

CS 138 Midterm Exam Solutions

Spring 2008

1. *In homework 1 you showed how to add a node to a chord system, which involved updating every node's finger table. Is it necessary to update all tables instantly? I.e., can the system be used before all finger tables are updated? Explain. (Hint: you're not being asked to say anything about how finger tables are constructed. However, you do need to know the purpose of finger tables.)*

An important property of the Chord system is that finger tables are merely an optimization—we can search for keys without them (assuming correct successor pointers). If we add nodes to the system, but don't instantly update their finger tables, we can rely on the successor pointers to find keys.

2. *It is said to be easier to migrate a virtual machine from one real machine to another than it is to migrate a Unix process. Explain why. (Hint: consider the state information required to represent both. Where is this state? What other system components can modify it?)*

Virtual machines are self-contained and isolated; they have no shared state with other system components and thus can be easily moved from one (real) machine to another. Processes, on the other hand, have many components of their state that are shared with other parts of the system. For example, they have open files that are potentially shared with other processes. They may share memory and synchronization objects with other processes. In addition there are issues such as delivery of signals that don't come up with virtual machines. Thus a process can't simply be moved from one machine to another; one has to account for the sharing and signal delivery that must continue to take place.

3. *Assume that the server used in parts a and b below never crashes.*
- a. *TCP provides reliable transmission of data. Explain why DCE RPC, when layered on top of TCP, does not provide exactly once semantics. Be sure to give an example.*

The problem is that, despite the server's never crashing, network connections may fail. Thus, for example, shortly after a client sends an RPC request to the server, and before the client receives the server's response, the connection might be lost. The client can then create a new connection, but it will be uncertain as to whether its request had ever reached the server. It can repeatedly send the request until it gets a response, resulting in at-least-once semantics, or it can give up after the first request, resulting in at-most-once semantics.

- b. *Explain how you might modify DCE RPC so that it does provide exactly once semantics when layered on top of TCP.*

The client might provide sequence numbers with its requests (in addition to the sequence numbers used by TCP). The server could keep track of the most recent request and sequence number it's received and cache its response. Thus the client could safely retransmit; the server could detect retransmissions and repeat its original response.

4. *Below is the algorithm given in class for sending totally ordered multicasts using logical clocks:*

- *To send multicast:*
 - *tag message with sender's timestamp (<time, sender ID>)*
 - *sender receives own multicast*

- *On receipt of message*
 - *queue message in timestamp order*
 - *multicast an acknowledgement*
- *On receipt of acknowledgement*
 - *link to acknowledged message*
- *Deliver message to application when*
 - *message is at front of receive queue*
 - *has been acknowledged by all*

Are the multicasts causally ordered as well? Explain.

Yes. Recall that the use of logical clocks produces both a total and causal order of events. If event *a* “happens before” event *b*, then the logical-clock time of *a* is less than that of *b*. This is because, on receipt of a message, a process’s logical clock is adjusted to be greater than the time stamp on the message, which is the logical-clock time of the sender when the message was sent. Thus the causality induced by messages is reflected in logical-clock values.

So, since the multicasts are ordered by their logical-clock values, they are causally ordered.

5. *Below is the description of the two-phase-commit protocol given in class:*

- *Phase 1*
 - *coordinator prepares to commit:*
 - *asks participants to vote either “commit” or “abort”*
 - *participants respond appropriately*
- *Phase 2*
 - *coordinator decides outcome:*
 - *if all participants vote commit, outcome is commit, otherwise outcome is abort*
 - *outcome sent to all participants*
 - *participants do what they’re told*

Recall that blocking can occur if the coordinator fails. Explain why it can’t be eliminated even if a new coordinator is elected. (“Blocking” means that participants must wait until the coordinator comes back up, since otherwise they can’t come to a consensus that includes participants that might have voted, then failed.)

It could be the case that one of the participants had failed after voting, and all the remaining participants had voted to commit. Even if a new coordinator is elected, it won’t be able to determine how the failed participant had voted and whether the original coordinator had started to send the outcome to all participants. The failed participant might have voted to abort, in which case all participants must abort. On the other hand, the failed participant might have voted to commit, and the original coordinator, after getting commit votes from all, had notified the failed participant of the outcome before both had crashed (and thus the final outcome must be commit). The only way to find out what happened is to wait for the original coordinator to restart.