

CSE 223B Distributed Computing and Systems
Winter 2008

Name: _____

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) Viewstamped Replication (Harp)

One of your fellow 223B students, Ben Bitdiddle, missed class the day we discussed “Replication in the Harp File System” by Liskov, Ghemawat, Gruber, and Johnson, Shriram, and Williams. He’s confused about a couple of points, and was hoping you could clarify them for him.

A) (5 points) Ben understands that Harp keeps a redo log, and maintains four different pointers into the log. The purpose of the CP seems obvious, the AP less so. Give Ben three reasons why the primary maintains an AP separate from the CP.

- 1. By allowing the AP to trail the CP, the primary can respond to the client without actually having to commit operations to disk, decreasing the response time.*
- 2. Once decoupled from the CP, the AP can advance at arbitrary speed. Harp leverages this capability to batch writes, increasing the throughput of the disk.*
- 3. By separating the AP from the CP, the primary and backup need not push updates to disk simultaneously, which may allow them to avoid simultaneous software failures. Harp insures that the AP of the backups trail the AP at the primary for precisely this reason.*

B) (5 points) The GLB also seems odd. Ben observes that the only time the entries between the GLB and LB are useful is in the event a node fails and loses its log, but not the contents of its disk. Moreover, each node in Harp is plugged into UPS, which provides enough time to write the log to disk in the case of power failure. Ben is puzzled: what could lead to such a failure? Describe a particular sequence of events that would require a node to communicate entries between its GLB and LB to another node.

If a node crashes due to a software bug the UPS will not help. In that case, a node may lose its log but not its disk, requiring it to learn about log entries up until its previous LB, which may or may not be lower than another node’s LB. If the crashed node had the lowest LB, the node assisting in recovering will send all the entries from the GLB forward.

- C) (5 points)** While carefully reading the paper, Ben noticed that the backup uses cumulative acknowledgements, implying it cannot process events out of order. Ben doesn't see why this is important—as long as the records are not lost, he can't see why it matters in what order they're stored. Help him out: In particular, describe a sequence of events that would lead to incorrect behavior if the order of two event entries were reversed in the backup's log with respect to the primary's log.

If a backup records two modifications to the same disk block in opposite order from the primary, a client may see inconsistent results from subsequent reads. In particular, the client may first read that block at the primary, seeing the second update. Afterwards, the primary crashes, and the backup becomes primary in the next view. If the client subsequently reads the same block in the new view, it will see the results of the first update, as that will be the "most recent" according to the new primary (the old backup).

- D) (10 points)** After your explanations, Ben now thinks he's got the hang of things. In particular, he's figured out a way to improve Harp. One of the more mysterious features of Harp is the use of a witness, which typically performs no actions. Ben correctly observes that the witness does not even need to begin logging events until after it is promoted. This seems silly to him, and he suggests that there is no need to designate a witness ahead of time. In particular, Ben proposes that a primary and backup should run alone in the normal case. If one of them detects failure of the other, they can then recruit any available node (assuming there are lots of spare machines laying around) to become a promoted witness in the next view; once the primary and backup once again establish they are both alive and reachable, the promoted witness can provide any missing log entries and a new view formed with the original primary and backup as before. Ben is quite proud of himself, but it seems unlikely the authors would have missed such an obvious optimization. Describe a sequence of events that would be properly handled by Harp, but would lead to incorrect behavior with Ben's optimization.

Suppose the network partitions so that the primary and backup cannot reach each other. Both will recruit new nodes to serve as promoted witnesses, and there will be two simultaneous views. Once the partition heals, it will be impossible to reconcile the state at the primary and backup in order to form a new view.

2. (25 points) **Memory consistency (Ivy/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.

- A) (10 points)** Assuming *sequential* consistency as implemented by IVY, list the possible set of outputs from the following distributed program. Assume all variables are stored on distinct pages and initially set to 0.

```
CPU 1: acq(l1); x = 3; rel(l1); acq(l2); y = 6; z = 4; rel(l2);
CPU 2: acq(l1); z = 9; rel(l1);
CPU 3: acq(l2); print x, y; rel(l2); print z;
```

```
0 0 0 // CPU 3, ...
0 0 4 // CPU 3, CPU 1, CPU 3, ...
0 0 9 // CPU 3, CPU 2, CPU 3, ...
3 0 0 // CPU 1, CPU 3, ... CPU 2 ...
3 0 4 // CPU 1, CPU 3, CPU 1, CPU 3, CPU 2
3 0 9 // CPU 1, CPU 3, CPU 2, CPU 3, ...
3 6 4 // CPU 1, CPU 3, ...
3 6 9 // CPU 1, CPU 3, CPU 2, CPU 3, ...
```

3 6 0 cannot happen due to the locking of l2.

- B) (10 points)** Now, considering *lazy release consistency* as implemented in TreadMarks, list the possible set of outputs from the distributed program above, assuming garbage collection *does not* occur.

0 0 0 or 3 6 4. Other combinations are possible only if garbage collection occurs. Recall that TreadMarks sends all updates, not just those within a particular acquire/release pair, so CPU 3 will receive the $x = 3$ update from CPU 1 along with $y = 6$ and $z = 4$.

- C) (5 points)** TreadMarks keeps a complete history of write notices (up until garbage collection, anyway), as opposed to simply keeping the most recent data. Why? Could we implement LRC without it?

By keeping a history, it can compute precisely the diffs asked for by a node, which can reduce the communication requirements. LRC could be implemented by only keeping the most recent versions, but the entire pages would need to be shipped around (since a node may not be able to compute a diff from the version the requester is currently holding, likely leading to a less efficient implementation.

3. (25 points) **Two-phase commit**

For this question, we will consider the QuickSilver system as described in “Recovery Management in QuickSilver,” by Haskin, Malachi, and Chan.

- A) (10 points)** Section 4.2 describes four possible vote responses, including vote-commit-volatile and vote-commit-recoverable. The authors observe that if no servers respond with vote-commit-recoverable, the TM (transaction coordinator) does not log any record of the transaction. This seems worrisome—what if the TM crashes before completing the transaction? Will the transaction eventually commit or abort? Why is that safe?

There are two cases. Either the TM crashes before issuing any commits, or it issues some commits and then crashes. In the first case, the transaction will eventually time out and abort, and all parties will agree. In the second case, the transaction will again eventually time out and abort, as the TM will have no record of committing it. The issue is that some parties may have received a commit. These nodes, however, have no 'recoverable' state, so there's nothing for them to 'rollback': in other words, even if they had correctly received an abort instead of the commit they did receive, they wouldn't have done anything differently! Hence, this partial abort is safe in this case.

- B) (15 points)** In two phase-commit, we've seen that sometimes the nodes need to block to ensure correctness. Suppose a transaction coordinator is managing a transaction between three servers, *A*, *B*, and *C*. Like our banking example from class, *A* and *B* are debiting and crediting accounts, respectively, while *C* is simply keeping an audit log of the transaction. Normally, the TC would issue “prepare” messages to all three, and, upon receiving “prepare-ok” messages in response, issue a “commit.” Suppose, instead, each of the following scenarios occurred. In each instance, say whether or not the proposed action would be safe, and explain why or why not.

- i.** *B* responded with a “prepare-ok,” but has not received a commit message. In the mean time, it learns *A* received a commit. Can *B* commit as well? *C* is, after all, just logging the transaction. Why or why not?

*Yes. Once the TC decides to commit—evidenced here by *A* having received the commit message—the transaction will eventually be committed everywhere.*

- ii.** What if, instead of hearing from *A*, *B* contacted *C* and learned that *C* knew nothing about the transaction. Could *B* go ahead and abort? Do either need to remember anything about the transaction afterwards?

*Since *C* knows nothing, it either already sent an abort message and forgot about it, or has not yet received a prepare message. In either case, it is safe for *B* to abort. *C*, however, must remember and promise to abort the transaction should it ever receive a prepare message regarding this transaction.*

- iii.** Finally, suppose *B* hears from both *A* and *C* that they all sent a “prepare-ok,” but none of them have received a commit message. Can they decide to commit themselves? How about abort? Why?

*They can neither commit nor abort. The TC may have decided to do either, and the messages are just delayed. If *B* decides to commit/abort, but, just then, *A* hears from the TC to do the opposite, we have a problem.*

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 aims to significantly decrease the management overhead of large-scale deployments as compared to Harp, but still relies on some of the same techniques under the covers.

- A) (5 points)** Why do Frangipani servers keep their logs in Petal? In particular, because each log is private, wouldn't it be better to keep it on a local disk? Similarly, assuming all nodes are operating properly, why does each server need a lock for its private log?

Logs are kept in Petal so that another server can run recovery should a server crash. Locks are required to ensure that only one server is playing a given log—either because two servers may attempt recovery simultaneously, or if a server is declared dead but is simply slow.

- B) (10 points)** What semantics does Frangipani guarantee across server failures? Suppose, for example, that a client creates a file, writes some data, and then closes the file. What could happen if the server the client is using crashes somewhere during this process? After recovery, what can you say about the state of the filesystem?

Frangipani uses write-ahead logs, so any actions that the client received responses to were written to a server log, so they will be reflected on disk after recovery. The client may timeout and retry its operations at the same server or elsewhere, but no server will process its request to write the file until it obtains the lock, which can only happen after the writes at the original server have reached Petal—either by the original server itself before it crashed, or by replaying its log during recovery. It is important to note, however, that Frangipani only logs meta-data, so there are no guarantees about the state of the file data, just as with the traditional Unix filesystems.

- C) (5 points)** Suppose another client was aware of the file's creation, and was attempting to modify the file through another Frangipani server at the time the server above crashed. What will happen with these modifications? When (if ever) will they be allowed to complete?

There are two cases—either the second server was able to acquire the write lock from the first server before the latter crashed, or it wasn't. In the first case, the first server must have flushed all pending writes to Petal, so the second server can process the second client's write requests immediately. In the second case, it must wait until the first server's log is recovered and the lock released. In either case, there is no guarantee what (if any) data written by the first client will have made it to disk, but the second client's writes will be ordered strictly after them.

- D) (5 points)** Suppose the second server also crashed while attempting the modification. How does Frangipani know what order to replay the logs in? Does it need to replay one before the other, or interleave them? Does it matter?

The order in which the logs are replayed is undefined—they may even be replayed simultaneously—but it doesn't matter. At most one server could have crashed holding the write lock for any given file, so that is the only server's log that contains uncommitted updates for the file in question. Hence, no matter which order the logs are replayed, the correct order of writes will be maintained.

Some of you noted that the Frangipani logs also keep version numbers, so old writes will not be replayed. That is not a concern here, though, as only one server will hold the lock to be able to commit its logged writes. The version numbers are required to prevent a single server from replaying previously committed writes of its own if it subsequently reacquired the same lock.