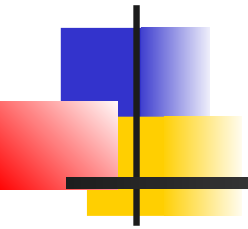
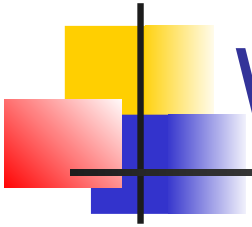


CAS CS 552
Intro to Operating Systems

Richard West

Distributed Systems Topics:

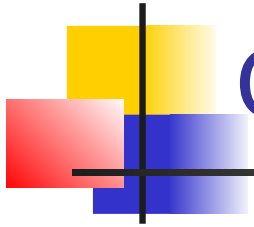
Distributed Shared Memory



What this covers...

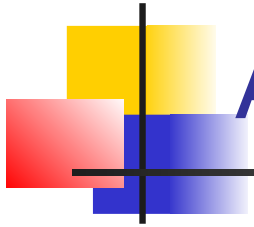
- Extend Lamport Clocks to Vector Timestamps
- Show how to order message delivery in group communication
- Show how message ordering relates to “consistency” of replicated data in distributed shared memory
- Compare various consistency protocols

- But first…



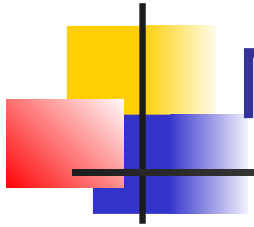
Group Communication

- RPC mechanism aimed at communication between two parties: client and server
- Sometimes need communication b/w multiple processes
  - e.g., client communicating w/ multiple file server processes in a distributed, fault tolerant filesystem
- Group communication: msg sent to a group must be received by all member processes
  - Group membership may change dynamically



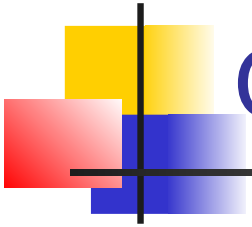
Atomicity (atomic broadcast)

- Msgs delivered to all group members or none at all
  - Like transactions, can have problems w/o atomicity, such as inconsistent data updates
  - e.g., P0 sends msg to group members P1, P2 & P3
  - P3 also sends a msg to P0, P1 & P2
  - P1 and P2 may receive the msgs from P0 and P3 in different orders
    - This can happen due to network effects



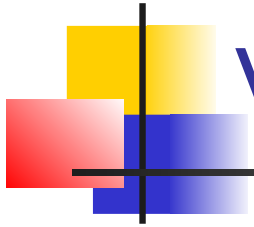
Message Ordering

- Global time ordering: all msgs delivered in exact order in which they were sent
  - Difficult since absolute global time does not hold across all processes (only relative)
  - Can we implement a system with weaker ordering?
    - Only guarantee consistent ordering of msgs across all processes (NOT necessarily their real-time sending order)



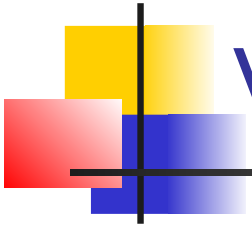
Causality

- Like Lamport's "happened before" relationship
- An event **a** is causally-related to event **b** if event **b** might have been influenced by event **a**
  - e.g., P0 sends msg1 to P1 and then P1 sends msg2 to P2;
  - msg1 & msg2 causally related since msg2 may have been sent as a consequence of msg1
- *Unrelated events* are said to be *concurrent*
- *Causal ordering:* Only ensure causally-related msgs/events are seen in the same order by all processes



Vector Timestamps 1/2

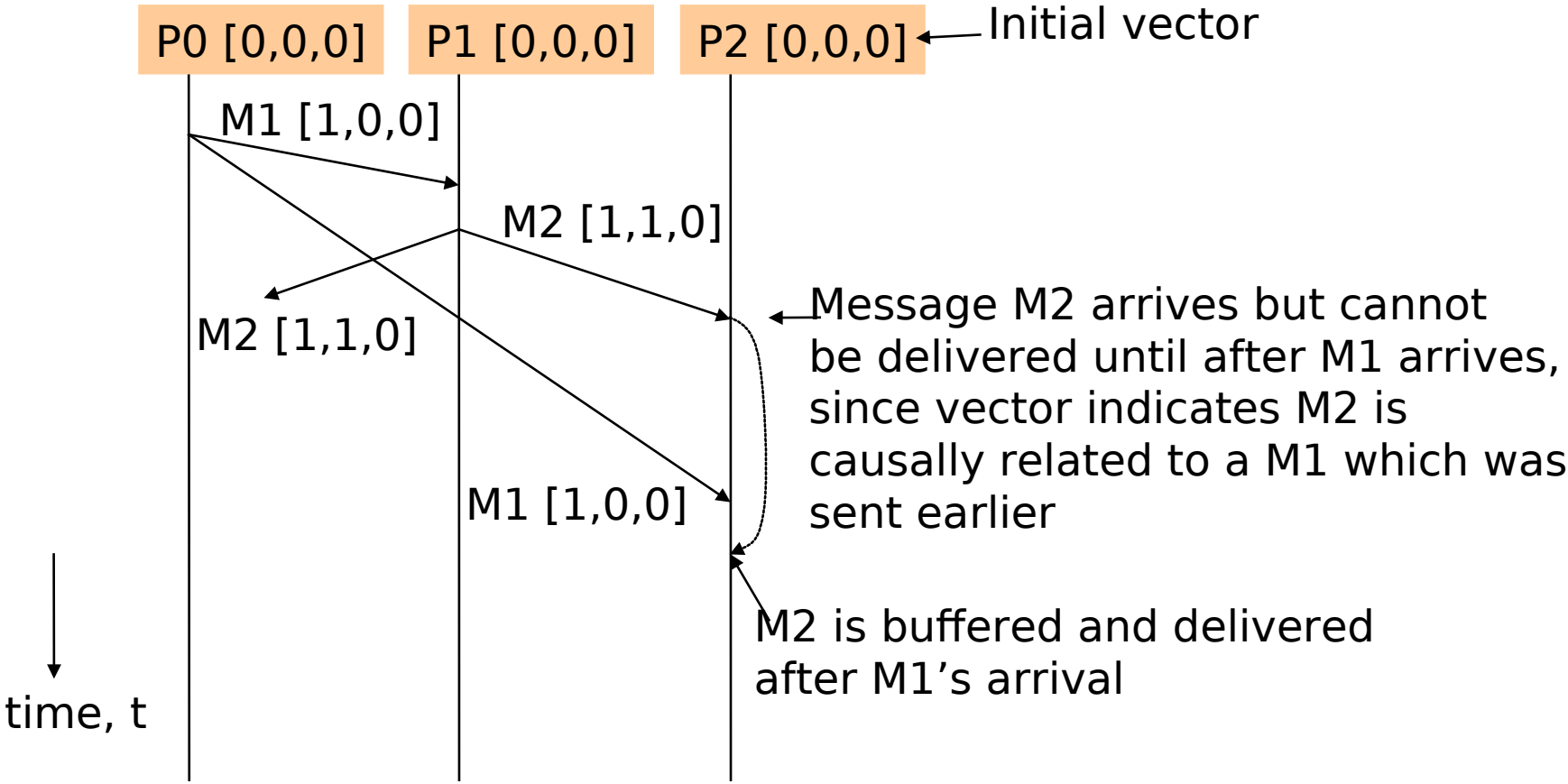
- In causal message ordering, each process maintains a vector with *n* components, one per group member
- *ith* component of vector is number of last message received in sequence from process *I*
- All vectors are initialized to zero
- When a process *i* has a message to send it increments the value of slot *i* (its own slot) in its vector and sends the vector in the message
- When a process receives a message it compares its own vector timestamp with the one in the received message



Vector Timestamps 2/2

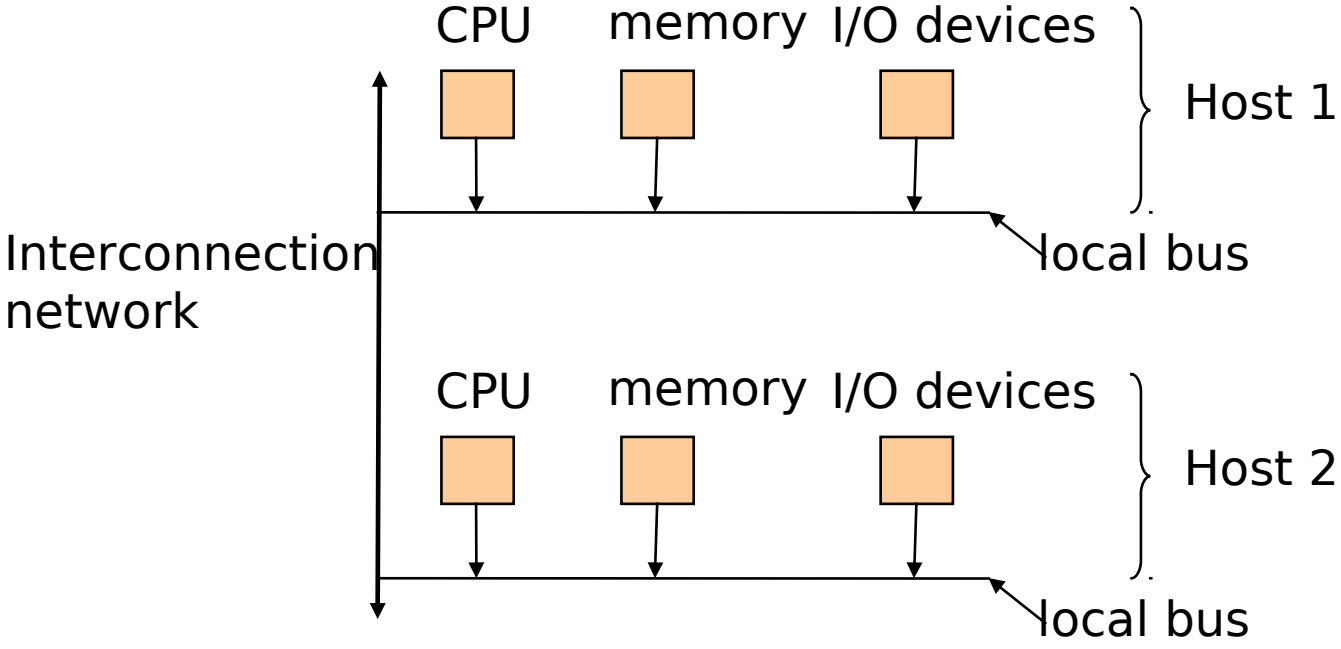
- Let Vi be the ith component of the vector timestamp in the incoming message
- Let Li be the ith component of the vector timestamp in the receiver process
- Suppose a message was sent by process j
- (1) 1st condition for acceptance of a message (i.e., message is ordered correctly):
  - Vj = Lj + 1 (i.e., this is next message in sequence from j)
- (2) 2nd condition for acceptance:
  - $Vi \leq Li \ \forall i \mid i \neq j$ (i.e., sender has not seen a message that the receiver has missed)
- If (1) and (2) hold, message is delivered, else it must be queued before other messages are delivered to the target process

Vector Timestamp Example

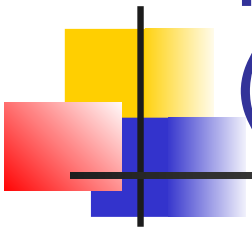


Distributed Shared Memory

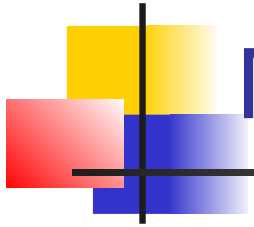
- Gives impression of a single physically shared memory accessible by all processors but is actually made of separate memories on different hosts



Non-uniform Memory Access (NUMA)



- Accessing remote memory is typically slower than local memory
  - Implies non-uniform memory access
  - e.g., BBN Butterfly, Cm*
- By replicating data in different memories can access local memory to retrieve data, thereby avoiding the cost of remote memory access
- Need to ensure copies of data items in different memories are *consistent*

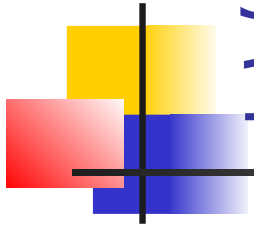


Memory Consistency

- All processors see the same (coherent) values of replicated data when necessary

- Strict Consistency
  - Any read to a memory location x returns the value stored by the most recent write operation to x
  - Impossible to enforce in general due to speed of light bounds

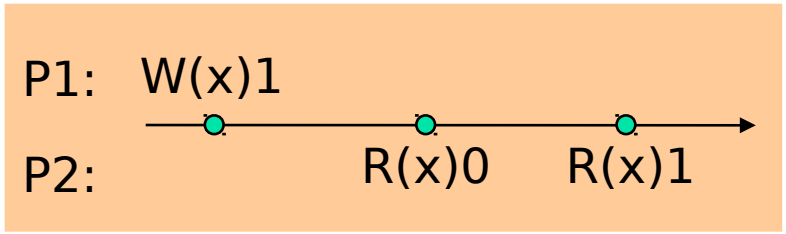
Sequential Consistency (Lamport 1979)



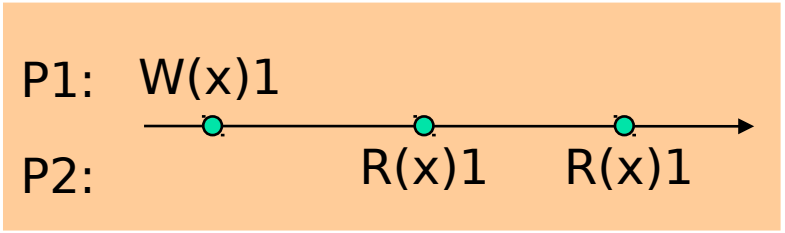
- Slightly weaker than strict consistency
- Lamport: “The result of any execution is the same as if the operations of all processors were executed in *some* sequential order, and the operations of each individual processor appear in this sequence in the order specified by its program”
  - i.e., any valid interleaving of read/write operations on data is acceptable, but all processors must see the same sequence of memory references

Example

- Let W(x)a => value a is written to variable x
- Let R(y)b => value b is read from variable y
- e.g.,

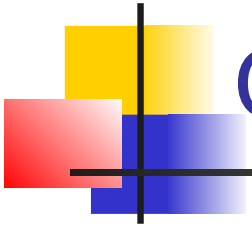


(a)



(b)

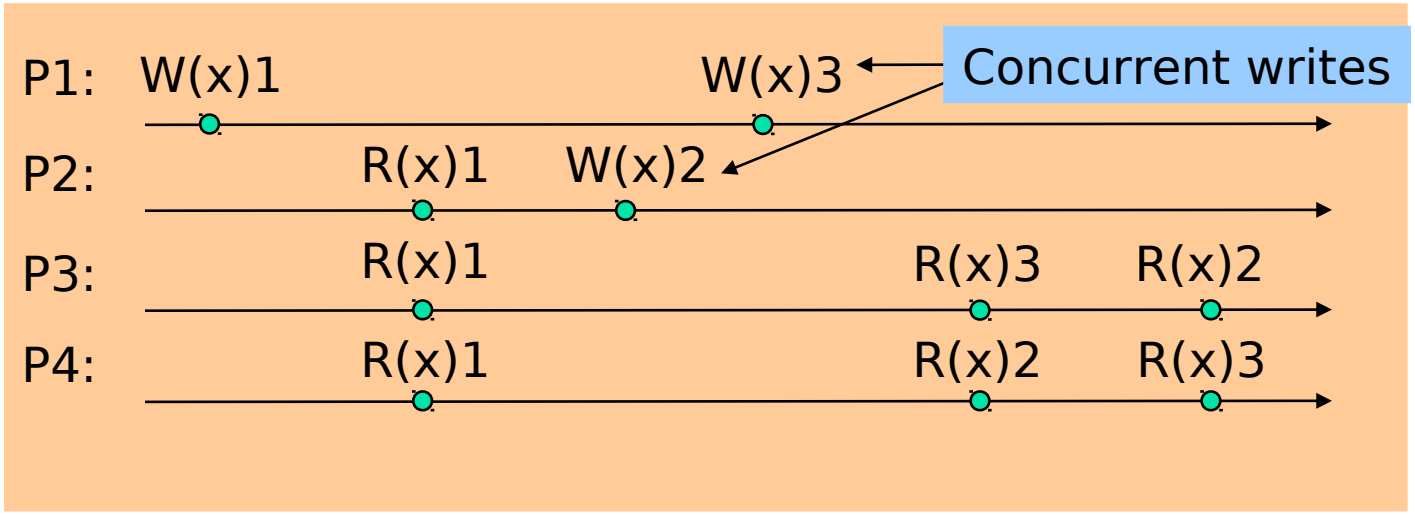
- Both (a) and (b) are sequentially consistent but (a) would not satisfy strict consistency since 1st read by P2 returns 0 which is after (in real-time) the write by P1



Causal Memory

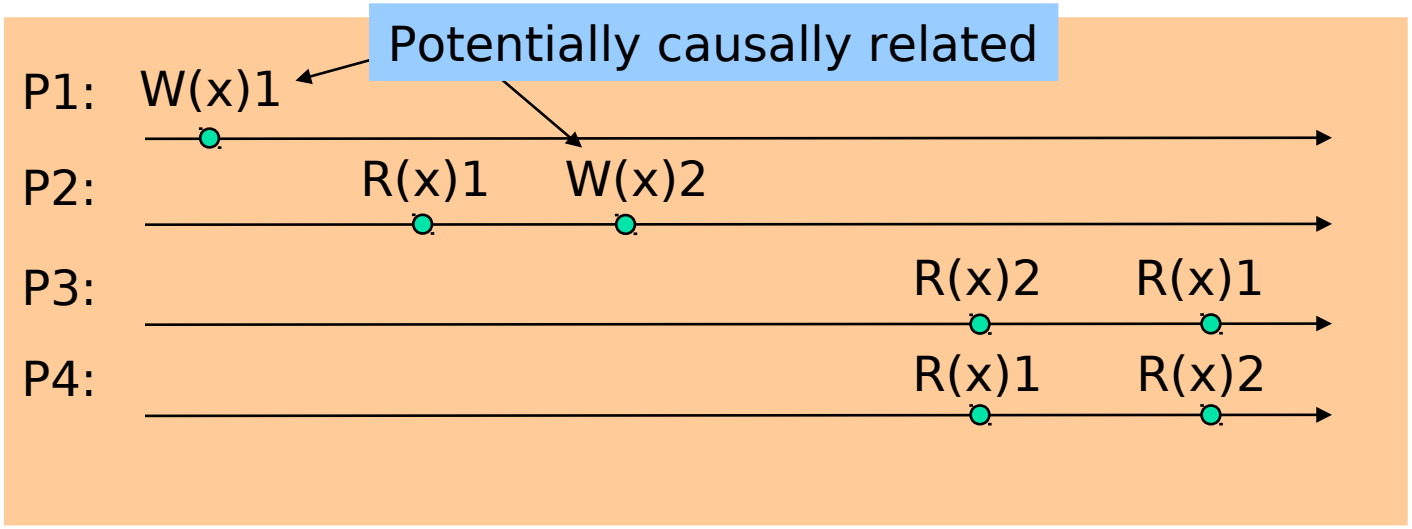
- Weakening of sequential consistency
- Writes that are potentially causally related must be seen by all processes in the same order
- Concurrent writes may be seen in different order on different machines

Example 1: Causal Memory



- This is a valid causal ordering but *not* a valid sequentially consistent ordering
  - With causal memory, processes (here P3 and P4) can see concurrent writes in any order

Example 2: Causal Memory



- The reads by P3 and P4 are not valid return values with causal memory
  - Due to potentially causally related writes by P1 and P2
  - W(x)1→R(x)1, R(x)1→W(x)2 so W(x)1→W(x)2
- If we removed R(x)1 from P2 the writes by P1 and P2 would be concurrent making this an acceptable causal memory ordering