CSE 486/586 Distributed Systems
Midterm Solutions
Monday, 3/5/12

DIRECTIONS

- Time limit: 45 minutes (3:05pm - 3:50pm)
- There are 50 points and 5 bonus points (whatever you score more than 50 will be your bonus points).
- This is a closed-book, no calculator, closed-notes exam.
- The only exception is your cheat sheet (one-sided, letter-sized).
- You should turn in your cheat sheet as well as your answer sheets at the end of the exam.
- Each problem starts on a new page.
- Please use a pen, not a pencil. If you use a pencil, it won't be considered for regrading.
- Each problem also explains its grading criteria. "Atomic" means that you get either all the points or no point, i.e., there are no partial points.

Name:	
UBITName:	

Problem #	Score
1	
2	
3	
4	
5	
6	

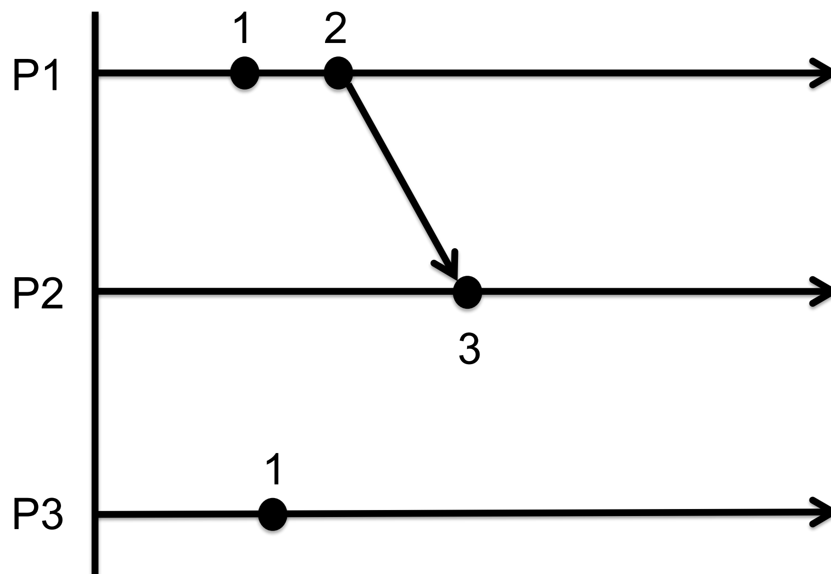
1. (a) Explain what fate-sharing is and how it was applied to the design of the Internet.
(**Grading**: atomic 3 points)
Answer:
The principle of fate-sharing is that it is acceptable to lose the state information associated with an entiry if, at the same time, the entity itself is lost. Following this principle in the design of the Internet resulted in a best-effort network.

(b) Explain what the end-to-end argument is. Describe this in one sentence and don't forget to include what the exception is.
(**Grading**: atomic 3 points)
Answer:
The principle of the end-to-end argument is, if a functionality must be implemented end-to-end, then do not implement it in the network, except when there are clear performance improvements.

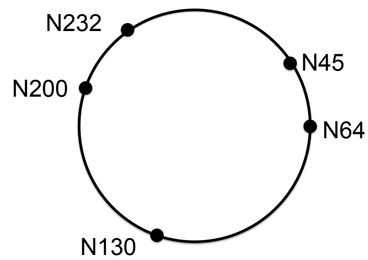
(c) Under what conditions can we design a reliable, totally-ordered multicast algorithm?
(**Grading**: atomic 3 points)
Answer:
Designing a reliable, totally-ordered multicast is essentially a consensus problem. Due to the impossbility of consensus result, it is impossible to design this in an asynchronous system. If we relax the assumptions of asynchronous systems, i.e., if there's a known delay bound or if we can say that there is no failure, then it becomes possible to design such an algorithm.

2. In Lamport clocks, it is ***not*** guaranteed that if $L(e) < L(e')$ then $e \rightarrow e'$, where $\rightarrow$ indicates the "happened-before" relation between two events and $L(e)$ denotes the timestamp of event e at whatever process it occurred at. Give an example that demonstrates, even if $L(e) < L(e')$, it is not true that $e \rightarrow e'$, using 3 processes below. Clearly mark which 2 events demonstrate it.
(**Grading:** atomic 4 points)

Answer:
Multiple solutions are possible. One example is given below, where P2's event and P3's event demonstrate it.



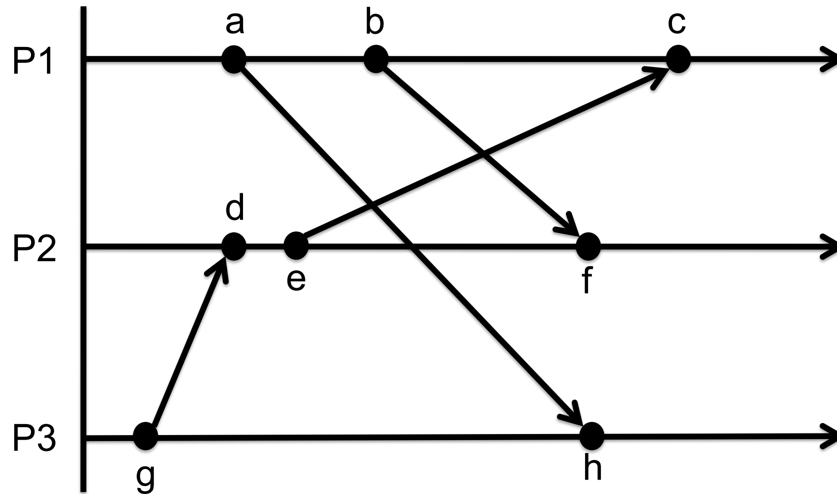
3. Below is a Chord ring with the id space of 2^8. There are 5 nodes in the system. What is the finger table at node 200 (N200)? You can use the table provided below.
(**Grading:** 5 points)



Answer:

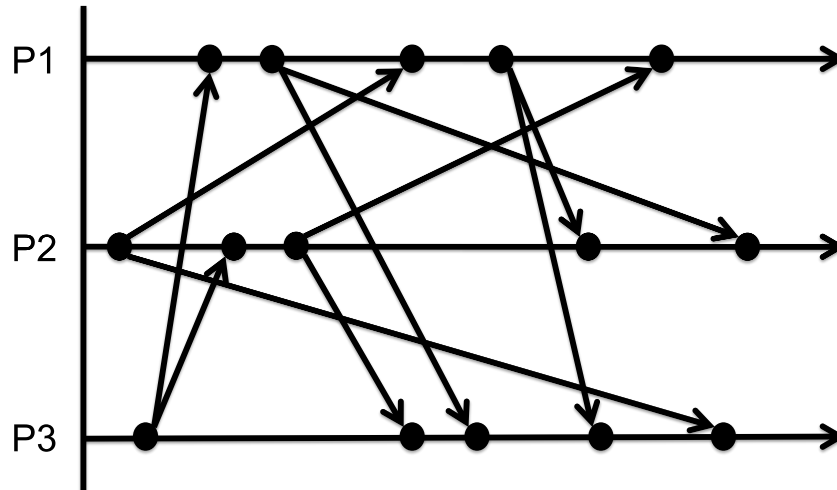
i	Finger
0	232
1	232
2	232
3	232
4	232
5	232
6	45
7	130

4. Using the vector logical clock discussed in class, list *all possible pairs of concurrent events* that appear in the timeline below. You can use || to denote a pair of concurrent events, e.g., *a* || *b* means *a* and *b* are concurrent.
(**Grading:** 12 points)



Answer:
a || *d*
a || *e*
a || *g*
b || *d*
b || *e*
b || *g*
b || *h*
c || *f*
c || *h*
d || *h*
e || *h*
f || *h*

5. Using the multicast algorithm that provides causal ordering discussed in class, mark the timestamps at the point of each multicast send and each multicast receipt. Also mark multicast receipts that are buffered, along with the points at which they are delivered to the application.
(**Grading:** 10 points)



Answer:
P1:
Accept, (0,0,1)
Send, (1,0,1)
Accept, (1,1,1)
Send, (2,1,1)
Accept, (2,2,1)

P2:
Send, (0,1,0)
Accept, (0,1,1)
Send, (0,2,1)
Buffered as (2,1,1), Timestamp still (0,2,1)
Accept (1,2,1)
Accept (2,2,1) (previously buffered)

P3:
Send, (0,0,1)
Buffered as (0,2,1), Timestamp still (0,0,1)
Accept (1,0,1)
Buffered as (2,1,1), Timestamp still (1,0,1)
Accept (1,1,1)
Accept (1,2,1) (previously buffered)
Accept (2,2,1) (previously buffered)

6. Tired of trying to understand distributed mutex algorithms developed by others, your instructor of CSE 490/590 has decided to write his own algorithm. His algorithm assumes that 1) network channels are reliable and preserve FIFO order; 2) there is no process failure. The following is the algorithm description, assuming that it runs at process p_i. The algorithm uses a local lock which is just meant to be used inside each process; it has nothing to do with distributed mutual exclusion.

#1: On initialization

```
state := RELEASED;
local_lock := UNLOCKED; // Per-process lock
cnt := 0;
For each process p_j (j ≠ i) in the group,
        rcv_j := 0;
```

#2: To enter the critical section

```
lock(local_lock);
state := WANTED;
cnt++;
For each process p_j (j ≠ i) in the group,
        Send a request to p_j with rcv_j attached to it;
unlock(local_lock);
Wait until (number of replies received = (N − 1));
state := HELD;
```

#3: On receipt of a request from p_j at p_i $(i \neq j)$

```
lock(local_lock);
tmp_rcv := rcv_i value attached in the request from p_j
if (state = HELD)
        Queue the request from p_j without replying;
else if (state = WANTED and tmp_rcv < cnt)
        Queue the request from p_j without replying;
else
        Reply immediately to p_j;
        rcv_j++;
unlock(local_lock);
```

#4: To exit the critical section

```
lock(local_lock);
state := RELEASED;
For each queued request,
        // Say the request is from p_j
        Send a reply to p_j;
        rcv_j++;
unlock(local_lock);
```

(Continue on the next page)

(a) Does this algorithm guarantee liveness? If the answer is yes, prove it. If the answer is no, give a clear example that does not satisfy liveness.
(**Grading:** 5 points)

Answer:

The answer is no. One example is two processes trying to enter the critical section at the same time, both for the first time, i.e., all counters are 0 at both processes. According to the protocol, they will each increment its *cnt* by one, set their state to WANTED, send a request to each other. Then they will both execute "else if" in #3, which will buffer the other process's request and never reply. This gives a deadlock.

(b) Does this algorithm guarantee safety? If the answer is yes, prove it. If the answer is no, give a clear example that does not satisfy safety.
(**Grading:** 10 points)

Answer:

The answer is yes. The intuition is that if a process is executing "else if" in #3, the system will always deadlock, meaning that there is no process that can enter the critical section. In other cases (HELD or RELEASED), only one process can enter the critical section. This guarantees safety as at most one process can enter the critical section. More formally, we can prove it by contradiction.

Suppose that there are two processes, p_0 and p_1 in the critical section (violating the safety guarantee). Let's say p_0 entered the critical section as a result of sending req_0, and p_1 entered the critical section as a result of sending req_1.

Now let's backtrack and consider the time p_0 received req_1 from p_1. p_0 could have been in one of three states (HELD, RELEASED, and WANTED) for req_0. Thus,

case 1: p_0 is in HELD. In this case, p_1 cannot go into the critical section until p_0 exits. This contradicts our assumption.

case 2: p_0 is in RELEASED. In this case, p_0 is not in the critical section, which contradicts our assumption.

case 3: p_0 is in WANTED. If $(rcv_j < cnt)$ is true, then p_0 will not reply and p_1 cannot go into the critical section, which contradicts our assumption. If $(rcv_j < cnt)$ is not true, the only case is $(rcv_j == cnt)$, which means that p_1 should have received p_0's req_0 and have replied to p_0 already (because network channels are reliable and FIFO). This means that p_0 should have been in HELD, not WANTED. This contradicts our assumption.