

CSE 223B Distributed Computing and Systems
Spring 2009

Name: SOLUTIONS

This exam has four questions. You have 80 minutes to answer all of them. Please write your name on this cover sheet and at the top of each page of the exam. Answer each question on its own page; you may use the back if you need extra space.

You will likely find some questions harder than others. It is suggested that you read through them all and attempt them in the order that will allow you to complete as much of the exam as possible. If you find any of the questions ambiguous, please write down any assumptions you make in order to answer the question.

This exam is open book; you may refer to your notes and any of the papers we have read in class. You may not, however, consult additional resources or use any electronic equipment.

Question	Score	Points
1		25
2		25
3		25
4		25
Total		100

1. (25 points) **Synchronization (TreadMarks)**

Answer the following questions based on the description of the TreadMarks system in the paper “TreadMarks: Distributed Shared Memory on Standard Workstations and Operating Systems” by Keleher, Cox, Dwarkadas, and Zwaenepoel. Consider the following distributed TreadMarks program, where `l1` and `l2` are locks, and `acq()` and `rel()` are the functions used to acquire and release locks, respectively. Assume all variables are initially set to zero.

```

      [1,0,0]          [2,0,0]  [3,0,0]          [4,0,0]
CPU 1: acq(l1); x = 1; rel(l1); acq(l2); y = 1; z = 2; rel(l2);
      [4,1,0]          [4,2,0]
CPU 2: x = 2; acq(l2); x = 3; z = 1; rel(l2);
      [2,0,1]          [4,2,2]          [4,2,3]          [4,2,4]
CPU 3: acq(l1); print x; acq(l2) print y; rel(l2); print z; rel(l1);

```

- A) (10 points)** Following *lazy release consistency* as implemented in TreadMarks, list the possible set of outputs from the distributed program above. There are no node failures, no messages are lost in the network, and no garbage collection. For example, if CPU 3 executed first, the program might output 0 0 0 .

```

0 0 0      0 0 1
1 1 2      1 0 0
1 1 1      1 0 1
3 1 2

```

- B) (10 points)** TreadMarks uses vector timestamps to order intervals with respect to each other. Suppose the output of the previous program was 1 1 1. For each `acq` and `rel` statement, show the local CPU’s vector clock. Assume all local clocks start at 0, so, for example if CPU 3 executes first, the clock at the `acq(l1)` statement would be `[0, 0, 1]`. For clarity, please write in the timestamps directly above the corresponding statements in the code listing above. **See above.**

- C) (5 points)** TreadMarks communicates only write notices initially, and calculates diffs upon request. Why? If TreadMarks proactively sent the updated pages along with the write notices, would it still be *lazy* release consistency? Regular release consistency? Or something else entirely?

TreadMarks calculates diffs simply to save bandwidth, which stems from two separate causes: first, diffs are often smaller than pages themselves. Secondly, since diffs are sent on-demand, changes that are not read need not be communicated. Proactively sending full pages along with/instead of write notices would not change the semantics. In particular, nodes still do not learn of write notices (or pages) until they go to acquire the lock, so the system still provides *lazy* release consistency.

2. (25 points) **Broadcast Ordering (ISIS)**

Birman and Joseph describe the ISIS toolkit in “Reliable Communication in the Presence of Failures.” ISIS provides causally and totally ordered communication support (CATOCS) through two different types of ordered broadcast in the presence of packet loss and unbounded delay:

- ABCAST: totally ordered atomic broadcast
- CBCAST: causally ordered broadcast

A) (10 points) Using at most three servers, show a sequence of correctly processed ABCAST messages that do not result in a causal ordering. In other words, some node displays a message before one that causally precedes it. Be sure to specify the responses and eventual ordering of each message in the ABCAST queues.

A broadcasts m to B and C
A broadcasts m' to B and C *// m' is causally ordered after m*
B receives m and responds with $B.0$
B receives m' and responds with $B.1$
C receives m' and responds with $C.0$
C receives m and responds with $C.1$
A broadcasts m' is assigned $B.1$
A broadcasts m is assigned $C.1$

All hosts display the messages as m' , m , the *inverse* of causal order.

B) (5 points) ISIS defines a group broadcast mechanism, GBCAST, which orders messages with respect to both ABCAST and CBCAST messages. The authors claim GBCAST can be used to deliver node failure messages. Describe why failure messages must be consistently ordered with respect to all other messages. In particular, what would happen if they were not?

ISIS requires that all messages sent by a node are received before the node fails; otherwise messages from failed nodes are rejected with a “you are dead” message. If a CBCAST message, say, from a node is reordered with respect to the GBCAST announcing its failure at two other nodes, one will accept the the CBCAST message, while the other will discard it, breaking the atomic broadcast semantics.

- C) (10 points) Despite having the GBCAST primitive which is used to broadcast node failures or recoveries, ISIS also defines a separate view change protocol. What additional functionality is provided by the view change protocol? Why can GBCAST not be used to provide it?

GBCAST, like any other atomic broadcast, is either received by all members of a view or none of them. If a node fails, it is by definition impossible to send any further messages to the view. Hence, a separate view change operation is required to allow the formation of a (single) new group in which to deliver the failure notification GBCAST along with any other messages.

In addition to enabling continued deliver of messages, the view change protocol provides a number of features, including notifying all nodes of the current view membership, and using a quorum-based consensus to ensure that only one view exists at a time, despite potential network partitions.

3. (25 points) Viewstamped Replication (Harp)

For this question, consider the Harp system described in “Replication in the Harp File System” by Liskov, Ghemawat, Gruber, and Johnson, Shriram, and Williams. Assume Harp is configured to share files across five servers, A , B , C , D , and E , with $n = 2$, and that A is the primary in the initial view, with B and C serving as backups and D and E as witnesses. For each of the following sequence of events, describe:

- i. Can the set of currently live servers form a new view and continue serving client requests? If a view cannot be formed, specify why and how the servers can tell.
- ii. Which nodes make up the view and which (if any) are promoted or demoted (including nodes that are just entering the view).
- iii. What data (if any) needs to be sent to the new nodes. From which server is it likely to be taken?

For example, if server C crashed, you might answer:

- i. Yes.
- ii. D or E is promoted to a backup.
- iii. B would send all log entries after the GLB.

A (5 points) Suppose that while C was still down, server A becomes partitioned from B , D and E .

- i. Yes, B , D , and E form a view.
- ii. B becomes primary; D or E (whichever was not already a backup) is promoted to a backup.
- iii. B would ensure D and E have all entries in the log up to the GLB.

B) (5 points) Server C reboots to join A .

- i. No new view (only two nodes); the previous one continues to operate.
- ii. N/A
- iii. N/A

C) (5 points) Server *B* crashes, and reboots in the partition with *A* and *C*. Unfortunately, its disk was lost.

i. No view is formed, as *D* and *E* alone are too few, and *B* is completely uninitiated, as *A* and *C* were not previously in a view so cannot confidently admit *B*. (To do so would be a mistake, because it would lose information from the previous view that is only at *D* and *E*.)

ii. N/A.

iii. N/A.

D) (5 points) Whatever bug *B* has was contagious, and *C* crashes again, rebooting in the partition with *D* and *E*, the contents of its disk intact.

i. Yes, *C*, *D*, and *E* form a view, as all three of them have state from their last view, and one of them (*D* and *E*, in this case) must have been in the most recent view by the quorum guarantee.

ii. *C* is primary, one of *D* or *E* remains backup.

iii. *D* or *E* will communicate the log up to the GLB to *C*.

E) (5 points) Finally, the partition is healed, and all five nodes can reach each other again.

i. Yes, all nodes will form a view.

ii. All nodes will return to their originally assigned roles.

iii. *B* will need to get the disk contents from *C*, while *A* and *B* can get the log up to the GLB from any of *C*, *D*, or *E*; likely one of the latter two.

4. (25 points) **Distributed File Systems (Frangipani)**

Thekkath, Mann, and Lee describe a filesystem layered on top of the Petal distributed block store in “Frangipani: A Scalable Distributed File System.” Frangipani attempts to provide similar semantics to Ivy, so it’s not surprising that it shares a number of similar mechanisms.

A) (5 points) Frangipani uses single-writer/multiple-reader locks like Ivy. In both cases, nodes must invalidate any stored copies of pages/blocks when they release the lock. In the case of Ivy, the manager issues invalidation requests to each node in the copy set, and a copy of the page is sent by the owner to the requesting node. Instead, Frangipani flushes dirty pages to Petal, where other nodes can retrieve them. But Frangipani nodes have no notion of a copyset; is this information still needed? If not, why not? If so, where is the analogous structure stored?

Yes, the information is still needed to prevent inconsistent views on the data. The lock service is in charge of maintaining the copy set, and will cause the other Frangipani nodes to invalidate their cached pages when revoking their locks.

B) (10 points) Unlike Ivy, Frangipani locks are actually leases: they need to be renewed every so often or they expire. What does Frangipani gain from this additional requirement? In particular, describe a situation that Frangipani gracefully handles that Ivy would be unable to recover from.

Frangipani gains robustness to failure. If an Ivy node crashes while holding a lock, that page cannot be accessed by anyone in system. In Frangipani, on the other hand, crashed nodes will time out, the recovery node will replay the log, and then release the log, allowing progress to be made.

- C) (10 points)** Some file system operations require modifying multiple on-disk structures, such as a file move operation. Each structure has its own lock, so completing such an operation requires holding multiple locks simultaneously, say locks 1 and 2. Suppose a node A grabs lock 1, but is waiting on node B to release lock 2. Node B fails before releasing the lock. While waiting for node B to recover, node A also fails. In a nightmare scenario, both recovery operations could be handed to the same server, C, which might try to recover the failure of node A first. Can this result in deadlock? Why or why not?

No, deadlock cannot occur, because the recovery daemon only replays actions in the log. Since node A did not succeed in acquiring both locks, it did not actually perform any actions that required both locks—the pending operation will simply not commit, and the client will need to retry. (Note the client never received a reply indicating it was done, so this is consistent with a lost request packet, for example.) So the recovering node will simply replay A's log—which is guaranteed to succeed since it is holding all the locks A needed for operations it committed—and then release lock 1. Node B's log can be recovered subsequently with no harm done.