Solutions

Name:_____ Duration: 90 minutes

Mid-Term Exam 1 CS 425/ECE 428 Distributed Systems (Spring 2018) Total Points: 50

**1.** (9 points)

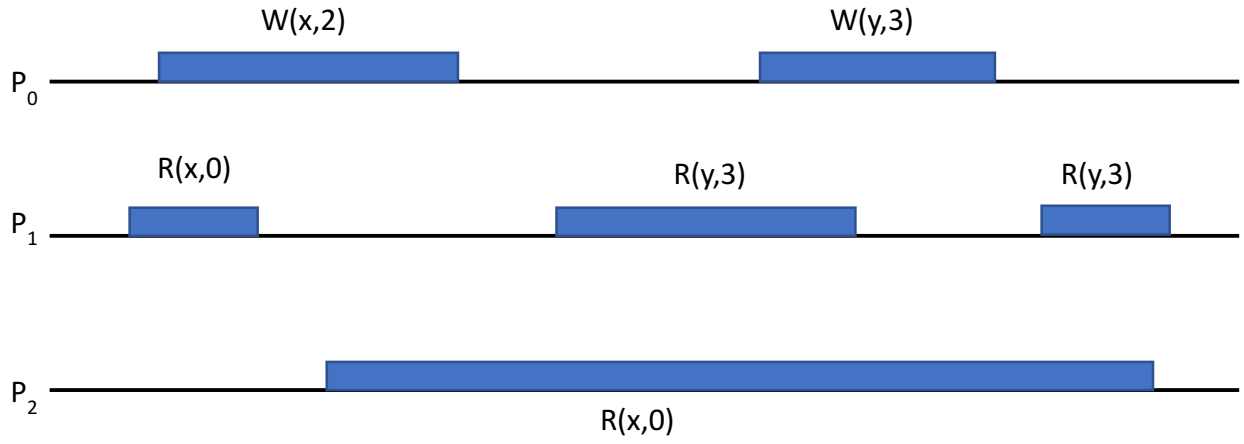
*** In each execution in this question, all variables are initialized to 0.**

*** In the figures in this question, R(x,v) denotes a Read operation of variable x that returns value v. For example, R(y,3) denotes R(y) operation that returns value 3.**

*** Each blue bar denotes the period between invocation and acknowledgment of the specified operation.**

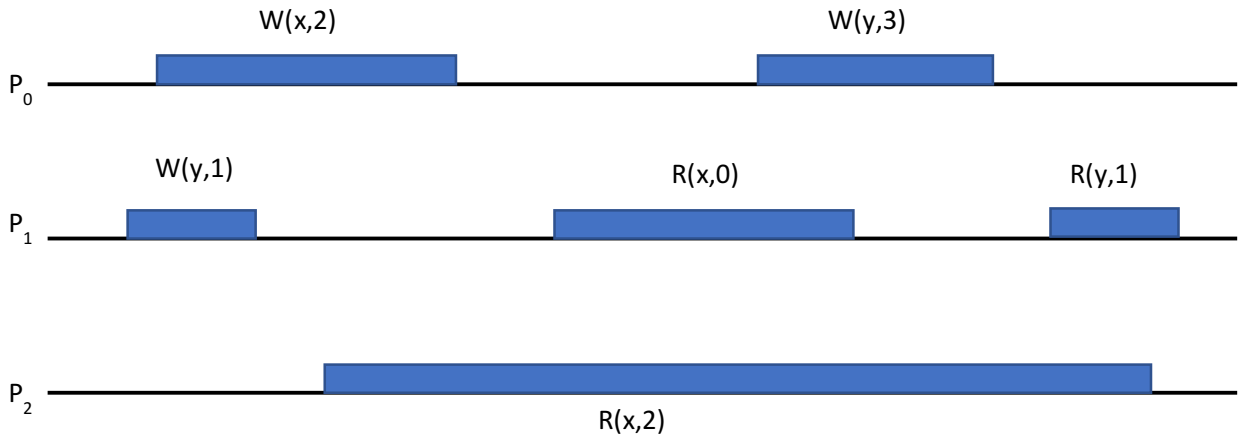
(a) Is the execution below linearizable? YES

If you answer NO, then delete a minimum number of operations to ensure that the modified execution is linearizable. Circle the operation(s) that you want to delete.



(b) Is the execution below sequentially consistent? YES

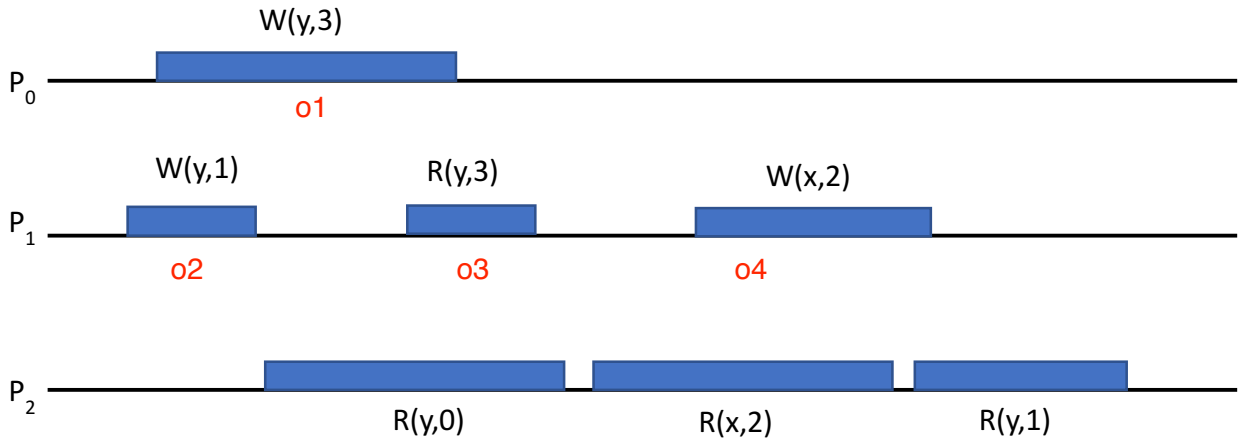
If you answer NO, then delete a minimum number of operations to ensure that the modified execution is sequentially consistent. Circle the operation(s) that you want to delete.




Text

(c) Is the execution below causally consistent? Provide a **justification** for your answer.

YES




o2->o3->o4
o5->o6->o7
o1->o3 (read-from)
o2->o7 o4->o6 (read-from)

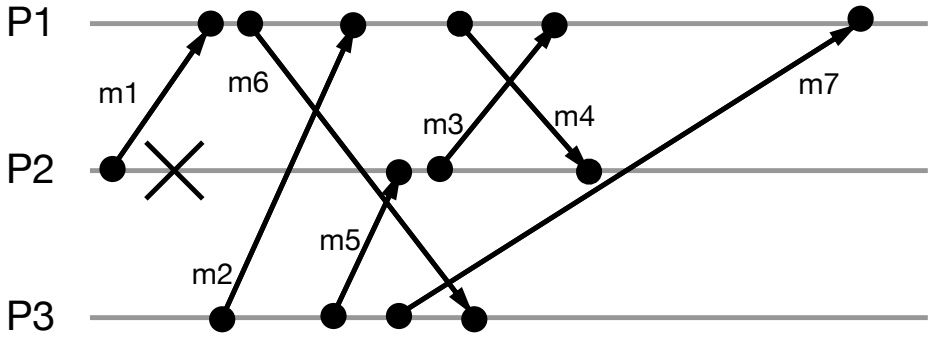
Permutation for p0: o1, o2, o4
permutation for p1: o2 o1 o3 o4
permutation for p2: o5 o1 o2 o4 o6 o7

2. (7 points) The two parts of this question are independent of each other. Let $C_{ij}$ denote the channel state recorded for the channel from process Pi to process Pj.

   (a) Suppose that process P2 initiates the Chandy-Lamport algorithm at the time shown by X in the figure. In this case, is it possible for the Chandy-Lamport algorithm to record the channel states below? **Explain your answer.**

$C_{12}$ = {}  $C_{13}$ = {m6}
$C_{21}$ = {}  $C_{23}$ = {}
$C_{31}$ = {m2,m7}  $C_{32}$ = {}

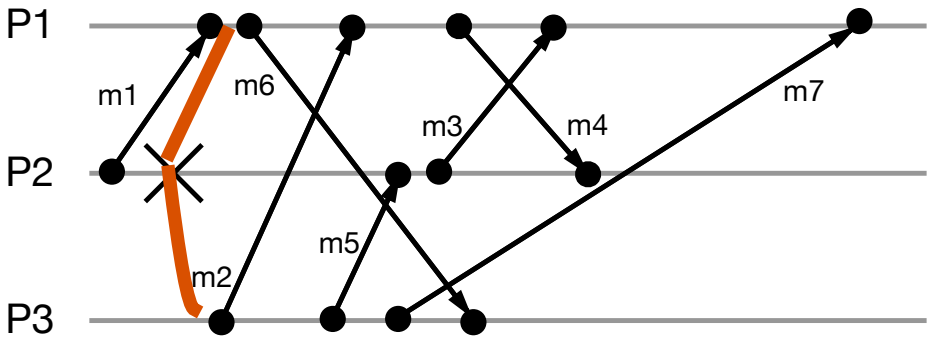
NO.
Chandy-Lamport records consistent snapshot. Since m2, m7 are channel state, P3 must have recorded its local state after sending m7, and therefore, after sending m5. But m5 is not received before P2 records its state. Thus, m5 must be part of channel state.



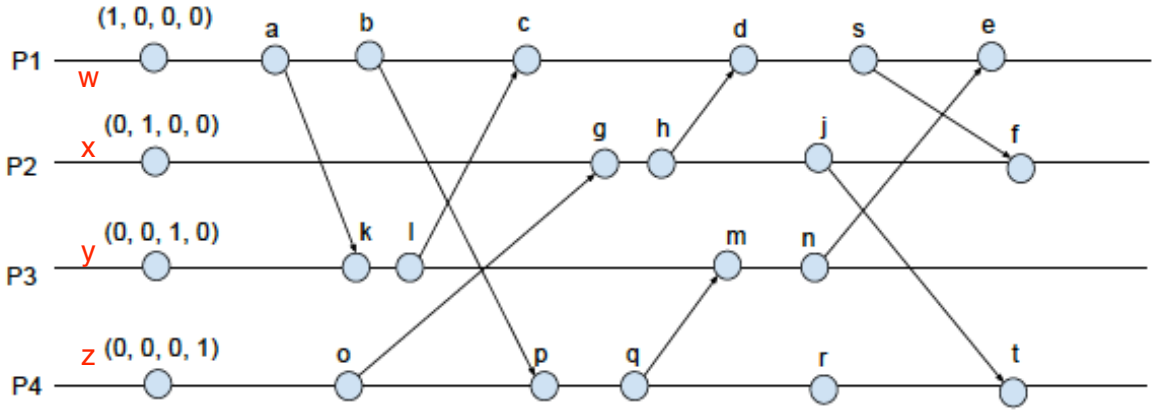

   (b) Suppose that process P2 initiates the Chandy-Lamport algorithm at the time shown by X in the figure. In this case, is it possible for all the channels to be recorded as empty?

If you answer YES, then show the markers sent by process P2 (i.e., when the markers are sent by P2 and received by the other processes).

YES.



3. (4 points) The figure below shows the vector timestamp of the first event at each process.




(a) List all the events that happened-before (→) event d.

w,a,b,c
x,g,h
y,k,l
z,o

(b) Determine the vector timestamp of events g and d.

g: (0,2,0,2)

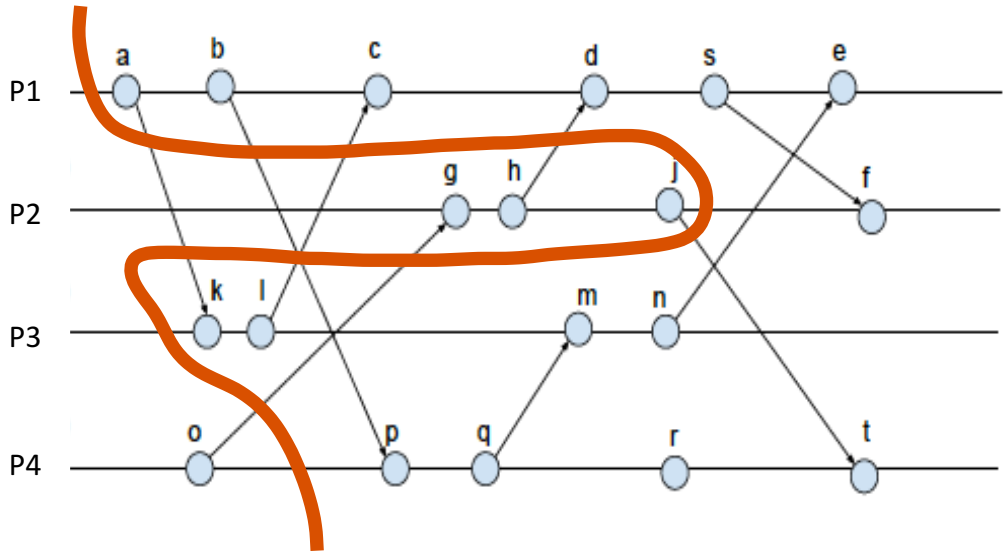
d: (5,3,3,2)

(c) List all the events that are concurrent with event d.

j,m,n
p,q,r,t

4. (4 points) In the execution below, show the consistent cut that includes event **j** and the **least possible** number of other events. **List the events** that are in your chosen consistent cut.

Events in the cut:
g,h,j,o



5. (7 points) Consider the asynchronous approximate consensus algorithm provided in the handout for this exam, **but with (*n-f*) in steps 3 and 4 changed to (*n-2f*)**. There are at most *f* crash failures.

(a) Suppose that $n = 2f+1$. Will the **modified** algorithm achieve approximate consensus for any specified value of ε in presence of up to f crash failure? Explain your answer.

No.
Each process waits for only 1 message.
Thus, each process may potentially use only its own message in step 4. This will imply that the state of each process remains equal to its input.

(b) Suppose that $n = 4f+1$. Will the **modified** algorithm achieve approximate consensus for any specified value of ε in presence of up to f crash failures? Explain your answer.

Yes.
Define F = 2f.
Then n = 2F+1.
Observe that the modified algorithm can tolerate F = 2f faults.
Thus, the algorithm will perform correctly.

6. (7 points) Consider servers A and B that want to synchronize their clocks with each other. In this system, the message delay from A to B is constant, and this delay is known be exactly twice the delay from B to A.

Is it possible to design a clock synchronization algorithm for servers A and B that achieves 0 skew? If you answer NO, explain why. If you answer YES, provide a suitable algorithm, and explain why the skew achieved is 0.

You may assume that the processing delays at servers A and B are 0.

YES.

Algorithm steps:

1. Server A sends message to server B.

2. Server B responds to server A's message with local clock value T.

3. Server A measures RTT (round-trip time) for the above exchange, i.e., the duration from when A sent message in step 1 until the time when A received message in step 2.

4. Server A sets its clock equal to T+(RTT/3), on receipt of the response from B.

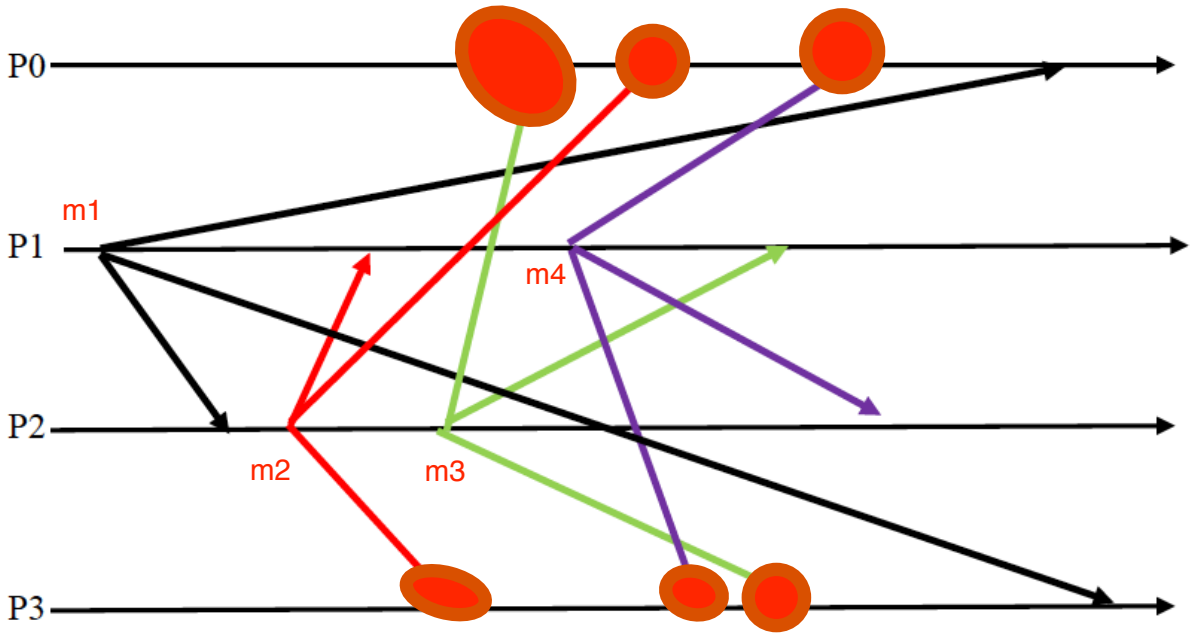
Observe that the delay of the message from server B to A is exactly RTT/3 due to the specified condition on delays.
Thus, at the time when A gets B's response, B's clock will be T+RTT/3.
Step 4 ensures 0 skew.

Analysis assumes no drift.

7. (7 points) In the execution below, processes send messages to each other to implement **causally-ordered multicast.** To simplify the picture, messages sent by each process to itself are not shown, but you may assume that such messages are received instantaneously.

(i) Identify the messages that are buffered at the processes to ensure causally-ordered multicast delivery. (Circle the receive event for the buffered messages to identify those messages.)

(ii) For each message buffered as above, determine the earliest instant of time at which the message may be delivered, while ensuring causally-ordered multicast. (To identify the instant of time, draw an arrow that begins at the time when the message is received to the time at which the message may be delivered.)




send(m1) —> send(m2) send(m2)—>send(m3) send(m2)—>send(m4)

Allowed order
of delivery:

m1, m2, m3, m4
or
m1, m2, m4, m3

8. (5 points) State true or false:

False (a) To implement eventual consistency, it must be possible for the replicas to synchronize their clocks without any skew.

False (b) Consider an implementation of eventual consistency that supports "read-my-write" guarantee. Suppose that a process P writes value 3 to variable V, and then reads variable V. This read of V must return value 3.

False (c) In the Paxos protocol, if a certain acceptor has responded to an *accept* request with proposal number *n*, then that acceptor will never respond to an *accept* request with proposal number greater than *n*.

False (d) In the Paxos protocol, for any given proposal number *n*, each acceptor responds to at most one *prepare* request with proposal number less than *n*.

False (e) If each message is guaranteed to incur at most 10 ms delay, then the Paxos protocol guarantees that a majority of acceptors will accept a proposal within 100 ms after some proposer sends a *prepare* request to the acceptors.

## Logical Clock (Lamport Clock)

**Li intialized to 0 at process Pi**

- **Compute event at process Pi:**
  - **Increment Li by 1**
  - **New value of Li is the timestamp of the compute event**
- **Send event at process Pi: Consider e = send(m)**
  - **Increment Li by 1**
  - **New value of Li is the timestamp of send event e**
  - **Piggyback the timestamp of e with message m**
- **Receive event at process Pi: Suppose (m,t) where m is a message, and t is the piggybacked timestamp, is received at event e at Pi**
  - **Update Li as Li := max(Li, t)+1**

## Vector Logical Clocks

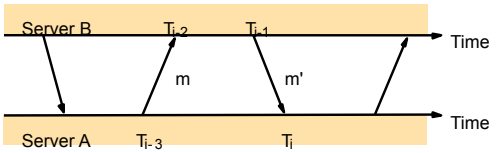
- **With Lamport Logical Timestamp**
  - **e → f ⇒ timestamp(e) < timestamp (f), but**
  - **timestamp(e) < timestamp (f) ⇒ {e → f} OR {e and f concurrent}**
- **Vector Logical time addresses this issue:**
  - **Each process maintains a vector clock, length = number of processes**
  - **At each event, process i increments $i^{th}$ element of vector $V_i$**
    - **→ The new $V_i$ is the timestamp of the event**
  - **A message carries the Send event's vector timestamp**
  - **For a receive(message) event at process k … let $V_{message}$ denote vector timestamp received with the message**

$$V_k[j] = \begin{cases} \text{Max}(V_k[j], V_{message}[j]), & \text{if } j \text{ is not } k \\ V_k[j] + 1 & j = k \end{cases}$$

## Comparing Vector Timestamps

- **$VT_1 = VT_2$,**
  **iff $VT_1[i] = VT_2[i]$, for all i = 1, … , n**
- **$VT_1 \leqslant VT_2$,**
  **iff $VT_1[i] \leqslant VT_2[i]$, for all i = 1, … , n**
- **$VT_1 < VT_2$,**
  **iff $VT_1 \leqslant VT_2$ &**
  **$\exists$ j (1 ⩽ j ⩽ n & $VT_1[j] < VT_2[j]$)**
- **$VT_1$ is concurrent with $VT_2$**
  **iff (not $VT_1 < VT_2$ AND not $VT_2 < VT_1$)**

## Theoretical Base for NTP




- $t$ and $t'$: actual transmission times for $m$ and $m'$ *(unknown)*
- $o$: true offset of clock at $B$ relative to clock at $A$
- $o_i$: estimate of actual offset between the two clocks
- $d_i$: estimate of accuracy of $o_i$; total transmission times for $m$ and $m'$; $d_i = t + t'$

$$T_{i-2} = T_{i-3} + t + o$$
$$T_i = T_{i-1} + t' - o$$

This leads to

$$d_i = t + t' = T_{i-2} - T_{i-3} + T_i - T_{i-1}$$
$$o = o_i + (t' - t)/2, \quad \text{where}$$
$$o_i = (T_{i-2} - T_{i-3} + T_{i-1} - T_i)/2.$$

It can then be shown that

$$o_i - d_i/2 \leq o \leq o_i + d_i/2.$$

## Causal Ordering using vector timestamps

Algorithm for group member $p_i$ $(i = 1, 2 \ldots, N)$

*On initialization*
  $V_i^g[j] := 0$ $(j = 1, 2 \ldots, N)$;

The number of group-g messages from process j that have been seen at process i so far

*To CO-multicast message m to group g*
  $V_i^g[i] := V_i^g[i] + 1$;
  *B-multicast*$(g, <V_i^g, m>)$;

*On B-deliver*$(<V_j^g, m>)$ *from* $p_j$, *with* $g = group(m)$
  place $<V_j^g, m>$ in hold-back queue;
  wait until $V_j^g[j] = V_i^g[j] + 1$ and $V_j^g[k] \leq V_i^g[k]$ $(k \neq j)$;
  *CO-deliver m*;  // after removing it from the hold-back queue
  $V_i^g[j] := V_i^g[j] + 1$;

## Linearizability

**An execution is linearizable if there exists a permutation that is**

***valid*,**

***per-process order-preserving*, and**

***real-time order-preserving***

EXAM 1 HANDOUT

# Approximate Consensus in Presence of Crash Failures

Consider an asynchronous system consisting of $n$ processes. named $1, 2, 3, \cdots, n$, with $x_i$ denoting the input of process $i$.

All processes can communicate with each other.

At most $f$ processes may crash.

The following algorithm performed by each fault-free process $i$ achieves approximate consensus provided thet $n \geq 2f + 1$. Specifically, after $r$ becomes sufficiently large at all the fault-free processes, for fault-free processes $i, j$, we will have $y_i$ apporoximately equal to $y_j$ (more specifically, for any positive constant $\epsilon$, after $r$ exceeds a certain threshold at all the nodes, $|y_i - y_j| < \epsilon$).

1. Initialization:
   $y_i := x_i$
   $r := 1$
2. Send message $(y_i, r)$ to all the processes including self.
3. Wait until $(n - f)$ messages of the form $(*, r)$ are received (including message from self).
4. Update $y_i$ = average of the $n - f$ values in the above $n - f$ messages. Note that the value is the first field in the tuple in each message.
5. $r := r + 1$
6. Go to step 2

The algorithm above is said to perform "asynchronous rounds", with $r$ being the round index. Unlike a synchronous round, an asynchronous round may complete at different processes at very different times. However, due to the inclusion of round index in the messages, each process can use the appropriate messages in its computation in each round.

The above algorithm may be terminated after a large enough number of rounds (termination not shown in the code above) by when the $y_i$ values for different nodes are guaranteed to be within a desired constant.