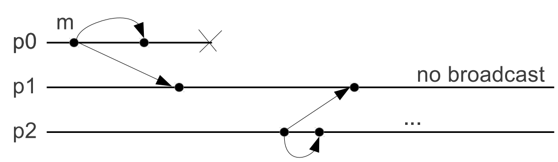
[1]if p does not crash while broadcasting m, case that could lead to m not being delivered by every process

2. m_1 is a broadcast of p_1 and m_2 is a broadcast of p_2 and m_2 was sent after p_2 delivered m_1

3. $m_1 \to m'$ and $m' \to m_2$ entails $m_1 \to m_2$ (transitivity)

So, in an execution where the correct processes keep broadcast messages, the causal delivery property ensures that all the delivered messages of a process before m will be delivered before delivering m, ensuring agreement even in the case of crashes.

However, we cannot guarantee that every process will send an infinite number of messages, so the following execution is possible:



As you can see, p_1 delivers message m sent by p_0 just before p_0 crashed. Due to the crash, m was not delivered by p_2. If p_1 stays inactive (as it happens in the execution above), p_3 is not guaranteed to deliver message m, hence violating agreement.

Consequently, the broadcast algorithm of the question **does not guarantee the agreement property** in executions that there are crashes.

3 View Synchronous Communication (10 points)

Question 1. (4 points) See the lecture's slides or the book.

Question 2. (4 points) See algorithm 1 for the solution and table 1 for the grading scheme.

Table 1: Grading scheme for question 3.2

A	Acts as RB when no stop event is triggered	0.5 point	
B	Does not deliver any message after emitting stopOk	0.5 point	
C	Understands that a process cannot just stop by itself: a process does not emmit stopOk before it is sure that all other processes requested to stop or failed.	1.5 point	
D	Gathers and delivers the messages delivered by the correct processes in order to guarantee agreement	1 point	
E	Handles failures correctly and with TRB only (no need for a failure detector)	1 point	
F	Uses one instance of TRB per process	0.25 point	
G	Got all the details right	0.25 point	

Algorithm 1 Implementation of Stoppable Broadcast using UTRB

```
Implements:
    StoppableBroadcast, instance sb
Uses:
    UniformTerminatingReliableBroadcast, instances utrb.p_i with sender p_i ∈ Π
    ReliableBroadcast, instance rb
upon event ⟨sb, Init⟩ do
    delivered ← ∅
    trbdone ← ∅
upon event ⟨sb, Broadcast | m⟩ do
    trigger ⟨rb, Broadcast | m⟩
upon event ⟨rb, Deliver | p, m⟩ do
    if (p, m) ∉ delivered then
        delivered ← delivered ∪ {(p, m)}
        trigger ⟨sb, Deliver | p, m⟩
    end if
upon event ⟨sb, Stop⟩ do
    trigger ⟨utrb.p_i, Broadcast | delivered⟩
upon event ⟨utrb.p_i, Deliver | p_i, m⟩ do
    trbdone ← trbdone ∪ {p_i}
    if m ≠ ϕ then
        forall (s, m') ∈ m' do
            if m' ∉ delivered then
                delivered ← delivered ∪ {m'}
                trigger ⟨sb, Deliver | s, m'⟩
            end if
    end if
upon event trbdone = Π do
    trigger ⟨sb, StopOk⟩
```

Question 3. (4 points) See algorithm 2 for the solution and table 2 for the grading scheme.

Table 2: Grading scheme for question 3.3

A	Understood that one instance of SB per view is needed for liveness	1.5 point	
B	Understood that SB guarantees that all processes that stop deliver the same set of messages	1.5 point	
C	Handles multiple concurrent view changes correctly	1 point	
D	Used and understood Block and BlockOk (indication or request, no broadcast comes after BlockOk)	0.25 point	
E	Understood Stop and StopOk (indication or request, no broadcast should be emitted after Stop)	0.25 point	
H	Got all the details right	0.5 point	

Algorithm 2 View-Sychronous Communication using Stoppable Broadcast

```
Implements:
    ViewSynchronousCommunication, instance vs
Uses:
    StoppableBroadcast, instances sb.i, i ∈ ℕ
    GroupMembership, instance gm
upon event ⟨vs, Init⟩ do
    viewId ← 0
    changingView ← false
    nextView ← ⊥
upon event ⟨vs, Broadcast | m⟩ do
    trigger ⟨sb.viewId, Broadcast | m⟩
upon event ⟨sb.i, Deliver | p, m⟩ such that i = viewId do
    trigger ⟨vs, Deliver | p, m⟩
upon event ⟨gm, View | v⟩ such that nextView = ⊥ do
    nextView ← v
upon event nextView ≠ ⊥ and changingView = false do
    changingView ← true
    trigger ⟨vs, Block⟩
upon event ⟨vs, BlockOk⟩ do
    trigger ⟨sb.viewId, Stop⟩
upon event ⟨sb.viewId, StopOk⟩ do
    viewId ← nextView.id
    changingView ← false
    trigger ⟨vs, View | nextView⟩
    nextView ← ⊥
```

4 Shared Memory (8 points)

Question 1. (2 points)

1. not safe

```
P1 [  W(1)  ]
P2             [ R()->0 ]
```

2. safe, but not regular

```
P1 [  W(1)  ]      [ W(2) ]
P2             [ R()->0 ]
```

3. regular, but not atomic

```
P1 [  W(1)  ]   [     W(2)    ]
P2             [ R()->2 ]
P3                         [ R()->1 ]
```

4. atomic

```
P1 [  W(1)  ]   [     W(2)    ]
P2             [ R()->1 ]
P3                          [ R()->2 ]
```

Question 2. (6 points)

1. The solution in the course has each reader also write its value to every process before returning its value. This ensures that nobody has an older version than the reader. However, due to the fact that readers can also issue writes, the processes need to check the timestamp before accepting a write.

   The algorithm in the exam has the readers only check the timestamp of other processes before returning. Since only the Writer process can issue write commands, there is no longer the need to check the timestamp when accepting a write command — the timestamp of new writes is guaranteed to be newer, since the writer ensures increasing timestamps. However, as shown below, the algorithm in the exam does not solve 1-N atomic registers.

2. The algorithm does not solve the 1-N atomic register problem.

   By reading the timestamps from everyone, the algorithm can indeed detect potential concurrent writes. However, the versioning implemented by the algorithm is not complete. The algorithm only holds two versions of data, "new" and "old", which is not enough. Since only a successful Read sets val_{old}, it might happen that the value of val_{old} is very old. The issue is shown in the execution below.

```
Writer   [  W(1)  ]           [     W(2)    ]
Reader1                         [ R()->0 ]
Reader2             [R()->1]      [ R()->1 ]
```

Suppose all registers are initialized to 0 ($val_{new} = val_{old} = 0$). There are two writes issued. The first write is not concurrent with any other operation and terminates successfully. Thus, both $Reader1$ and $Reader2$ will store 1 in val_{new} and 1 in ts_{new}. The first read by $Reader2$ completes successfully and set val_{old}, val_{new}, ts_{old} and ts_{new} to 1. Then, during the second write, $Reader1$ receives the new value before attempting the read, thus val_{new} is 2 and ts_{new} is 2. Notice that, val_{old} and ts_{old} are still 0 for Reader1. However, $Reader2$, does not yet receive the new values by the time Reader1 starts reading. $Reader1$ issues a read, detects that $Reader2$ has old values and thus returns val_{old}. However, val_{old} is still 0 at this point, since Reader1 never had a successful non-concurrent read. This execution does not respect an atomic register specification (and not even a regular register specification).