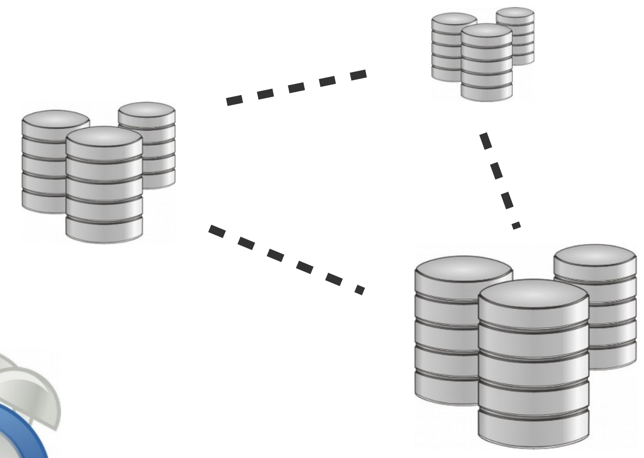
CS 470
Spring 2025

Mike Lam, Professor



Synchronization and Consistency

Content taken from the following:

"Distributed Systems: Principles and Paradigms" by Andrew S. Tanenbaum and Maarten Van Steen (Chapters 6, 7 and 11)

Various online sources

Synchronization

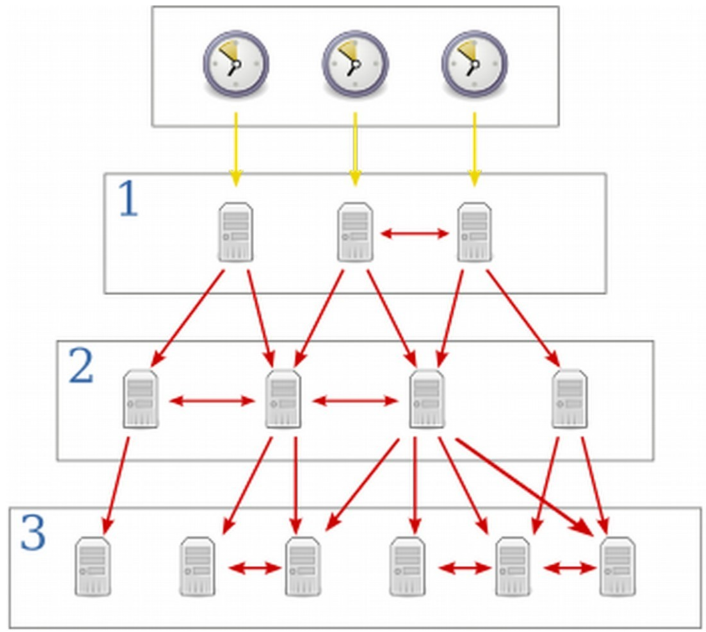
- In a shared-memory system:
  - Core mechanism: mutual exclusion
  - Conditions, semaphores, and barriers
- In a distributed-memory system:
  - Core mechanism: message passing
  - Coordinated clocks
    - Absolute vs. logical
  - Election and consensus algorithms
  - Consistency models and protocols

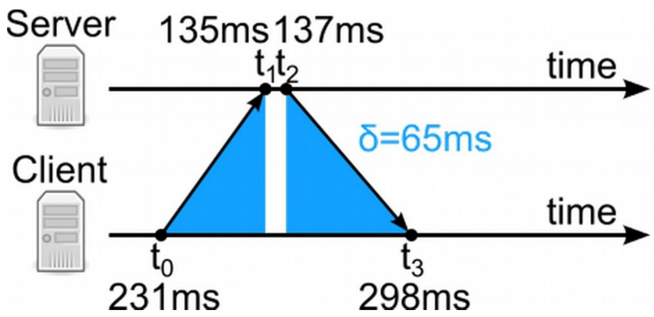
Clocks / Timers

- Measuring time
  - Movements of sun, moon, and stars
  - Unwinding of wound spring
  - Quartz crystal oscillating under tension
  - Energy transitions of a caesium 133 atom
- Synchronizing absolute clocks
  - Calendars and leap year/second adjustments
    - Coordinated Universal Time (UTC)
  - Clock skew
    - Network Time Protocol (NTP)

Network Time Protocol

- Reference clocks (hardware-based)
- Stratums 1-15 and 16 (unsynced)
- 64-bit time values (<1 ns resolution)





Time offset:

$$\theta = \frac{(t_1 - t_0) + (t_2 - t_3)}{2}$$

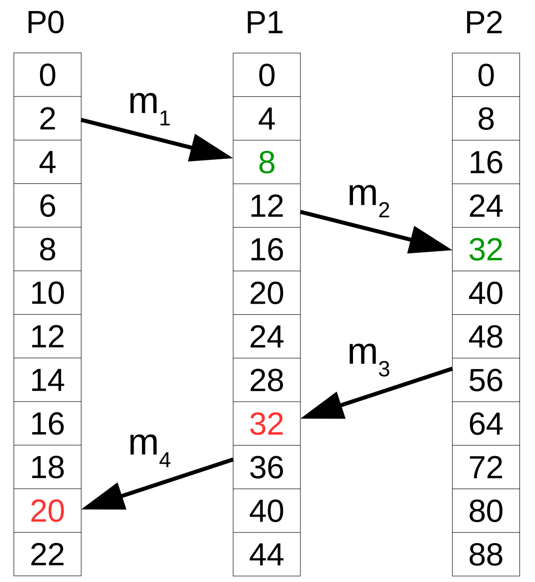
Round-trip delay:

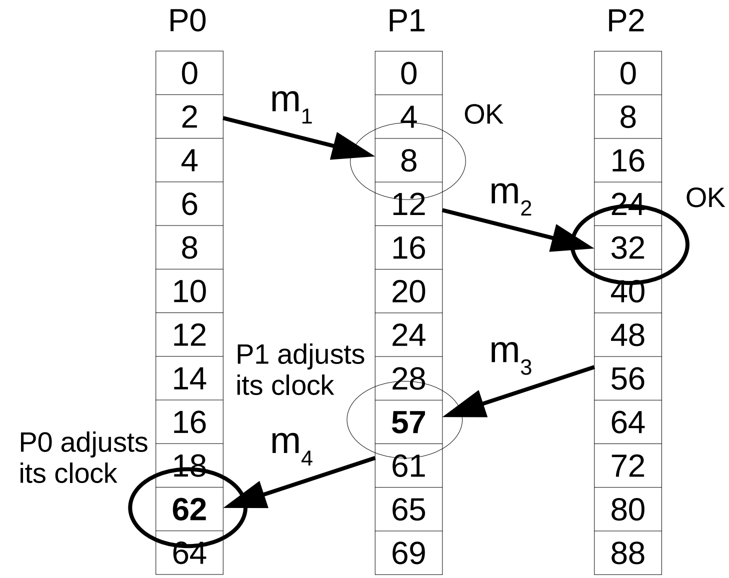
$$\delta = (t_3 - t_0) - (t_2 - t_1)$$

from https://en.wikipedia.org/wiki/Network_Time_Protocol

Logical clocks

- Lamport clocks / timestamps
  - Invented by Leslie Lamport in 1978
  - Core notion: "happens-before" (total ordering)
    - Assigns clock value C(x) to any event x
    - Increment local clock before sending
    - Include local clock when sending
  - Adjust local clock after communications
    - Must preserve "happens-before" ordering
    - Always forwards—never backwards!
  - If a happened before b, then C(a) < C(b)
    - Converse is not necessarily true!
    - Does not capture any notion of causality

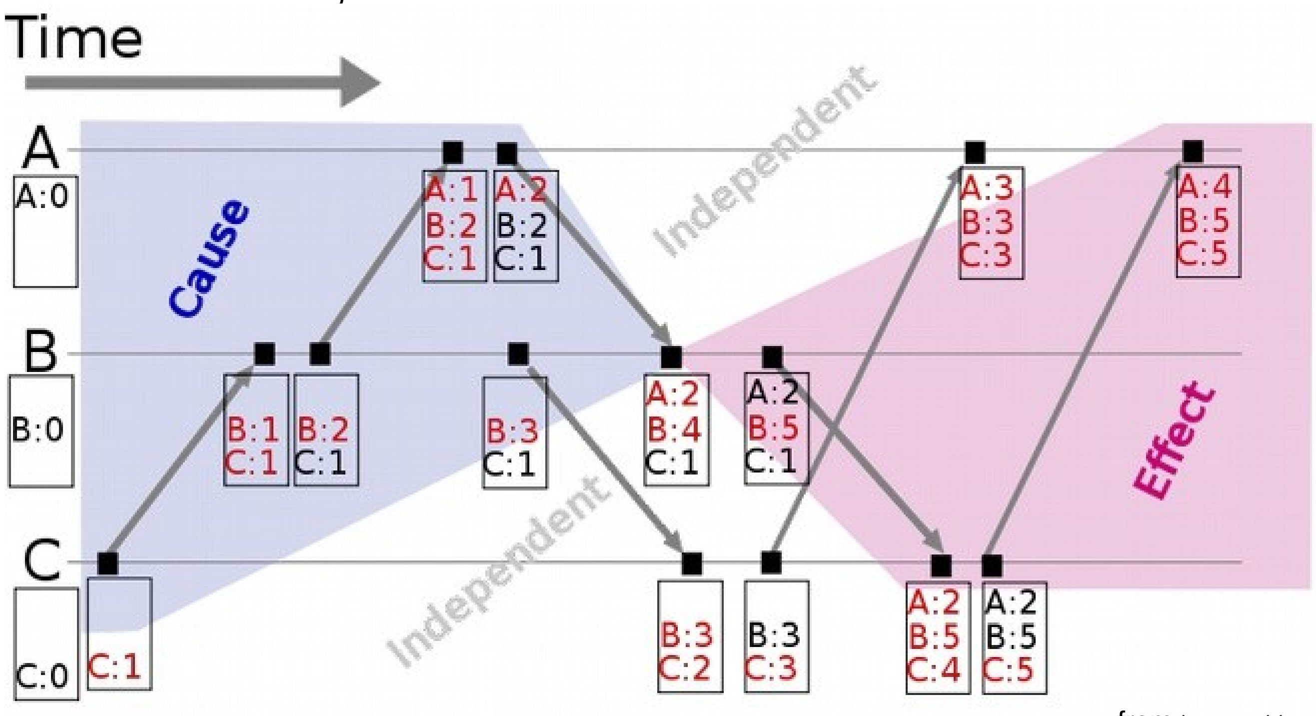




For more info:
http://dl.acm.org/citation.cfm?id=359563

Vector clocks

- Vector clocks restore a notion of causality (partial ordering)
  - Keep a vector of clock values instead of only one
  - VC_i is the logical clock at process P_i
  - $VC_i[j] = k$ means that P_i knows that k events have occurred at P_j (i.e., P_i's knowledge of P_j's local time), any of which could have causality influence



from `https://en.wikipedia.org/wiki/Vector_clock`

Distributed mutual exclusion

- Clocks provide *time*-based synchronization
- What about *task*-based synchronization?
- How can we implement mutual exclusion in a distributed system?

Distributed mutual exclusion

- Token-based (often used in ring networks)
    - Simple; slow; susceptible to lost tokens
- Permission-based
    - Centralized (single coordinator)
        - Easy to implement; single bottleneck and point of failure
    - Decentralized (multiple coordinators, need majority vote)
        - More resilient; can be slow; possibility of starvation

Election algorithms

- If a coordinator is needed, there are various election strategies available to choose one
- Bully algorithm
  - Always defer to higher-numbered nodes
- Ring algorithm
  - Enforce one-way election traffic
- Wireless algorithms
  - Choose the *best* coordinator (e.g., CPU speed, battery life, etc.)

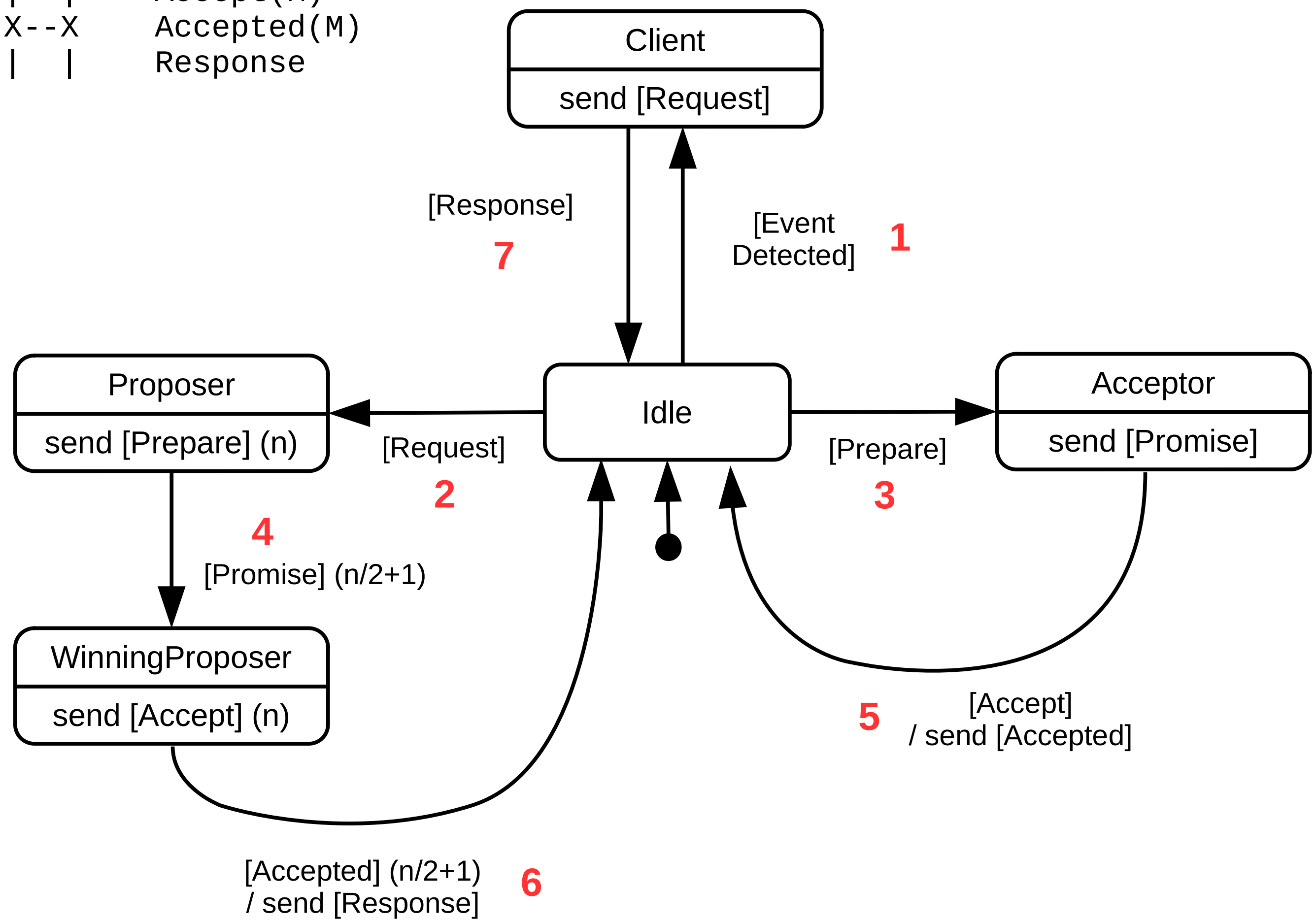
Distributed consensus

- Elections (and related auctions) are a specialized form of the general problem of determining consensus in a distributed system
- Paxos protocol: two-phase rounds
  - Prepare / promise: A proposer creates a proposal with value N larger than any value it has previously used and sends it to a quorum of acceptors, who respond with a promise to ignore future proposals with a value less than N
  - Accept / accepted: If a proposer receives enough promises, it sets a final value M for its proposal and sends it to a quorum of acceptors, who accept it if M is greater than any other proposals it has promised to
  - Real protocol has multiple ways to handle failures and lack of consensus

```
   Client   Proposer      Acceptors
1    |         |          |  |  |
2    X-------->|          |  |  |     Request
3    |         X--------->|->|->|     Prepare(N)
4    |         |<---------X--X--X     Promise(N)
5    |         X--------->|->|->|     Accept(M)
6    |         |<---------X--X--X     Accepted(M)
7    |<--------X          |  |  |     Response
```

Distributed consensus

```
     Client    Proposer      Acceptors
1      |          |            |  |  |
2      X-------->|            |  |  |    Request
3      |          X--------->|->|->|    Prepare(N)
4      |          |<---------X--X--X    Promise(N)
5      |          X--------->|->|->|    Accept(M)
6      |          |<---------X--X--X    Accepted(M)
7      |<--------X            |  |  |    Response
```



Replication

- All of these protocols require a lot of communication
  - Communication is expensive!
- Alternative: keep redundant data
  - Replica: a copy of data
    - In a distributed system, every process could have a replica
  - Goal: improved availability/locality and therefore performance
    - Related concepts: mirroring and caching
    - Relieve single-node access bottlenecks

Replicas

- Server-initiated (e.g., mirroring)
  - Updates are pushed to other replicas
- Client-initiated (e.g., caching)
  - Updates are pulled from other replicas
  - Write-through vs. write-back
- Peer-to-peer
  - Nodes have symmetric roles
  - Requires well-defined protocol for enforcing consistency
- Issue: keeping replicas consistent
  - Propagating updates
  - Events (reads/writes) will arrive at different times
  - But maybe we're ok with some inconsistency

Replication and consistency

- Theme: **loosen consistency constraints** to *decrease communication overhead*
  - Tradeoff: performance vs. consistency

Replication and consistency

- CS 374 pop quiz: What does ACID stand for in the context of database consistency?
  - A. Accessible, Continuous, Integral Data
  - B. Atomic, Consistent, Isolated, Durable
  - C. Atomic, Constant, Integrated, Data-agnostic
  - D. Agnostic, Continuous, Isolated, Durable
  - E. Accessible, Consistent, Integrated Database

Replication and consistency

- Theme: **loosen consistency constraints** to *decrease communication overhead*
  - Tradeoff: performance vs. consistency

Traditional databases:
ACID - Atomic, Consistent, Isolated, Durable

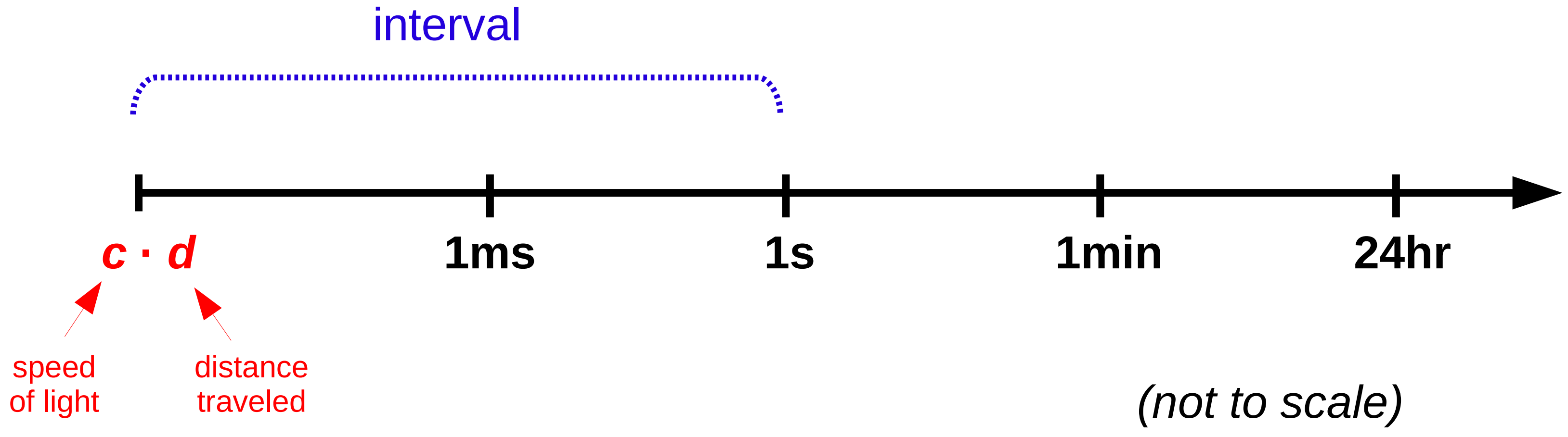
Distributed systems:
BASE - Basically Available, Soft-state, Eventually consistent

Replication

- Consistency model: contract between entities and data stores
  - If the entities follow the rules, the data store will be consistent
- Data-centric models (global view)
  - Strict / continuous consistency (absolute time)
  - Sequential consistency (logical time)
  - Causal consistency (logical causality)
- Client-centric models (local view)
  - Monotonic reads
  - Monotonic writes
  - Read-your-writes
  - Writes-follow-reads

Strict / continuous consistency

- All events are seen "instantaneously" by all nodes
  - Issue: speed of light (~3 x 10^8 m/s) prevents instantaneous updates, especially in large-scale distributed systems
  - To be practical, designate an interval of allowable deviation



Sequential consistency

- Every node sees events in the same order
  - Events must have a total order (i.e., they must be linearizable)
  - Important: a particular node need not see ALL events
    - But the order of the ones it sees must not violate the total order
  - Notation: "W(x)a" means "write value a to item x"
    - (corresponding notation for reads)

P0:	W(x)a				
P1:		W(x)b			
P2:			R(x)b		R(x)a
P3:				R(x)b	R(x)a

Sequentially-consistent

P0:	W(x)a				
P1:		W(x)b			
P2:			R(x)b		R(x)a
P3:				R(x)a	R(x)b

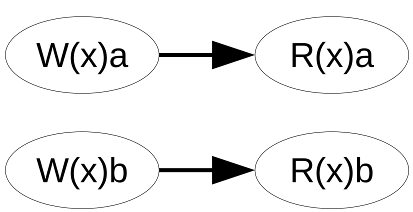
NOT sequentially-consistent

Causal consistency

- Causally-related events must be seen in order
  - Reads are causally-related to corresponding writes
  - Writes are causally-related to previous operations on the same node
  - Can be implemented using vector clocks
  - To verify, build global causality chain and check each process's view

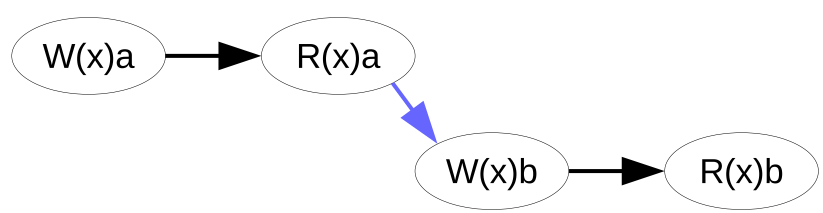
P0:	W(x)a			
P1:		W(x)b		
P2:			R(x)b	R(x)a
P3:			R(x)a	R(x)b

Causally-consistent



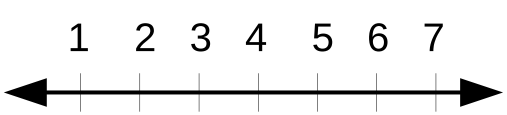
P0:	W(x)a				
P1:		R(x)a	W(x)b		
P2:				R(x)b	R(x)a
P3:				R(x)a	R(x)b

NOT causally-consistent

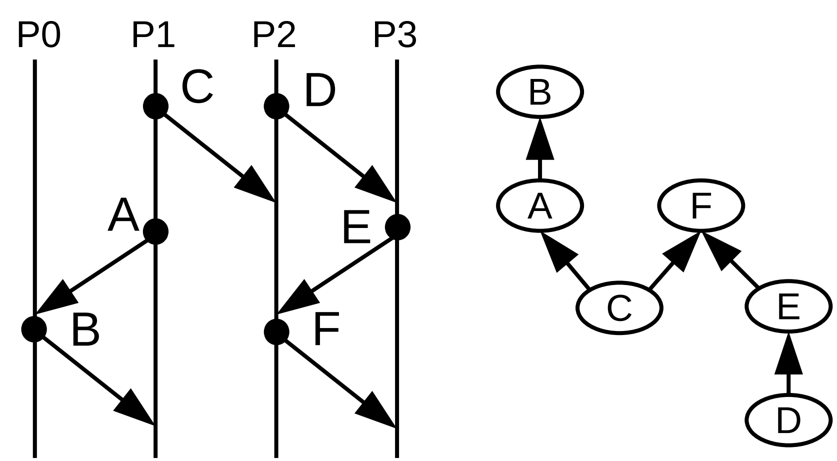


Partial vs. total ordering

- Ordering: definition of “<” operator
  - Usually over pairs of entities (for us, messages)
  - Total ordering: definition of “<” for **all** pairs (w/ transitivity)
    - Depicted graphically using a line
  - Partial ordering: definition of “<” for **some** pairs (also w/ transitivity)
    - Depicted graphically using a graph or lattice



“<” ≡ “less-than”
(total ordering)



“<” ≡ “happens-before” (partial ordering)

Implication

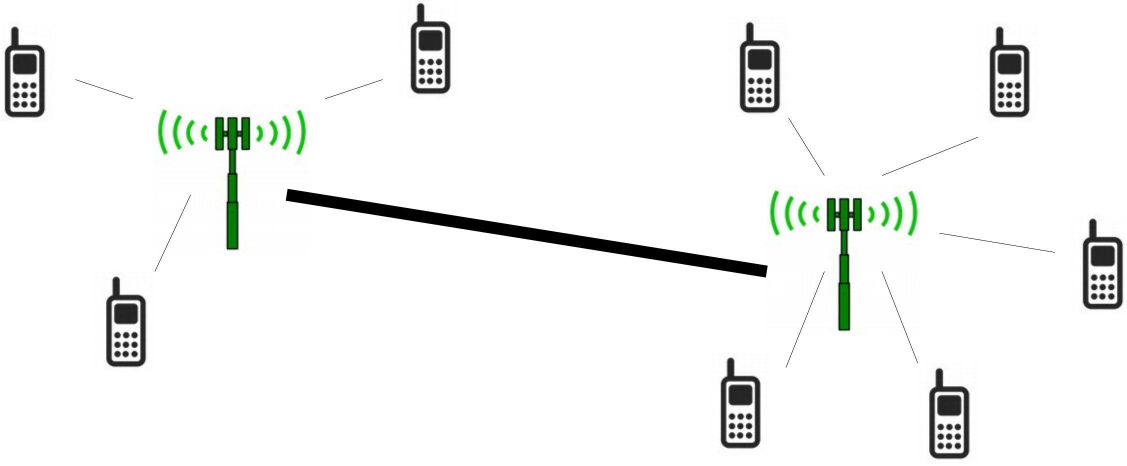
- Sequential consistency implies causal consistency
  - There is no way for the partial ordering of causal consistency to contradict the total ordering implied by sequential consistency
    - Both properties (writes before reads on same data & strict ordering for events on single processes) used to build the partial ordering are already enforced by any valid total ordering
  - Thus, every sequentially-consistent sequence must also be causally-consistent
  - Colloquially: causal consistency is ***looser*** than sequential consistency

Client-centric consistency

- Previous models focused on a **global** view of data
  - Sometimes called data-centric consistency models
- In a distributed system, we may only be interested in the **local** view at any given node
  - This motivates client-centric consistency models

Client-centric consistency

- Original application: Bayou database system for mobile computing
  - Developed in mid-1990s
  - Massive number of replicas
  - Multiple networks and unreliable connectivity
  - Data-centric, global consistency models are infeasible
  - Theme: loosen the constraints!
  - **Four** different consistency models (not mutually exclusive)



For more info:
`http://dl.acm.org/citation.cfm?id=504497`

Monotonic reads / writes

- Monotonic reads: if a process reads X, any successive read to X will see the same value or a more recent one
  - I.e., the process will never see an older version
  - E.g., distributed email database (messages shouldn't disappear when viewing a thread on the same client)

- Monotonic writes: if a process writes X, any successive write to X will see the effect of the first write
  - I.e., newer writes must wait for older ones to finish
  - E.g., local wiki edits (should never edit an older version than the most recent the client has) – may still introduce merge conflicts with respect to **other** clients' changes!

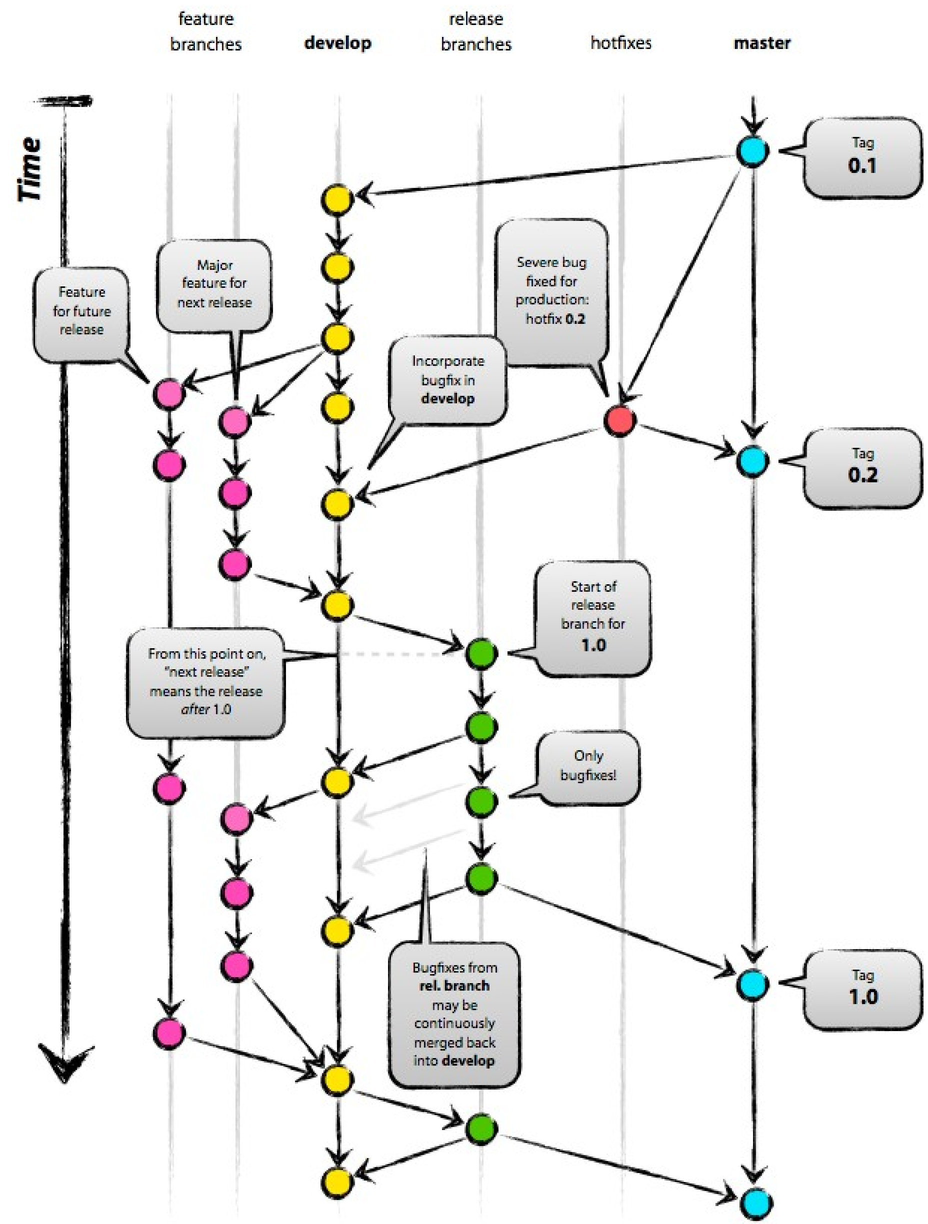
Read-your-writes / Writes-follow-reads

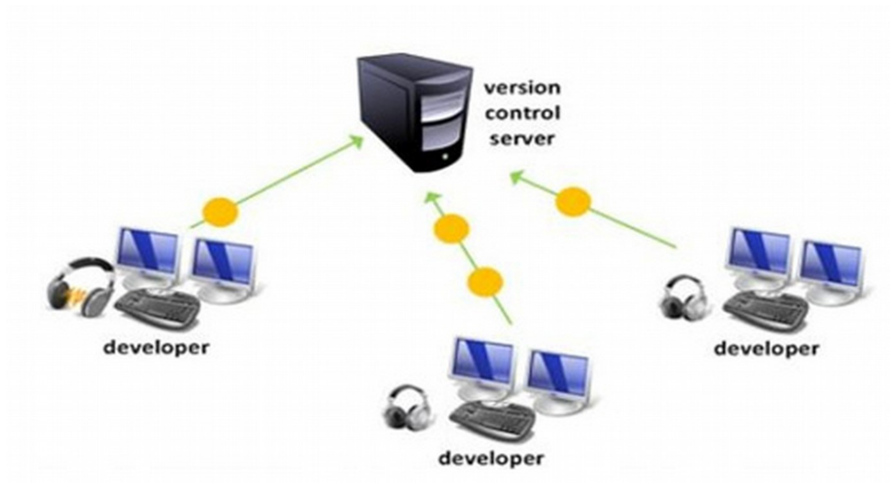
- Read-your-writes: if a process writes X, any successive read to X will see the effect of the write
  - I.e., reads will never see old versions
  - Closely related to monotonic reads
  - Systems that often temporarily lack this consistency:
    - Retrieving websites
    - Updating passwords
- Writes-follow-reads: if a process reads X, any successive write to X will see the same value or a more recent one
  - I.e., writes will never see old versions
  - E.g., posts to an email list

Consistency protocols

- Continuous consistency protocols
  - Bounding numerical deviation (# of updates)
  - Bounding staleness deviation (time of updates)
- Primary-based protocols
  - Primary: one replica that coordinates all writes for a data item
  - Remote-write: forward all writes to primary (similar to write-through)
  - Local-write: periodic updates sent to primary (similar to write-back)
- Replicated-write protocols
  - Active replication: multicast updates to all replicas
    - Need a reliable and efficient multicast protocol
  - Quorum-based voting: replicas vote on updates to replicas
    - Need a distributed voting/consensus protocol

Distributed version control





git

mercurial