CLASS NOTES for Oct 13, 2017 and Oct 14th, 2017
Distributed Systems
Fall 2017
Dr. Garg

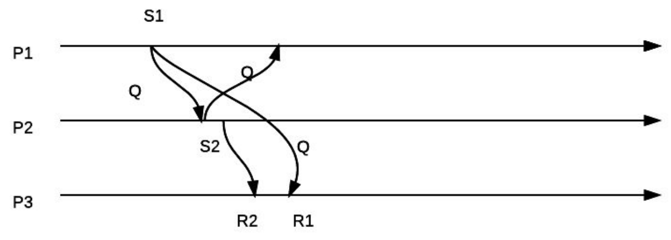
October 13, 2017

<u>Causal Ordering of messages</u>

If $s_1 \rightarrow s_2 \Rightarrow \neg\, r_2 < r_2$
Where (s_1, r_1) is the first message and (s_2, r_2) is the second message

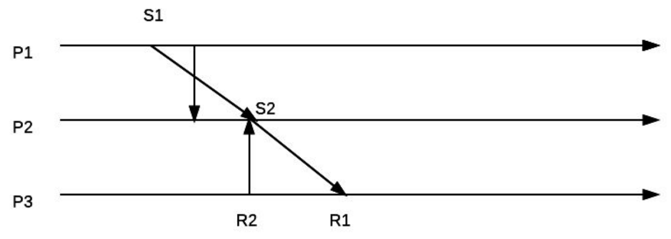
If 2 messages are sent to Pi such that the send of the first message happened before the send of the second message, then Pi must be delivered in the same order.

<u>Example</u>:



Here, P3 gets an answer with the context of a question.

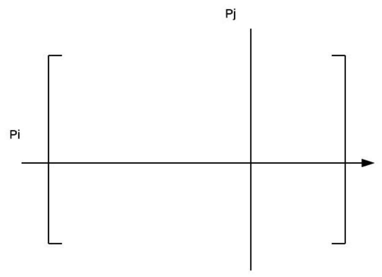
Another way to look at this:
A message chain should not overtake a direct message.



In this case, P3 should wait, then deliver R1

Causal ordering is stronger than
FIFO

We had a class discussion on how to solve this problem.
Dr. Garg's advise: Solve for correctness, then make it efficient.
Suggestion - that each process tells about all the messages send that it is aware of.

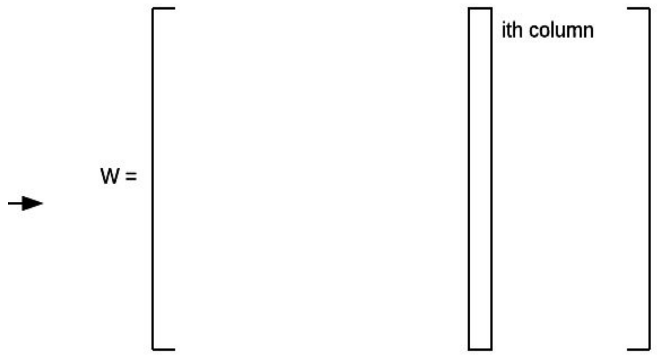


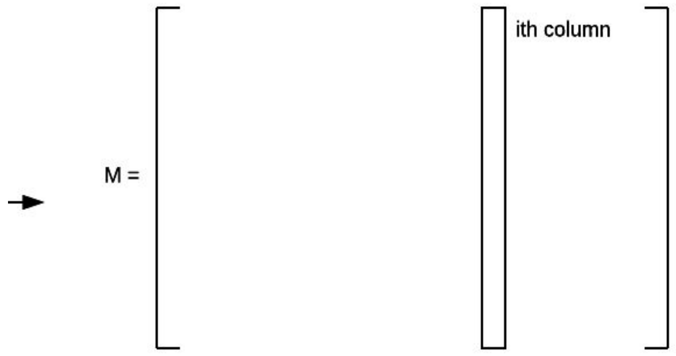
How many messages that Pi has sent to Pj that I know of

M[i, j] @ P_x
≡ # of messages send from P_i to P_j as a known to P_k.

(Uses that matrix clock)

Matrix that comes with the message:





How do you figure out if you need to deliver a message or not?
Compare the two matrices, and if any W (i^{th} column) <= M(i^{th} column), then, you don't deliver the message.

Update the entire matrix.

October 14, 2017

$S_1 \rightarrow S_2 \Rightarrow \neg (r_2 < r_1)$

<u>Code from perspective of process P_i:</u>

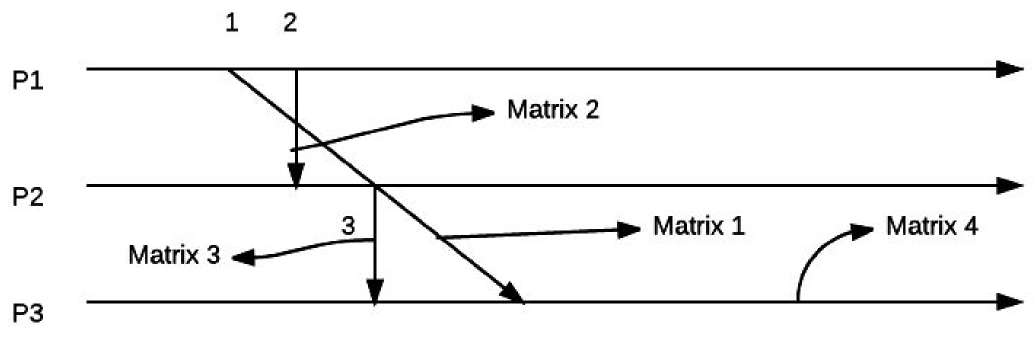
```
Var M:( array [1..N, 1..N] of int init all O;

To send message to $P_j$
        M[i,j] ++;
        Piggyback M with the message

Upon receiving a message from Pj tagged with matrix W
// Delivery enabled if W[j] ≤ M[j]

$\forall$ k: k ≠ j W [k, i ] ≤ M [k, i]
W [j, i] = 1 + M [j, i]

On delivery:
M:= max(M, W);        // componentwise
```

Order n^2 messages to be piggy-backed with every message.

Chapter 12

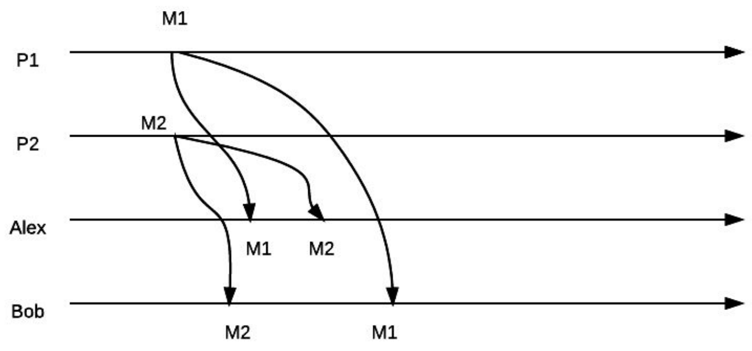
Example:

This is monotonically increasing.



Matrix 1 $\begin{bmatrix} 0 & 0 & 1 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \end{bmatrix}$ Matrix 2 $\begin{bmatrix} 0 & 1 & 1 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \end{bmatrix}$ Matrix 3 $\begin{bmatrix} 0 & 1 & 1 \\ 0 & 0 & 1 \\ 0 & 0 & 0 \end{bmatrix}$ Matrix 4 $\begin{bmatrix} 0 & 0 & 1 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \end{bmatrix}$

Total Ordering of messages
Example:



Everyone should get a consistent view of the world. This violates total order.

Total Order of Messages

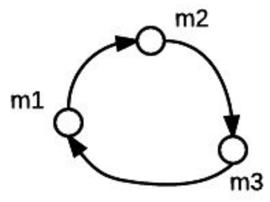
A computation satisfies total order if there exists a total order on all messages such that, if every process delivers all messages in an order that is consistent with that total order.

For m_1, m_2, two possible orders:

1. m_1, m_2
2. m_2, m_1

Everyone should deliver in a particular order.
Let order be m_1 m_2 m_3
Alice: m_1 m_2
Bob: m_2 m_3
Charlie: m_3 m_1

Charlie is ! total order of cycle



Let's try and come up with a centralized algorithm.

Centralized Algorithm:

- Send your message to the server
- Server now multicast all the messages

Assume:

1. FIFO (TCP guarantees FIFO)
2. Can have only one pending request

Distributed Algorithm:

We want to get total order to all messages. We have seen this in Mutex, ordered by Lamport's Algorithm (note that here, the symbol ≡ means "map that", CS is Critical Section)

- CS request ≡ request to multicast a message
- Request timestamp ≡ final timestamp of the message
- P_i entering CS ≡ message m=by P_i is delivered

Lamport's logical clock is running
Everyone has a queue/buffer
Message removed when it enters CS

As in Lamport's algorithm, when it enters CS in the order sent, there is no deadlock.

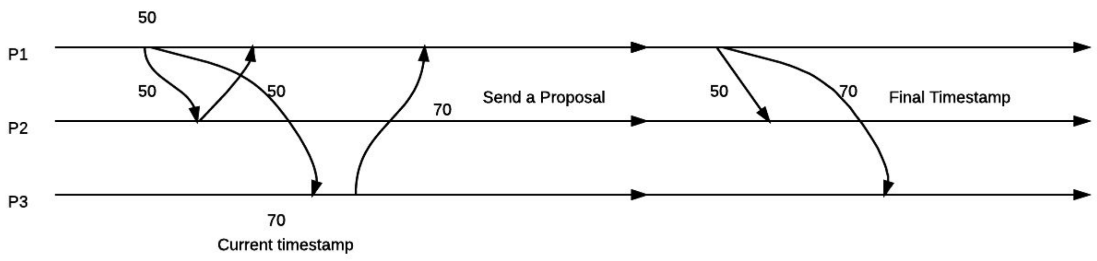
- Send out timestamp message to all including yourself
- On receiving a request message, send an ack
- P_i can enter CS if P_i sends release message to all

(We will deliver message when it is released)

Lamport's algorithm needs delivery to all, and our case can deliver to some, so we have to send to everyone with a "not meant for you" message.
Order would be 3(N-1) messages per multicast, so we need to improve on this.

<u>Skeen's Algorithm:</u>
Goal: Send out messages to only some people, and keep total order
- Use Lamport's clock
- The final timestamp is determined by the sender + recipient

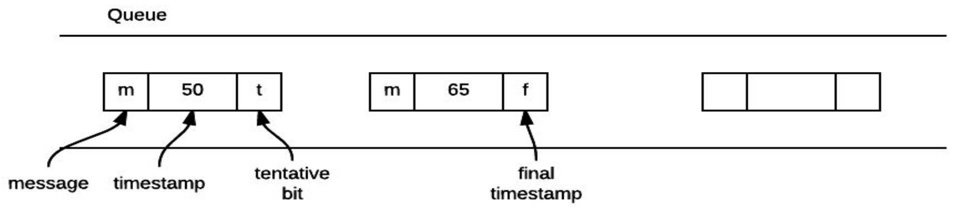
Note that is the previous algorithm, the final timestamp was only determined by the sender.



Takes max of all received proposals

<u>Property to maintain:</u>
Every process delivers messages in the order of the final timestamp of message.

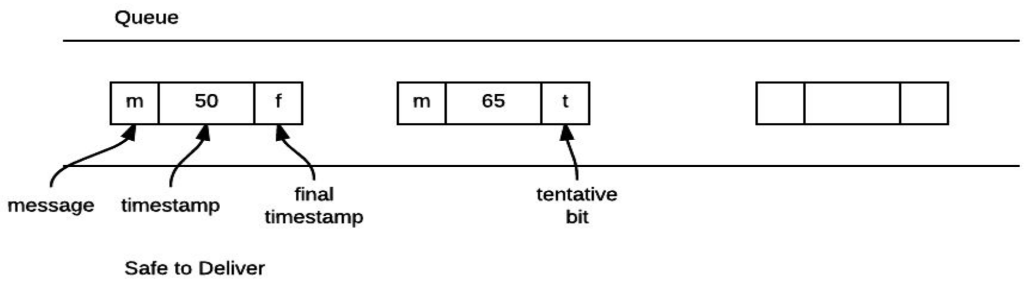
Now, let's look at it from a participant's (receiver's) perspective.



Ordered by timestamp, i.e., “total order”

Processes can deliver a message if its timestamp is smallest and it is final.

Let’s see another example:



Can you get a deadlock?
Every tentative message will definitely become final, so no deadlock.

How many messages/multicast?
G: group size
# of messages/delivery. Order is 3(g-1)

<u>Exercise</u>: Can we use Lamport’s algorithm for this? No. Why?

We can use Lamport’s algorithm for causal order?
($s_1 \rightarrow s_2$ .. $r_1 \rightarrow r_2$)

Skeen’s algorithm does not give causal order

<u>Exercise:</u> Come up with an example that maintains causal order.