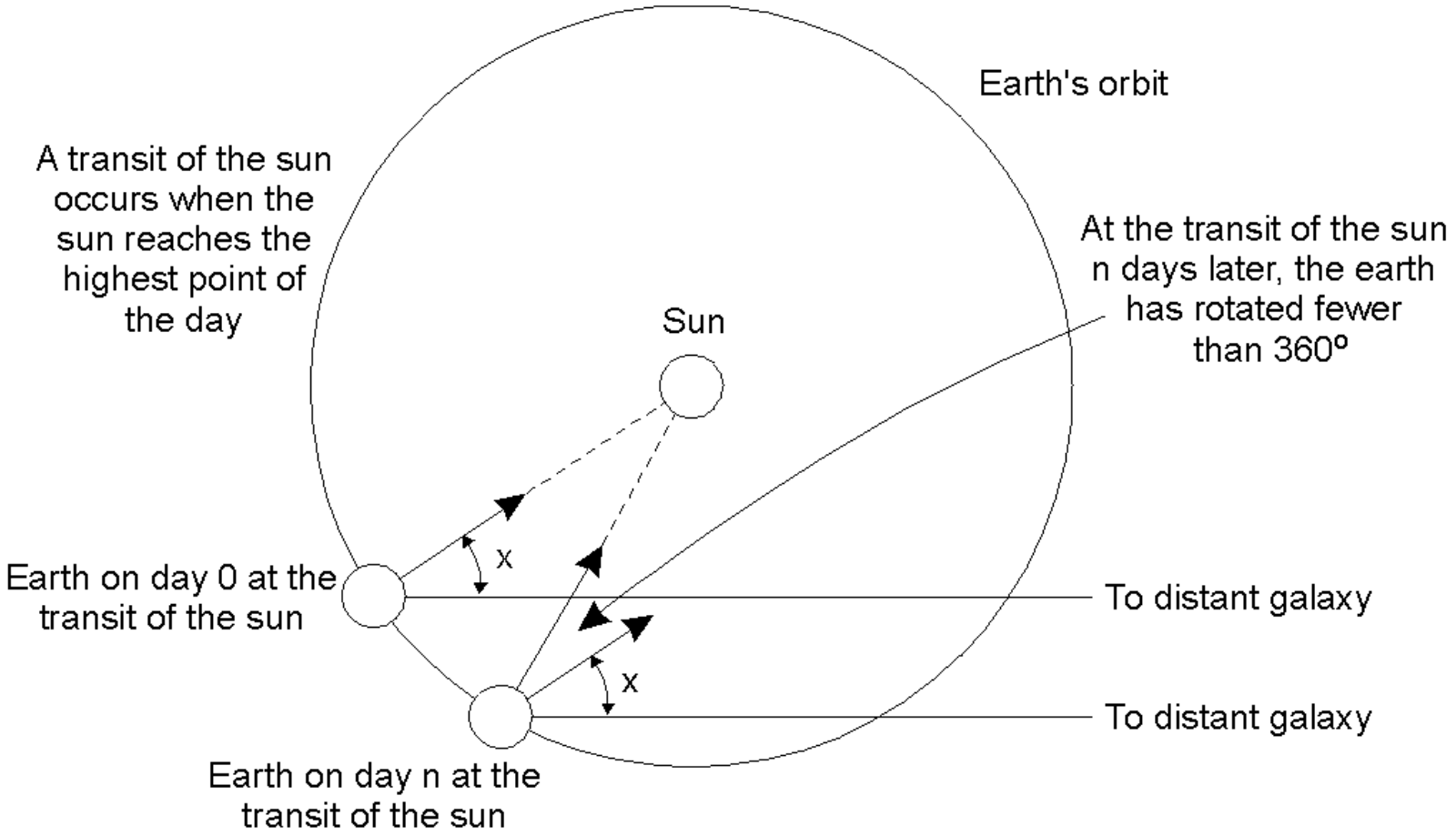
# Canonical Problems in Distributed Systems

- Time ordering and clock synchronization
- Global state – distributed snapshot
- Leader election
- Mutual exclusion
- Distributed transactions
- Deadlock detection

# Physical Clocks

- It is impossible to guarantee that crystals in different computers all run at exactly the same frequency. This difference in time values is **clock skew**.
- "Exact" time was computed by astronomers
  - The difference between two transits of the sun is termed a **solar day.** Divide a solar day by 24*60*60 yields a **solar second.**
- However, the earth is slowing! (35 days less in a year over 300 million years)
- There are also short-term variations caused by turbulence deep in the earth's core.
  - A large number of days **(n)** were used to the average day length, then dividing by 86,400 to determine the **mean solar second.**

# Physical Clocks




Computation of the mean solar day.

# Physical Clocks

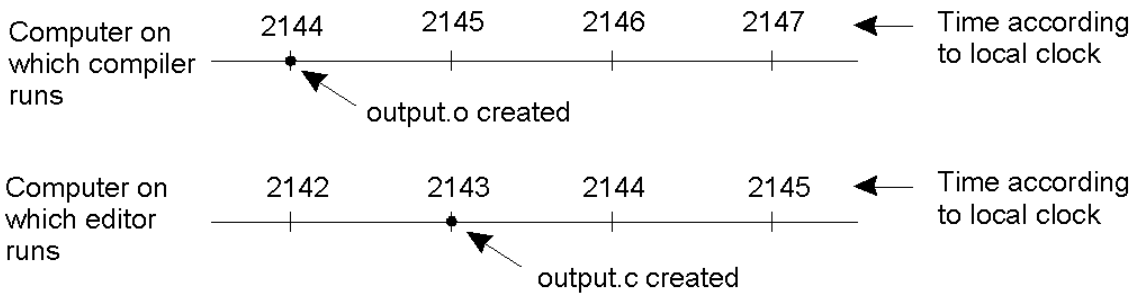
- Physicists take over from astronomers and count the transitions of cesium 133 atom
  - 9,192,631,770 cesium transitions == 1 solar second
  - 50 International labs have cesium 133 clocks.
  - The *Bureau Internationale de l'Heure (BIH) averages* reported clock ticks to produce the **International Atomic Time (*TAI*).**
  - The TAI is mean number of ticks of cesium 133 clocks since midnight on January 1, 1958 divided by 9,192,631,770 .

# Physical Clocks

- To adjust for lengthening of mean solar day, **leap seconds** are used to translate TAI into **Universal Coordinated Time (*UTC*).**
- UTC is broadcast by NIST from Fort Collins, Colorado over shortwave radio station WWV. WWV broadcasts a short pulses at the start of each UTC second. **[*accuracy 10 msec.*]**
- GEOS (Geostationary Environment Operational Satellite) also offer UTC service. **[*accuracy 0.5 msec.*]**

# Clock Synchronization

- Time in unambiguous in centralized systems
  - System clock keeps time, all entities use this for time
- Distributed systems: each node has own system clock
  - Crystal-based clocks are less accurate (1 part in million)
  - *Problem:* An event that occurred after another may be assigned an earlier time

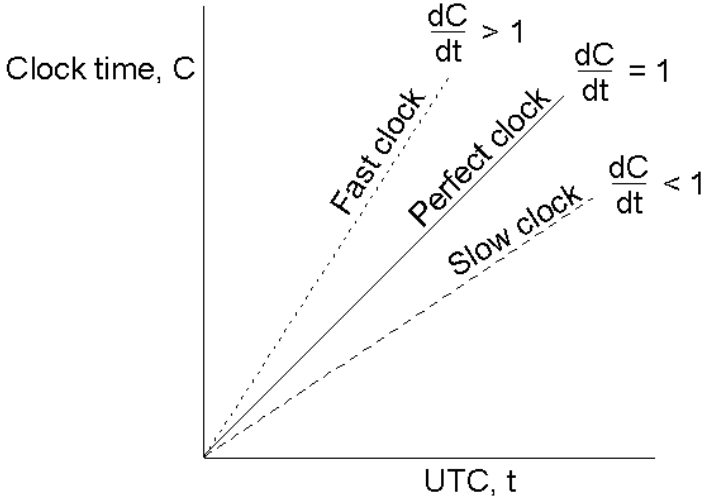


# Physical Clocks: A Primer

- Accurate clocks are atomic oscillators (one part in $10^{13}$)
- Most clocks are less accurate (e.g., mechanical watches)
  - Computers use crystal-based clocks (one part in million)
  - Results in *clock drift*
- How do you tell time?
  - Use astronomical metrics (solar day)
- Coordinated universal time *(UTC)* – international standard based on atomic time
  - Add leap seconds to be consistent with astronomical time
  - UTC broadcast on radio (satellite and earth)
  - Receivers accurate to 0.1 – 10 ms
- Need to synchronize machines with a master or with one another

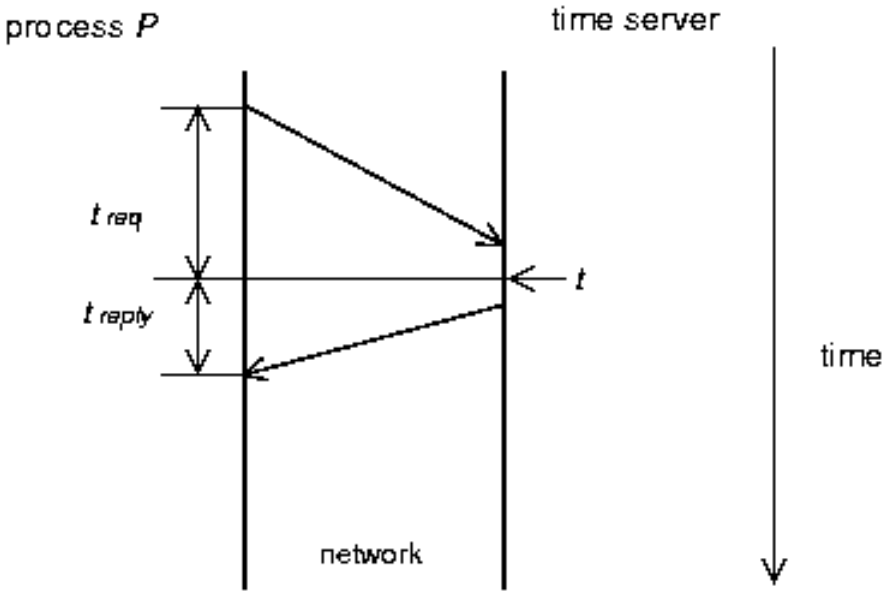
# Clock Synchronization

- Each clock has a maximum drift rate $\rho$
  - $1-\rho <= dC/dt <= 1+\rho$
  - Two clocks may drift by $2\rho\ \Delta t$ in time $\Delta t$
  - To limit drift to $\delta$ => resynchronize every $\delta/2\rho$ seconds



# Cristian's Algorithm

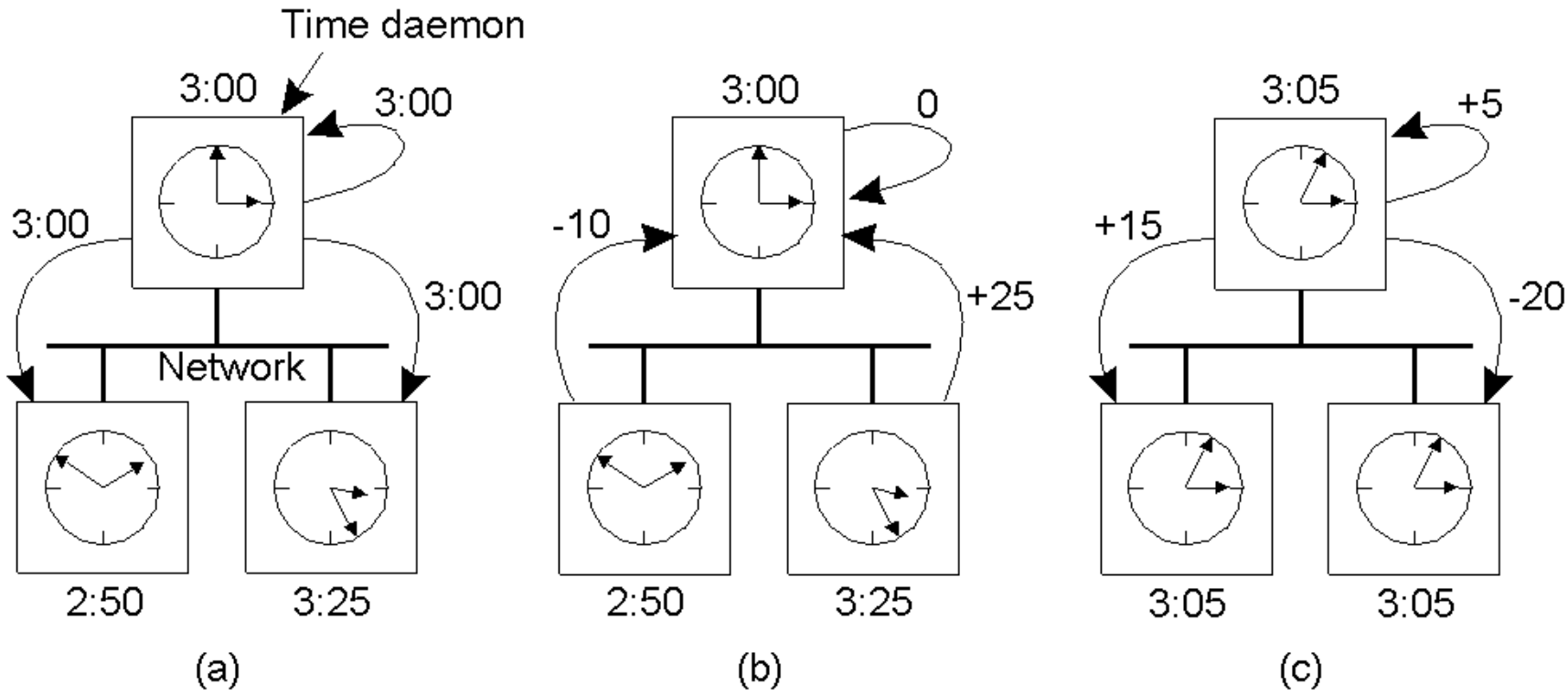
- Synchronize machines to a *time server* with a UTC receiver
- Machine P requests time from server every $\delta/2\rho$ seconds
  - Receives time *t* from server, P sets clock to $t+t_{reply}$ where $t_{reply}$ is the time to send reply to P
  - Use $(t_{req}+t_{reply})/2$ as an estimate of $t_{reply}$
  - Improve accuracy by making a series of measurements



# Berkeley Algorithm

- Used in systems without UTC receiver
  - Keep clocks synchronized with one another
  - One computer is *master*, other are *slaves*
  - Master periodically polls slaves for their times
    - Average times and return differences to slaves
    - Communication delays compensated as in Cristian's algo
  - Failure of master => election of a new master

# Berkeley Algorithm




a) The time daemon asks all the other machines for their clock values
b) The machines answer
c) The time daemon tells everyone how to adjust their clock

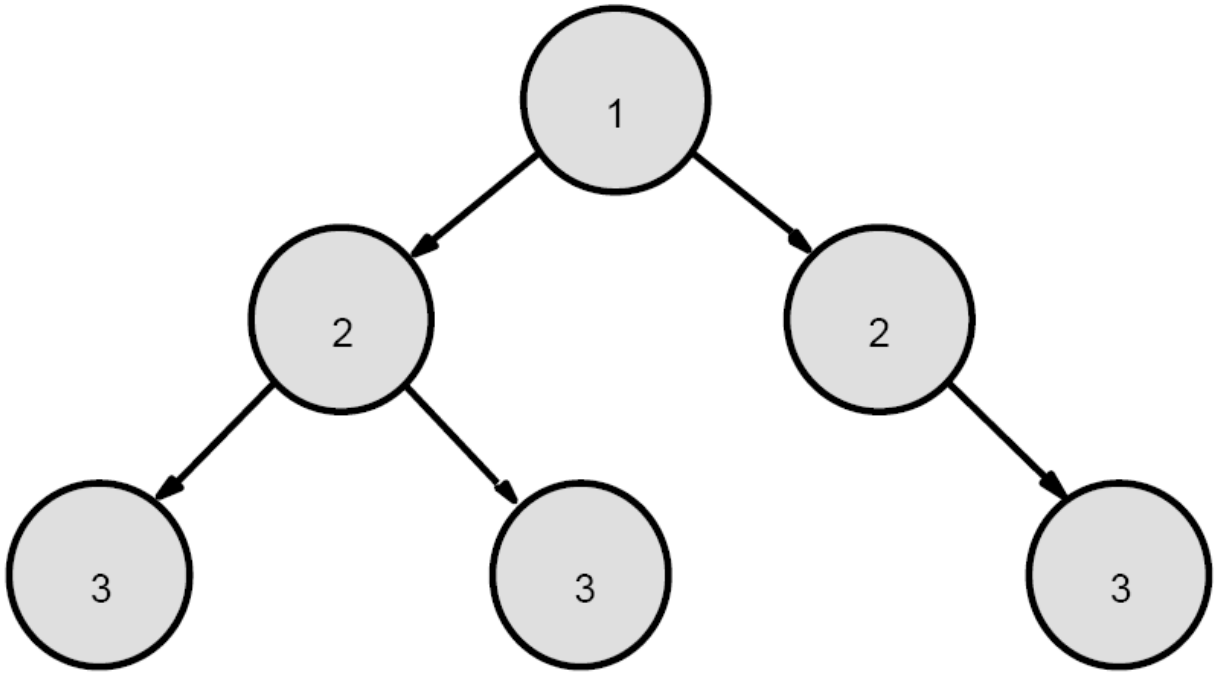
# Network Time Protocol (NTP)

- Intended for use across Internet
- Design aims
  1. Provide a service enabling clients across the Internet to be synchronised accurately to UTC
  2. Provide a reliable service that can survive lengthy losses of connectivity
  3. Enable clients to resynchronise sufficiently frequently to offset the rates of drift found in most computers
  4. Provide protection against accidental interference with the time service, whether malicious or accidental

# Network Time Protocol (NTP)

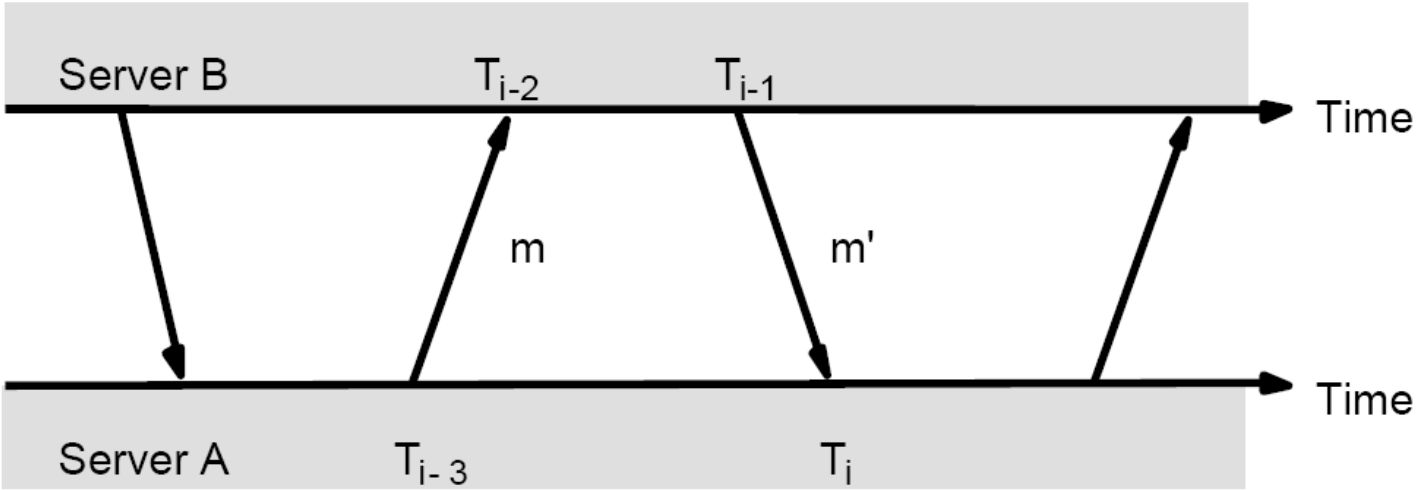
- Provided by a distributed network of servers
- Servers connected in a logical hierarchy called a *synchronisation subnet*, whose levels are *strata*
- Primary servers (stratum 1 - root)
  - Connected directly to a UTC source
- Secondary servers (stratum 2 - N)
  - Synchronised, ultimately, with servers at next highest stratum
  - Servers at stratum 2 synchronise with primary servers

# Network Time Protocol (NTP)




Note: Arrows denote synchronization control, numbers denote strata.

# Network Time Protocol (NTP)



# Network Time Protocol (NTP)

- If the true offset of B relative to A is *o* and the actual transmission times for *m* and *m'* are *t* and *t'* then

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

which gives us…

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

and

$$o = o_i + (t' - t) / 2$$

$$\text{where } o_i = (T_{i-2} - T_{i-3} + T_{i-1} - T_i) / 2$$

# Network Time Protocol (NTP)

- Using the fact that $t, t' \geq 0$, then we can show that
$$o_i - d_i/2 \leq o \leq o_i + d_i/2$$
- Therefore, $o_i$ is an estimate of the offset, and $d_i$ is a measure of the accuracy of the estimate
- NTP servers apply data filtering algorithm to successive pairs of $<o_i, d_i>$ pairs which estimates $o$ and calculates the quality of this estimate as a statistical quantity called the *filter dispersion*
- The higher the filter dispersion the lower the quality
- NTP servers select the most reliable peers to synchronise with – typically selecting ones with lower stratum numbers and/or lower synchronisation dispersion (filter dispersion with primary server – made available in exchange messages).

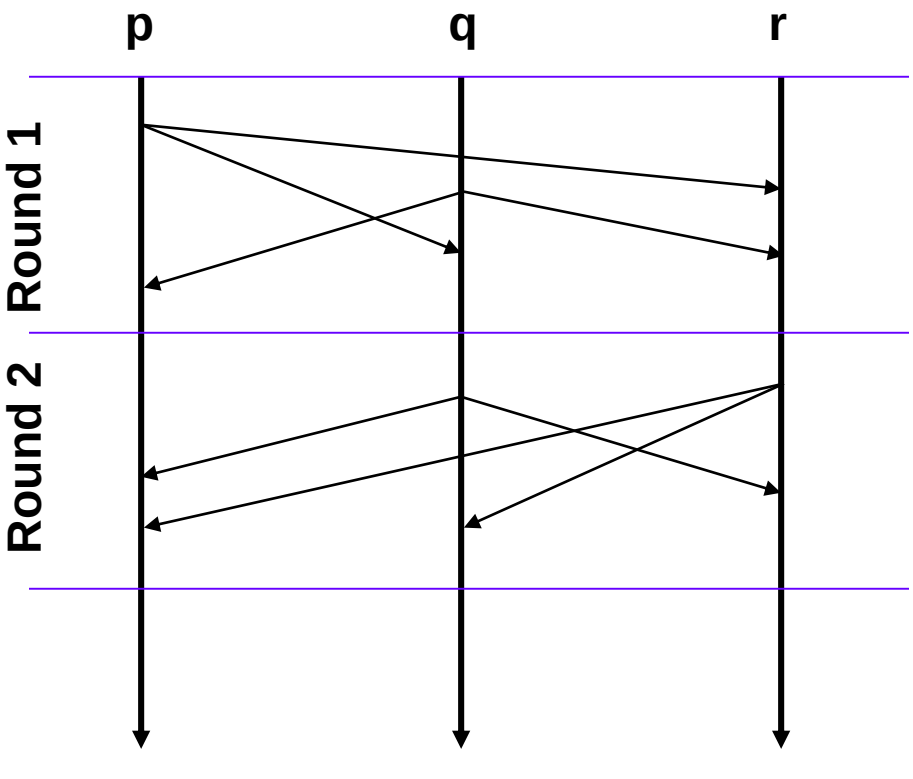
# Definitions
# Distributed System

- A set of autonomous computing nodes that cooperate to achieve a well defined goal
- Appears to its users as a single computing system
- Each node
  - Performs local computation steps
  - Cooperates with others through message passing

# Synchronous Distributed System

- The first kind of distributed system does synchronous execution
- Computations proceed in rounds (lock steps)
- We know an upper bound on the time needed to perform a local step at each node
- We know an upper bound on the time required for a message to move from the sender to a receiver
- Timeouts can be used for failure detection
  - This is the main strength of the synchronous model
  - Unfortunately, we usually won't be able to use this model

# Synchronous Distributed System




- **A Round**
- Send messages out
- Receive messages sent in this round
- Change your state

# Asynchronous Distributed System

- A more common kind of distributed system does asynchronous execution
- We **don't** know an upper bound on the time needed to perform a local step at each node
- We **don't** know an upper bound on the time required for a message to move from the sender to a receiver. But we **do** know that this time is finite.

- There are also partially synchronous distributed systems (see the book of Nancy Lynch 1996)

# Notations and Assumptions

- $A_1$: No shared variables among processes
- $A_2$: One or more executing threads in a process
- $A_3$: Communication is by message passing
  - `Send Action(Destination; Params)`
  - Sending a message is non-blocking

# Notations and Assumptions

- $A_4$: Algorithms are event-driven
  - Reaction upon receipt of an event
    (nodes execute events)
  - Events
    - Sending/receiving a message
    - Internal events
    - An event is buffered until it is handled
    - Dedicated thread to handle some events at any time

# Distributed Approaches

- Both approaches studied thus far are centralized
- Decentralized algorithms: use resync intervals
  - Broadcast time at the start of the interval
  - Collect all other broadcast that arrive in a period *S*
  - Use average value of all reported times
  - Can throw away few highest and lowest values
- Approaches in use today
  - *rdate*: synchronizes a machine with a specified machine
  - Network Time Protocol (NTP)
    - Uses advanced techniques for accuracies of 1-50 ms

# Logical Clocks

- For many problems, internal consistency of clocks is important
  - Absolute time is less important
  - Use *logical* clocks
- Key idea:
  - Clock synchronization need not be absolute
  - If two machines do not interact, no need to synchronize them
  - More importantly, processes need to agree on the *order* in which events occur rather than the *time* at which they occurred

# Event Ordering

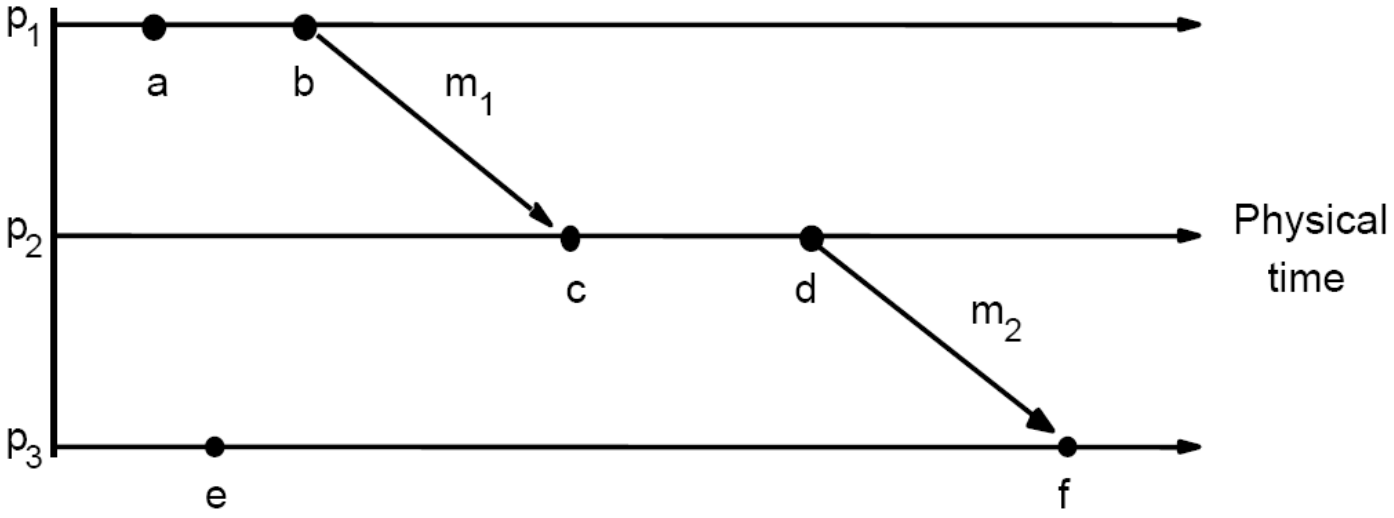
- *Problem:* define a total ordering of all events that occur in a system
- Events in a single processor machine are totally ordered
- In a distributed system:
  - No global clock, local clocks may be unsynchronized
  - Can not order events on different machines using local times
- Key idea [Lamport ]
  - Processes exchange messages
  - Message must be sent before received
  - Send/receive used to order events (and synchronize clocks)

# Happened Before Relation

- If *A* and *B* are events in the same process and *A* executed before *B*, then *A -> B*
- If A represents sending of a message and B is the receipt of this message, then A -> B
- Relation is transitive:
  - A -> B and B -> C => A -> C
- Relation is undefined across processes that do not exhange messages
  - Partial ordering on events

# HB Relation Example

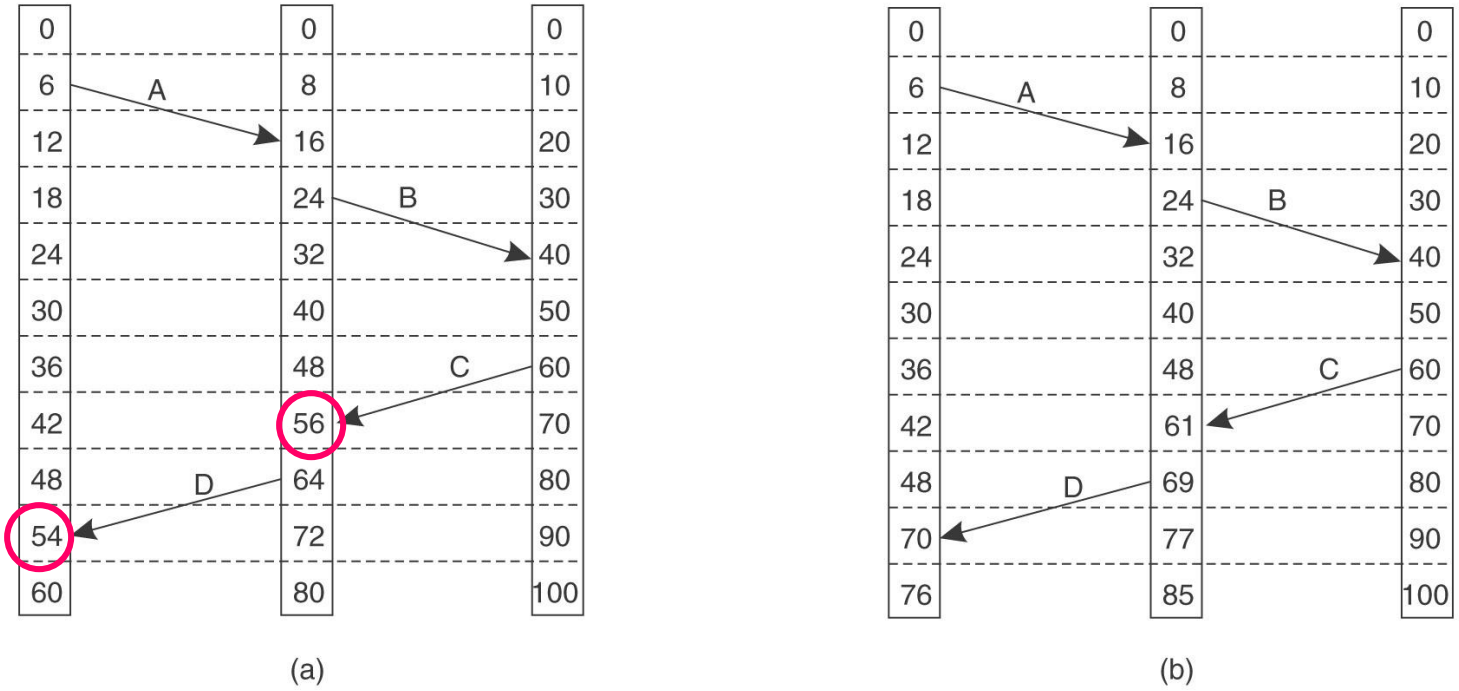
- $a \rightarrow b$
- $c \rightarrow d$
- $b \rightarrow c$
- $d \rightarrow f$
- …
- $a \rightarrow f$
- $a \not\rightarrow e$
- $e \not\rightarrow a$
- $e \| a$ (Concurrent)
  - No chain of messages
- No requirement for causation



# Event Ordering Using *HB*

- Goal: define the notion of time of an event such that
  - If A-> B then C(A) < C(B)
  - If A and B are concurrent, then C(A) <, = or > C(B)
- Solution:
  - Each processor maintains a logical clock $LC_i$
  - Whenever an event occurs locally at *i,* $LC_i = LC_i+1$
  - When *i* sends message to *j,* piggyback $LC_i$
  - When *j* receives message from *i*
    - If $LC_j < LC_i$ then $LC_j = LC_i + 1$ else do nothing
  - Claim: this algorithm meets the above goals

# Lamport's Logical Clocks




a) Each processes with own clock with different rates.
b) Lamport's algorithm corrects the clocks.

# Causality

- Lamport's logical clocks
  - If *A -> B* then *C(A) < C(B)*
  - Reverse is not true!!
    - Nothing can be said about events by comparing time-stamps!
    - If *C(A) < C(B)*, then ??
- Need to maintain *causality*
  - Causal delivery:If send(m) -> send(n) => deliver(m) -> deliver(n)
  - Capture causal relationships between groups of processes
  - Need a time-stamping mechanism such that:
    - If *T(A) < T(B)* then *A* should have causally preceded *B*

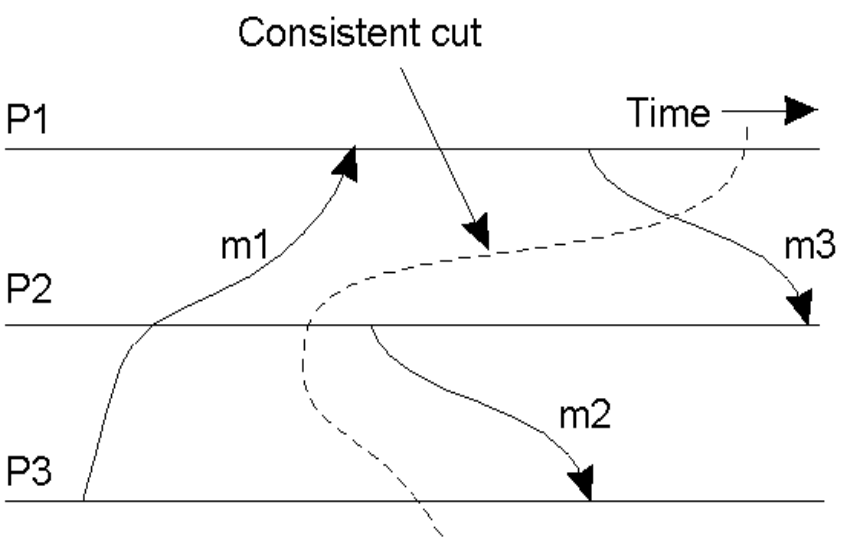
# Vector Clocks

- Each process *i* maintains a vector $V_i$
  - $V_i[i]$ : number of events that have occurred at i
  - $V_i[j]$ : number of events i knows have occurred at process j
- Update vector clocks as follows
  - Local event: increment $V_i[i]$
  - Send a message :piggyback entire vector V
  - Receipt of a message: $V_j[k] = \max( V_j[k], V_i[k] )$
    - Receiver is told about how many events the sender knows occurred at another process *k*
    - Also $V_j[i] = V_j[i]+1$
- *Homework:* convince yourself that if $V(A)<V(B)$, then *A* causally precedes *B*

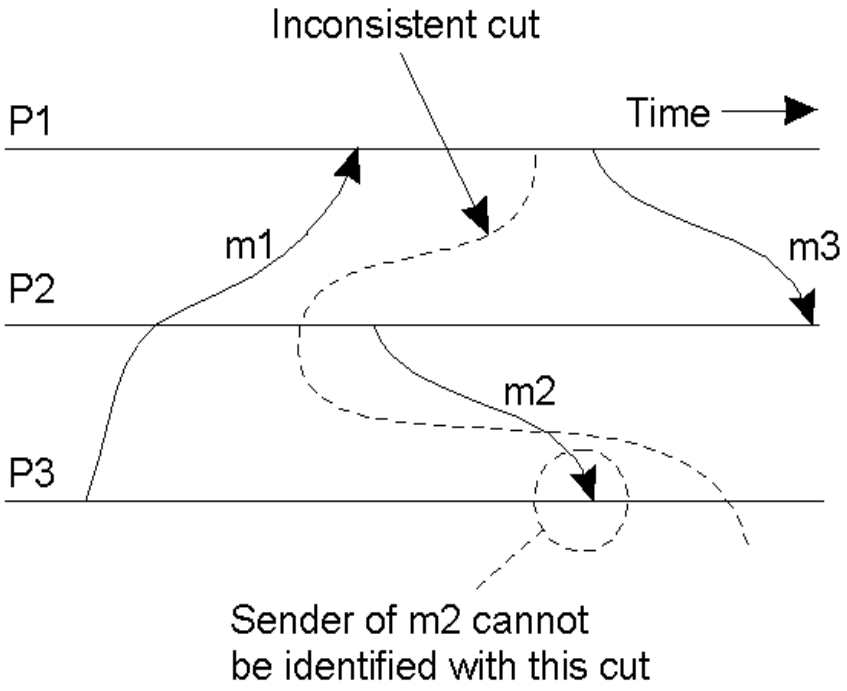
# Global State

- Global state of a distributed system
  - Local state of each process
  - Messages sent but not received (state of the queues)
- Many applications need to know the state of the system
  - Failure recovery, distributed deadlock detection
- Problem: how can you figure out the state of a distributed system?
  - Each process is independent
  - No global clock or synchronization
- Distributed snapshot: a consistent global state

# Global State (1)




(a)




(b)

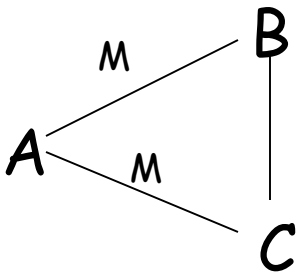
a) A consistent cut
b) An inconsistent cut

# Distributed Snapshot Algorithm

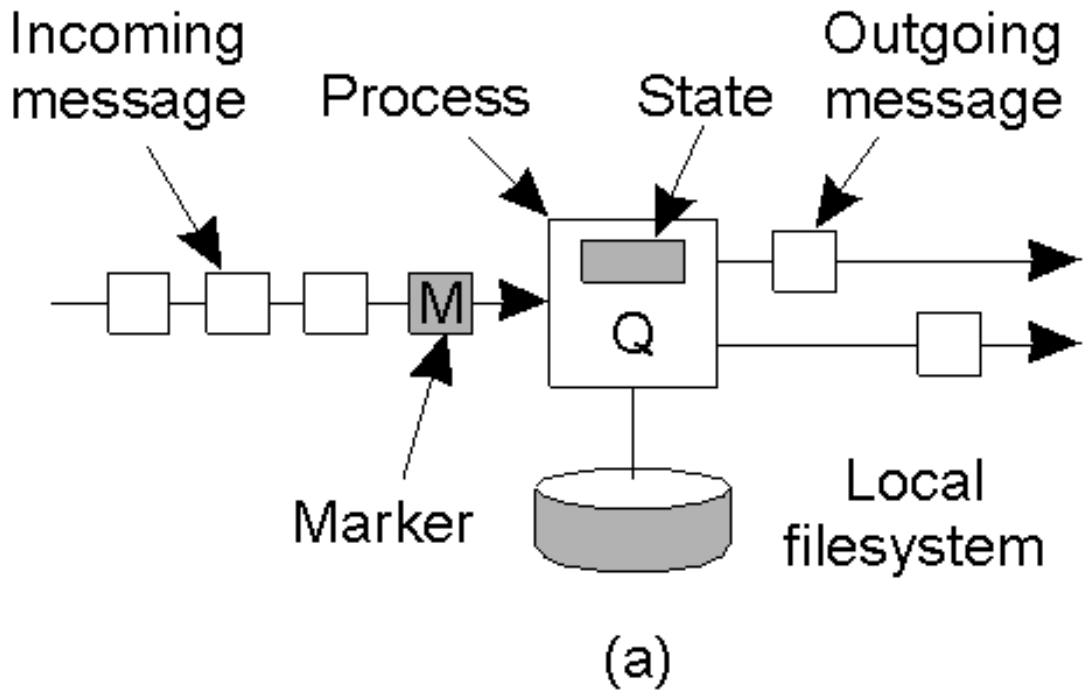
- Assume each process communicates with another process using unidirectional point-to-point channels (e.g, TCP connections)
- Any process can initiate the algorithm
  - Checkpoint local state
  - Send marker on every outgoing channel
- On receiving a marker
  - Checkpoint state if first marker and send marker on outgoing channels, stop sending , save messages on all other channels until:
  - Marker on a channel: stop saving state for that channel

# Distributed Snapshot

- A process finishes when
  - It receives a marker on each incoming channel and processes them all
  - State: local state plus state of all channels
  - Send state to initiator
- Any process can initiate snapshot
  - Multiple snapshots may be in progress
    - Each is separate, and each is distinguished by tagging the marker with the initiator ID (and sequence number)

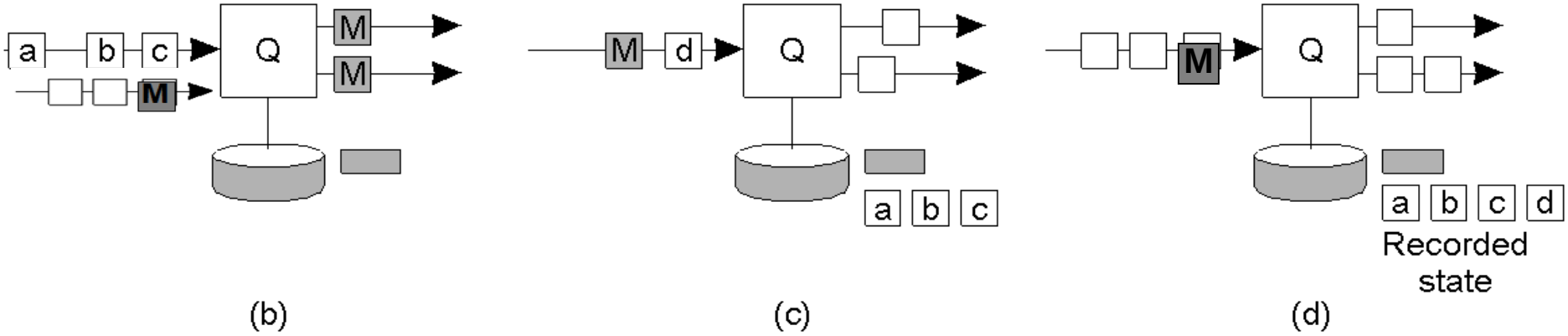


# Snapshot Algorithm Example




a) Organization of a process and channels for a distributed snapshot

# Snapshot Algorithm Example




b) Process Q receives a marker for the first time and records its local state

c) Q records all incoming message

*d)* *Q* receives a marker for its incoming channel and finishes recording the state of the incoming channel

# Termination Detection

- Detecting the end of a distributed computation
- Notation: let sender be *predecessor*, receiver be *successor*
- Two types of markers: Done and Continue
- After finishing its part of the snapshot, process *Q* sends a Done or a Continue to its predecessor
- Send a Done only when
  - All of *Q*'s successors send a Done
  - *Q* has not received any message since it check-pointed its local state and received a marker on all incoming channels

  Else send a Continue
- Computation has terminated if the initiator receives Done messages from everyone

# Election Algorithms

- Many distributed algorithms need one process to act as coordinator
  - Doesn't matter which process does the job, just need to pick one
- Election algorithms: technique to pick a unique coordinator (ex. *leader election*)
- Examples: take over the role of a failed process, pick a master in Berkeley clock synchronization algorithm
- Types of election algorithms: Bully and Ring algorithms

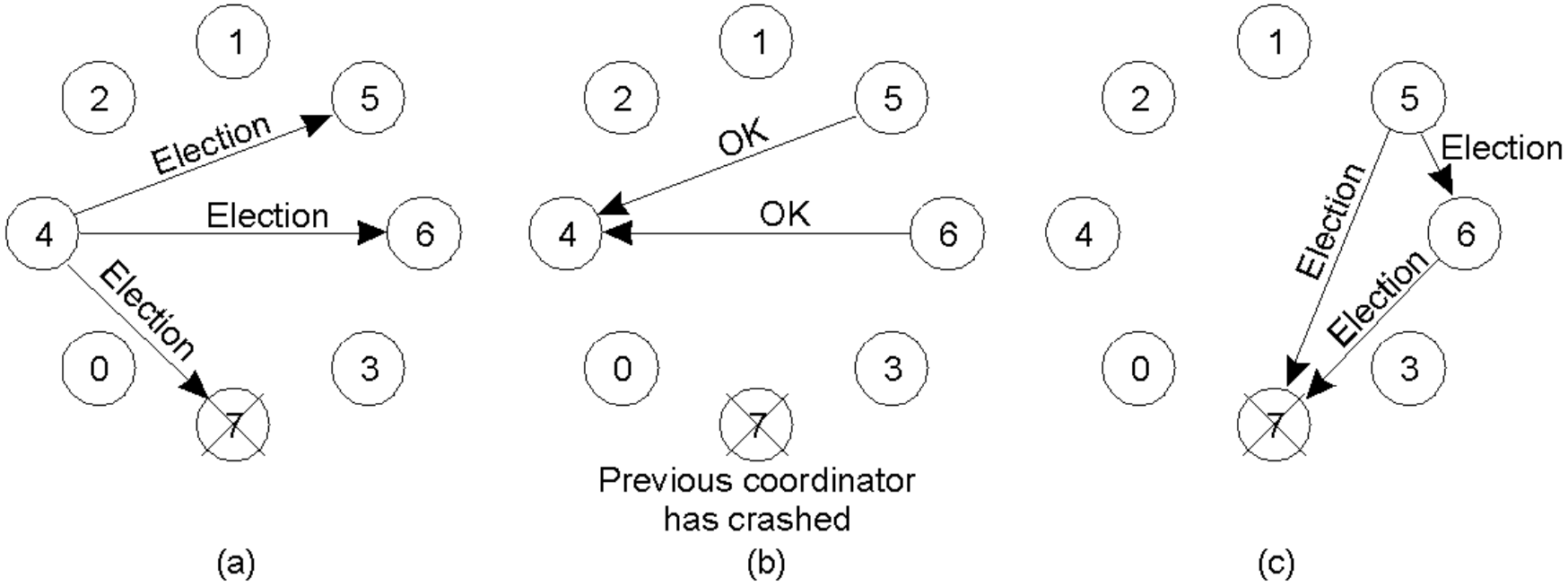
# Bully Algorithm

- Each process has a unique numerical ID
- Processes know the Ids and address of every other process
- Communication is assumed reliable
- *Key Idea*: select process with highest ID
- Process initiates election if it just recovered from failure or if coordinator failed
- 3 message types: *election, OK, I won*
- Several processes can initiate an election simultaneously
  - Need consistent result
- $O(n^2)$ messages required with $n$ processes

# Bully Algorithm Details

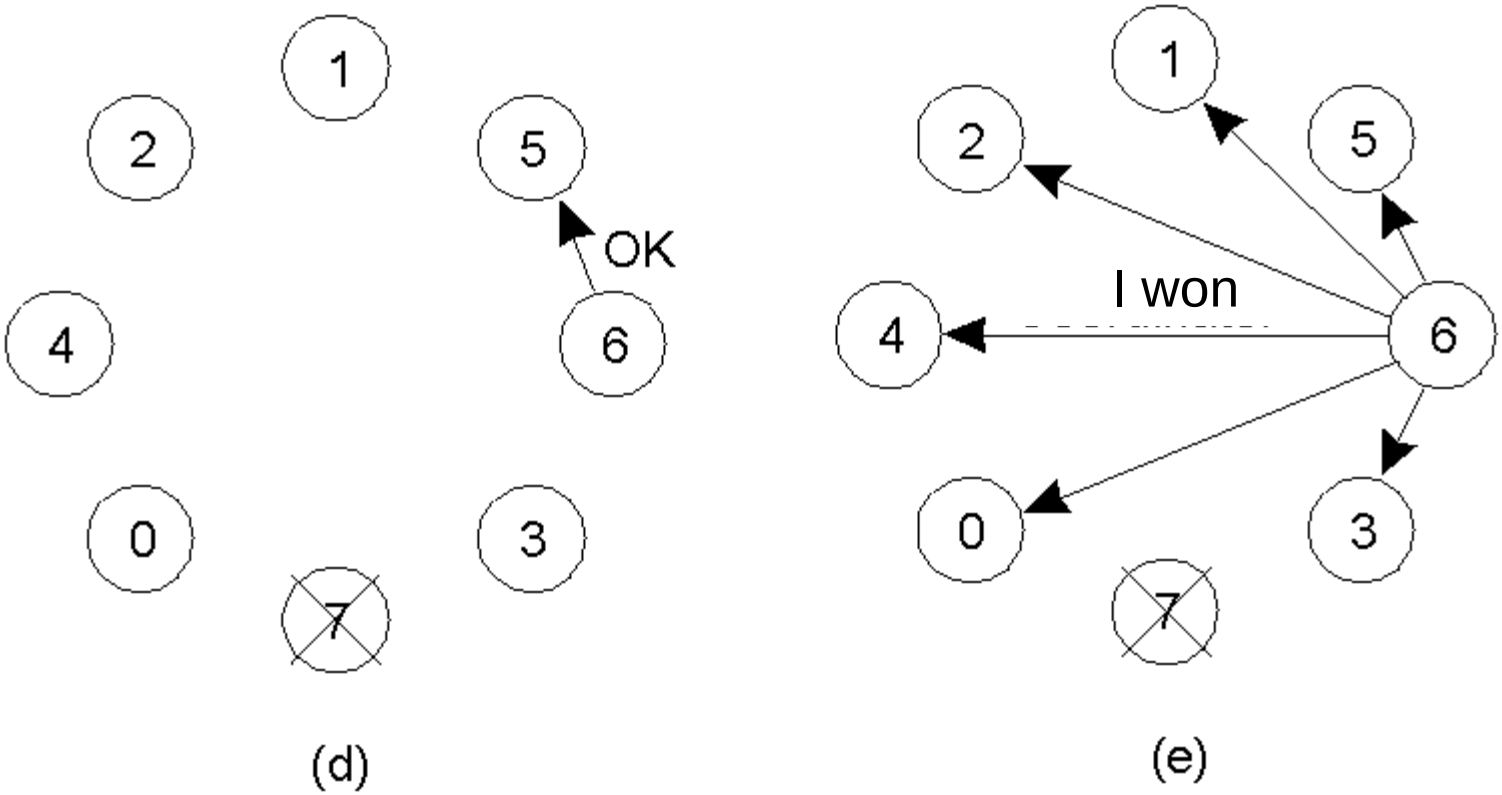
- Any process *P* can initiate an election
- *P* sends *Election* messages to all process with higher Ids and awaits *OK* messages
- If no *OK* messages, *P* becomes coordinator and sends *I won* messages to all process with lower Ids
- If it receives an *OK*, it drops out and waits for an *I won*
- If a process receives an *Election* msg, it returns an *OK* and starts an election
- If a process receives a *I won*, it treats sender an coordinator

# Bully Algorithm Example




- The bully election algorithm
- Process 4 holds an election
- Process 5 and 6 respond, telling 4 to stop
- Now 5 and 6 each hold an election

# Bully Algorithm Example



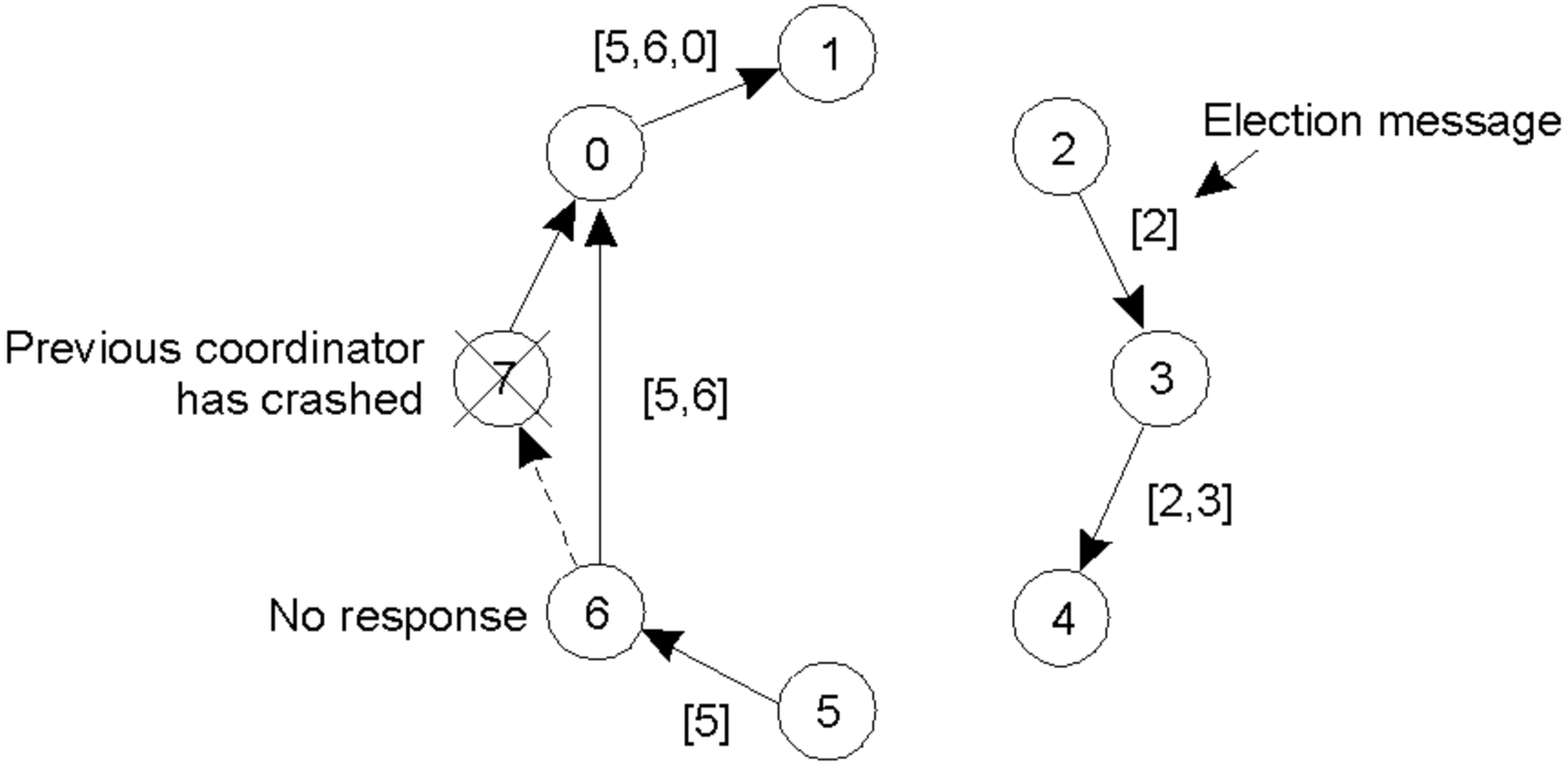

d) Process 6 tells 5 to stop

e) Process 6 wins and tells everyone

# Ring-based Election

- Processes have unique Ids and arranged in a logical ring
- Each process knows its neighbors
  - Select process with highest ID
- Begin election if just recovered or coordinator has failed
- Send *Election* to closest downstream node that is alive
  - Sequentially poll each successor until a live node is found
- Each process tags its ID on the message
- Initiator picks node with highest ID and sends a coordinator message
- Multiple elections can be in progress
  - Wastes network bandwidth but does no harm

# A Ring Algorithm



# Comparison

- Assume *n* processes and one election in progress

- Bully algorithm
  - Worst case: initiator is node with lowest ID
    - Triggers n-1 elections at higher ranked nodes: $O(n^2)$ msgs
  - Best case: immediate election: n-1 messages
- Ring
  - 2 (n-1) messages always

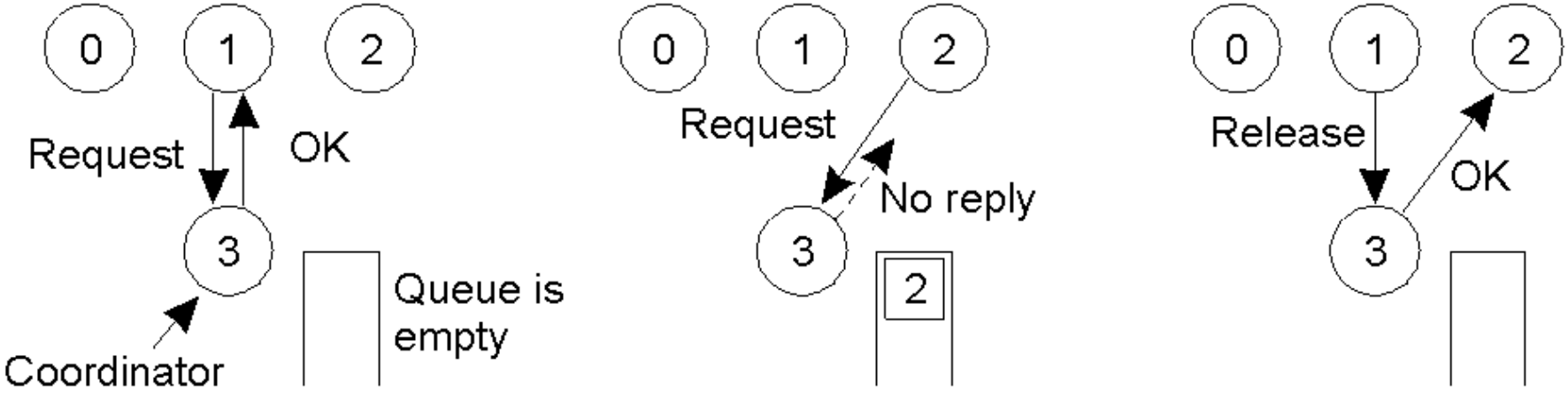
# Distributed Synchronization

- Distributed system with multiple processes may need to share data or access shared data structures
  - Use critical sections with mutual exclusion
- Single process with multiple threads
  - Semaphores, locks, monitors
- How do you do this for multiple processes in a distributed system?
  - Processes may be running on different machines
- Solution: lock mechanism for a distributed environment
  - Can be centralized or distributed

# Centralized Mutual Exclusion

- Assume processes are numbered
- One process is elected coordinator (highest ID process)
- Every process needs to check with coordinator before entering the critical section
- To obtain exclusive access: send request, await reply (grant OK)
- To release: send release message
- Coordinator:
  - Receive *request*: if available and queue empty, send grant; if not, queue request
  - Receive *release*: remove next request from queue and send grant OK

# Mutual Exclusion: A Centralized Algorithm




(a) (b) (c)

a) Process 1 asks the coordinator for permission to enter a critical region. Permission is granted

b) Process 2 then asks permission to enter the same critical region. The coordinator does not reply.

c) When process 1 exits the critical region, it tells the coordinator, when then replies to 2

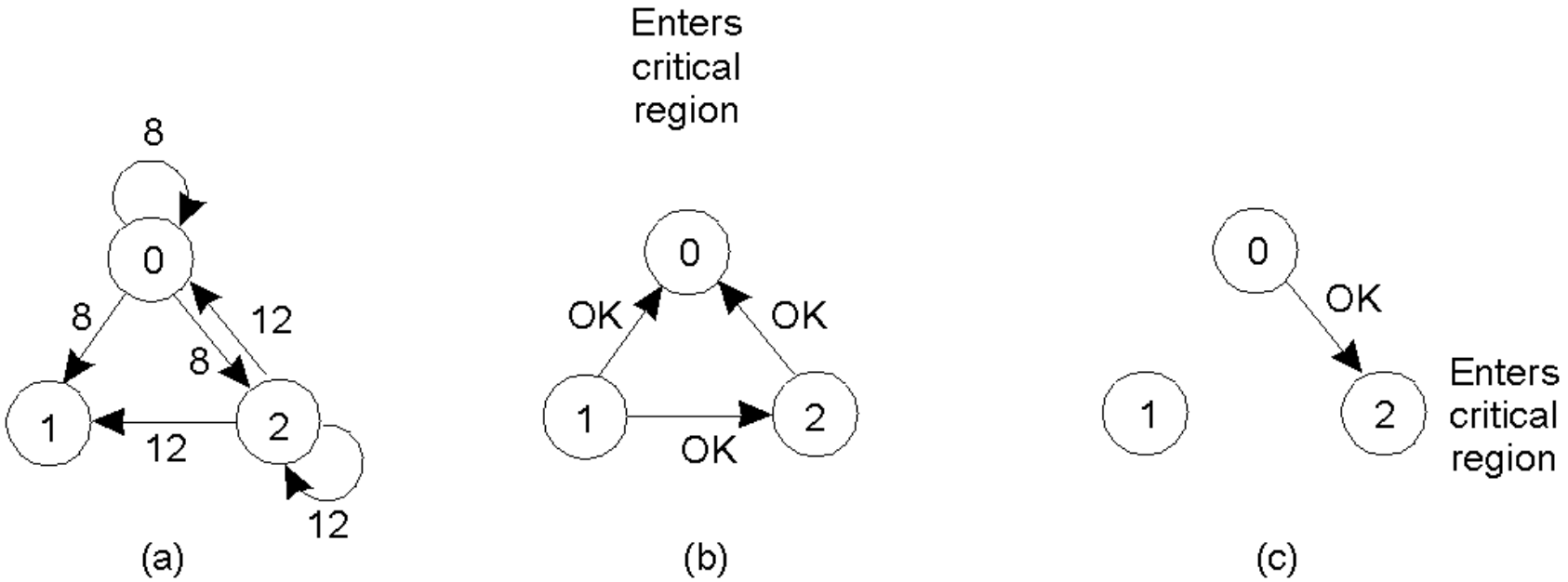
# Properties

- Simulates centralized lock using blocking calls
- Fair: requests are granted the lock in the order they were received
- Simple: three messages per use of a critical section (request, grant, release)
- Shortcomings:
  - Single point of failure
  - How do you detect a dead coordinator?
    - A process can not distinguish between “lock in use” from a dead coordinator
      - No response from coordinator in either case
  - Performance bottleneck in large distributed systems

# Distributed Algorithm

- [Ricart and Agrawala]: needs 2(n-1) messages
- Based on event ordering and time stamps
- Process *k* enters critical section as follows
  - Generate new time stamp $TS_k = TS_k+1$
  - Send *request(k,*$TS_k$*)* all other *n-1* processes
  - Wait until *reply(j)* received from all other processes
  - Enter critical section
- Upon receiving a *request* message, process *j*
  - Sends *reply OK* if no contention
  - If already in critical section, does not reply, queue request
  - If wants to enter, compare $TS_j$ with $TS_k$ and send reply if $TS_k<TS_j$, else queue

# A Distributed Algorithm




a) Two processes want to enter the same critical region at the same moment.

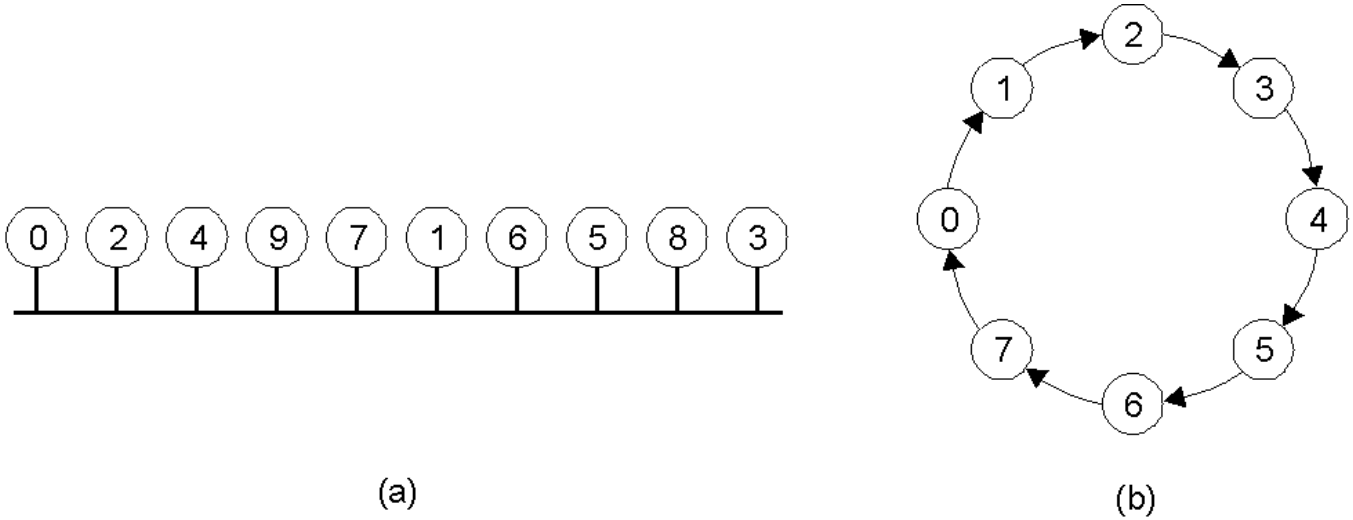
b) Process 0 has the lowest timestamp, so it wins.

c) When process 0 is done, it sends an OK also, so 2 can now enter the critical region.

# Properties

- Fully decentralized
- *N* points of failure!
- All processes are involved in all decisions
  - Any overloaded process can become a bottleneck

# A Token Ring Algorithm




a) An unordered group of processes on a network.

b) A logical ring constructed in software.

- Use a token to arbitrate access to critical section
- Must wait for token before entering CS
- Pass the token to neighbor once done or if not interested
- Detecting token loss in not-trivial

# Comparison

| Algorithm | Messages per entry/exit | Delay before entry (in message times) | Problems |
|---|---|---|---|
| Centralized | 3 | 2 | Coordinator crash |
| Distributed | 2 ( n – 1 ) | 2 ( n – 1 ) | Crash of any process |
| Token ring | 1 to ∞ | 0 to n – 1 | Lost token, process crash |

- A comparison of three mutual exclusion algorithms.

# Consistency and Replication

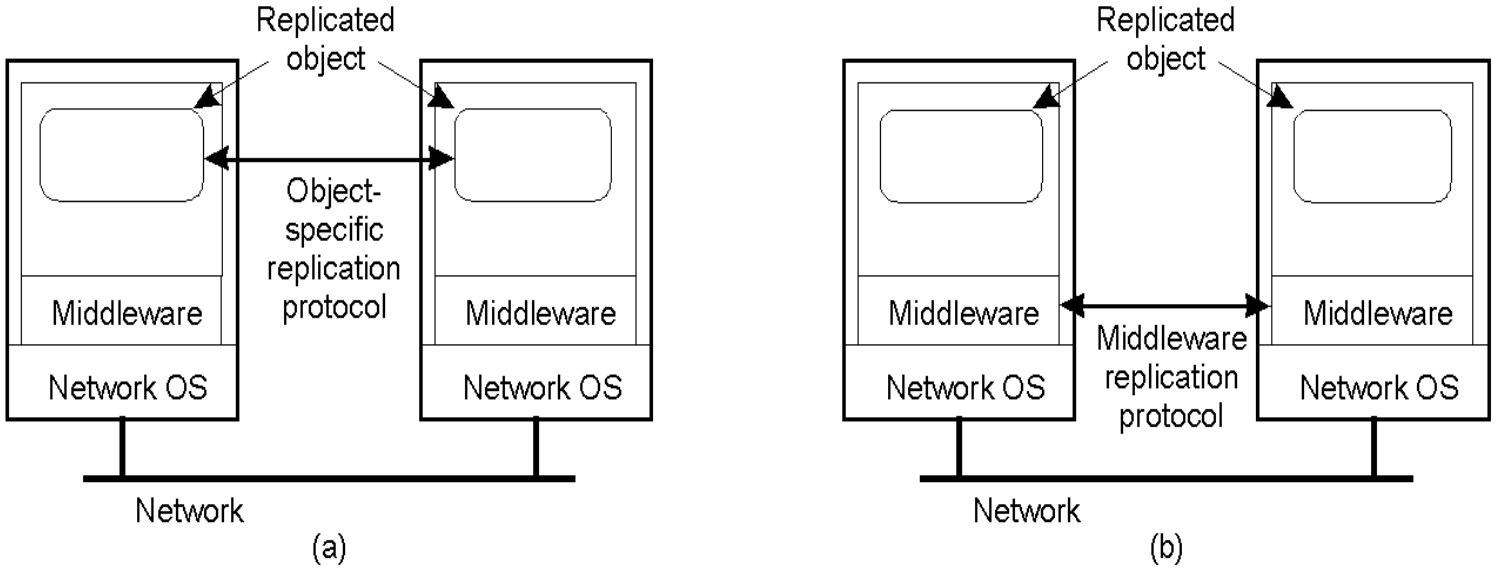
- Today:
  - Introduction
  - Consistency models
    - Data-centric consistency models
    - Client-centric consistency models
  - Thoughts for the mid-term

# Why replicate?

- Data replication: common technique in distributed systems
- Reliability
  - If one replica is unavailable or crashes, use another
  - Protect against corrupted data
- Performance
  - Scale with size of the distributed system (replicated web servers)
  - Scale in geographically distributed systems (web proxies)

- Key issue: need to maintain *consistency* of replicated data
  - If one copy is modified, others become inconsistent

# Object Replication

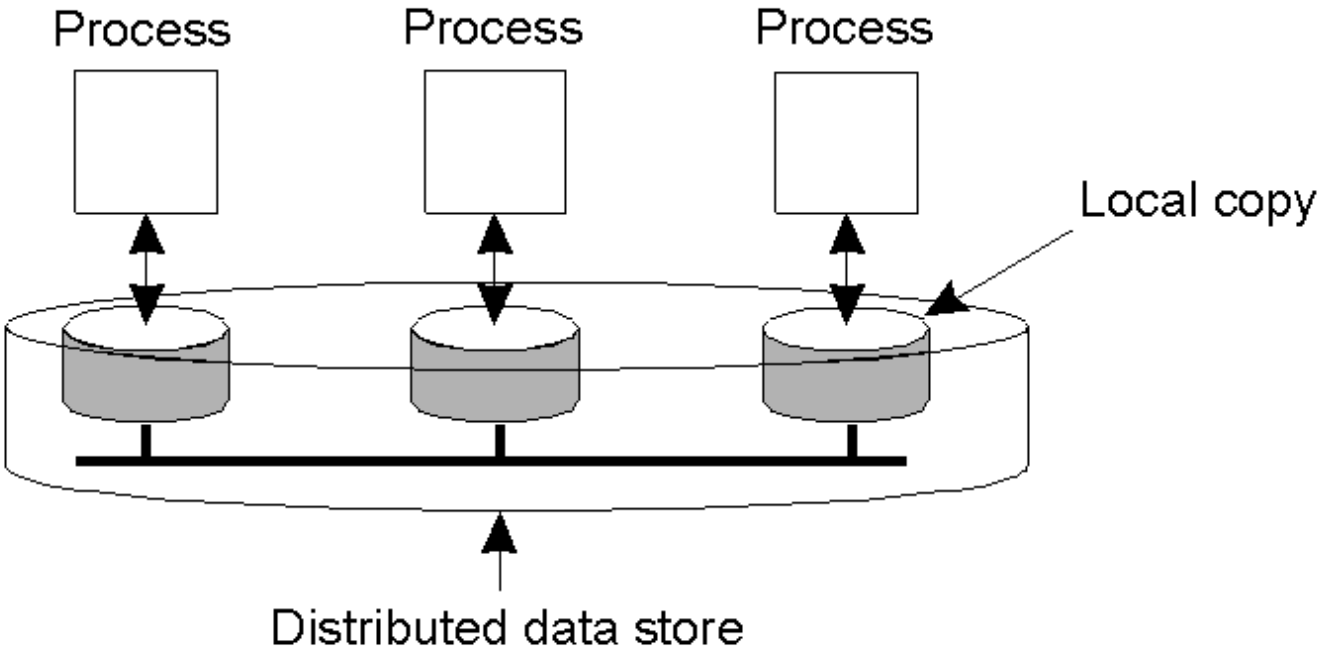



- Approach 1: application is responsible for replication
  - Application needs to handle consistency issues
- Approach 2: system (middleware) handles replication
  - Consistency issues are handled by the middleware
  - Simplifies application development but makes object-specific solutions harder

# Replication and Scaling

- Replication and caching used for system scalability
- Multiple copies:
  - Improves performance by reducing access latency
  - But higher network overheads of maintaining consistency
  - Example: object is replicated *N* times
    - Read frequency *R*, write frequency *W*
    - If *R*<<*W*, high consistency overhead and wasted messages
    - Consistency maintenance is itself an issue
      - What semantics to provide?
      - Tight consistency requires globally synchronized clocks!
- Solution: loosen consistency requirements
  - Variety of consistency semantics possible

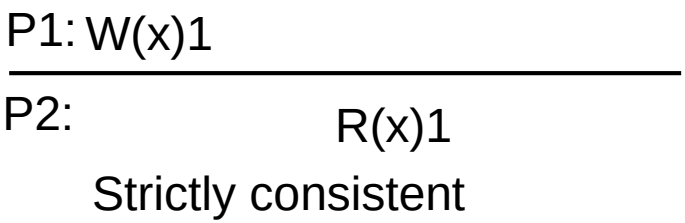
# Data-Centric Consistency Models



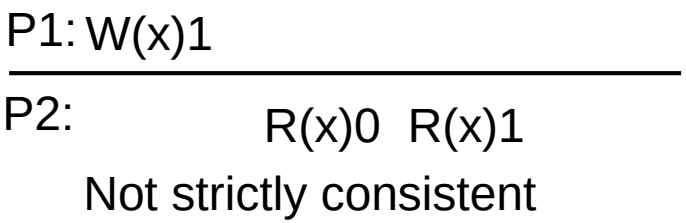

- Consistency model (aka *consistency semantics*)
  - Contract between processes and the data store
    - If processes obey certain rules, data store will work correctly
  - All models attempt to return the results of the last write for a read operation
    - Differ in how “last” write is determined/defined

# Strict Consistency

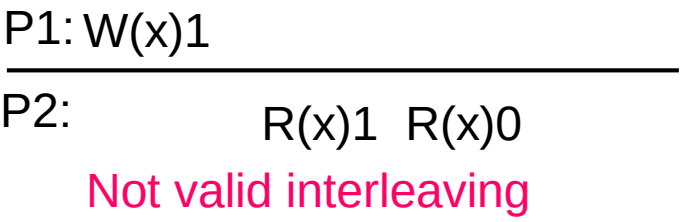
- Any read always returns the result of the most recent write
  - Implicitly assumes the presence of a global clock
  - A write is immediately visible to all processes
    - Difficult to achieve in real systems (network delays can be variable)




Strictly consistent



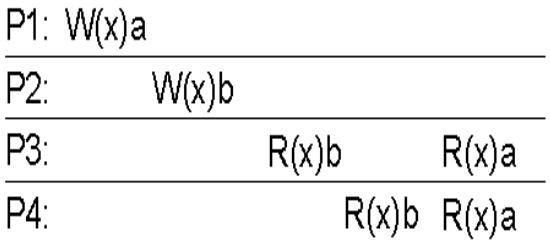

Not strictly consistent



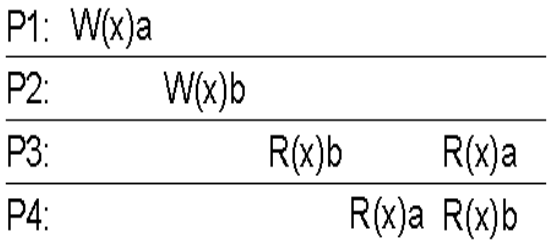

Not valid interleaving

# Sequential Consistency

- Sequential consistency: weaker than strict consistency
  - Assumes all operations are executed in some sequential order and each process issues operations in program order
    - Any valid interleaving is allowed
    - All agree on the same interleaving
    - Each process preserves its program order
    - Nothing is said about “most recent write”




(a) Permitted




(b) Not permitted

All processes must see the same sequence of memory references

# Linearizability

- Assumes sequential consistency *and*
  - If TS(x) < TS(y) then OP(x) should precede OP(y) in the sequence
  - Stronger than sequential consistency
  - Difference between linearizability and serializbility?
    - Granularity: reads/writes versus transactions
- Example:

| Process P1 | Process P2 | Process P3 |
|---|---|---|
| x = 1; | y = 1; | z = 1; |
| print ( y, z); | print (x, z); | print (x, y); |

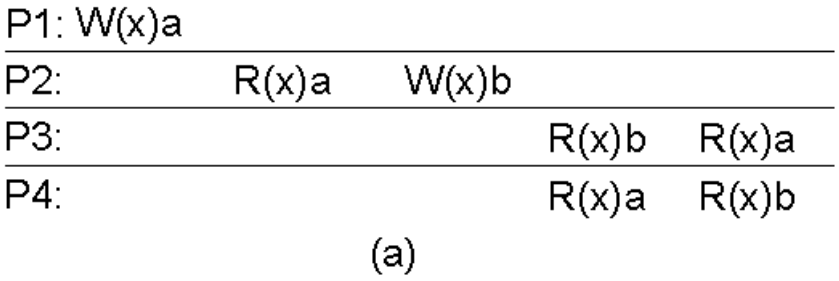
# Linearizability Example

- Four valid execution sequences for the processes of the previous slide. The vertical axis is time.

| | | | |
|---|---|---|---|
| x = 1; | x = 1; | y = 1; | y = 1; |
| print ((y, z); | y = 1; | z = 1; | x = 1; |
| y = 1; | print (x,z); | print (x, y); | z = 1; |
| print (x, z); | print(y, z); | print (x, z); | print (x, z); |
| z = 1; | z = 1; | x = 1; | print (y, z); |
| print (x, y); | print (x, y); | print (y, z); | print (x, y); |
| Prints: 001011 | Prints: 101011 | Prints: 010111 | Prints: 111111 |
| Signature: 001011 (a) | Signature: 101011 (b) | Signature: 110101 (c) | Signature: 111111 (d) |

# Causal consistency

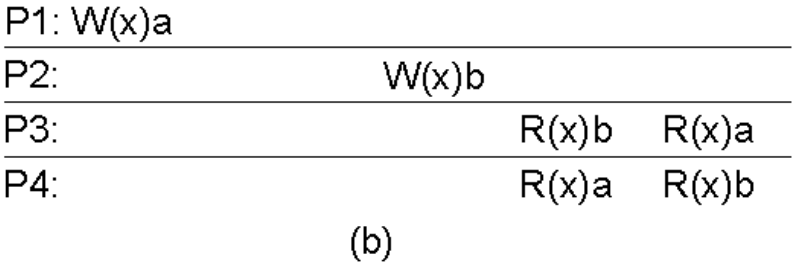
- Causally related writes must be seen by all processes in the same order.
  - Concurrent writes may be seen in different orders on different machines




(a)

Not permitted

W(x)a and W(x)b in causal relation




(b)

Permitted

W(x)a and W(x)b are concurrent

# Other models

- FIFO consistency: writes from a process are seen by others in the same order. Writes from different processes may be seen in different order (even if causally related)
  - Relaxes causal consistency
  - Simple implementation: tag each write by (Proc ID, seq #)
- Even FIFO consistency may be too strong!
  - Requires all writes from a process be seen in order
- Assume use of critical sections for updates
  - Send final result of critical section everywhere
  - Do not worry about propagating intermediate results
    - Assume presence of synchronization primitives to define semantics

# Other Models

- Weak consistency
  - Accesses to synchronization variables associated with a data store are sequentially consistent
  - No operation on a synchronization variable is allowed to be performed until all previous writes have been completed everywhere
  - No read or write operation on data items are allowed to be performed until all previous operations to synchronization variables have been performed.
- Entry and release consistency
  - Assume shared data are made consistent at entry or exit points of critical sections

# Summary of Data-centric Consistency Models

| Consistency | Description |
|---|---|
| Strict | Absolute time ordering of all shared accesses matters. |
| Linearizability | All processes must see all shared accesses in the same order. Accesses are furthermore ordered according to a (nonunique) global timestamp |
| Sequential | All processes see all shared accesses in the same order. Accesses are not ordered in time |
| Causal | All processes see causally-related shared accesses in the same order. |
| FIFO | All processes see writes from each other in the order they were used. Writes from different processes may not always be seen in that order |

(a) Not using synchronization operations

| Consistency | Description |
|---|---|
| Weak | Shared data can be counted on to be consistent only after a synchronization is done |
| Release | Shared data are made consistent when a critical region is exited |
| Entry | Shared data pertaining to a critical region are made consistent when a critical region is entered. |

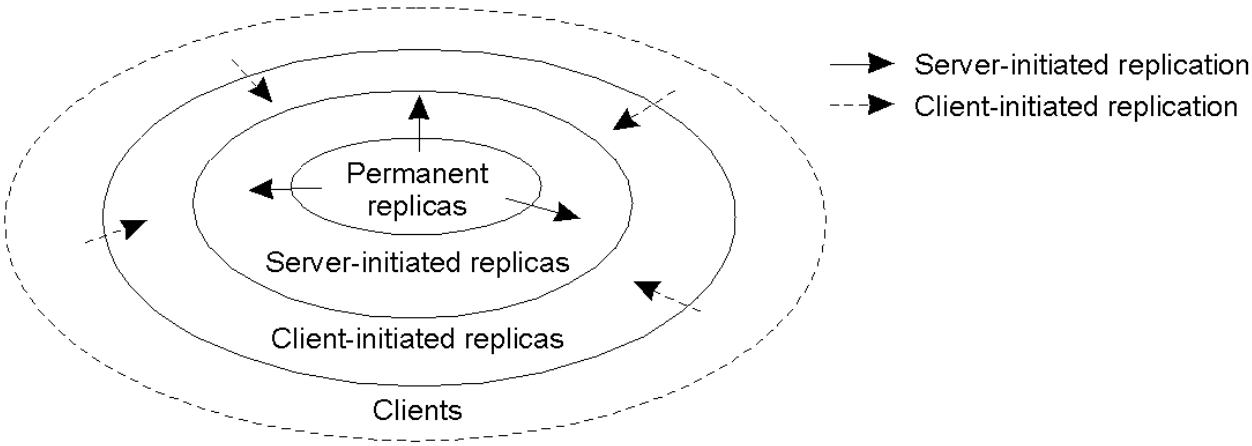
(b) Using synchronization operations

# Implementation Issues

- Replica placement
- Use web caching as an illustrative example
- Distribution protocols
  - Invalidate versus updates
  - Push versus Pull
  - Cooperation between replicas

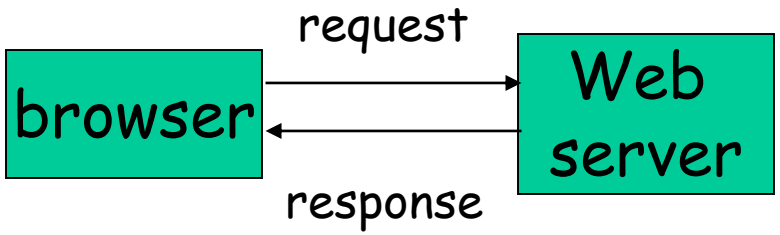
# Replica Placement

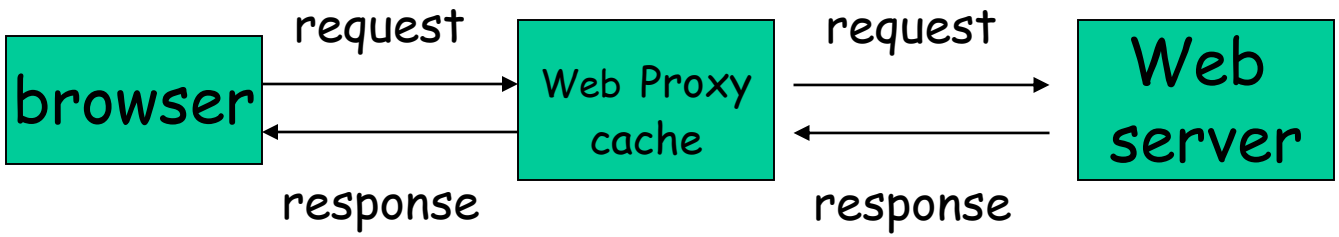
- Permanent replicas (mirroring)
- Server-initiated replicas (push caching)
- Client-initiated replicas (pull/client caching)



# Web Caching

- Example of the web to illustrate caching and replication issues
  - Simpler model: clients are read-only, only server updates data



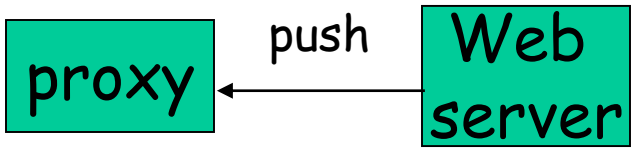



# Consistency Issues

- Web pages tend to be updated over time
  - Some objects are static, others are dynamic
  - Different update frequencies (few minutes to few weeks)
- How can a proxy cache maintain consistency of cached data?
  - Send invalidate or update
  - Push versus pull

# Push-based Approach

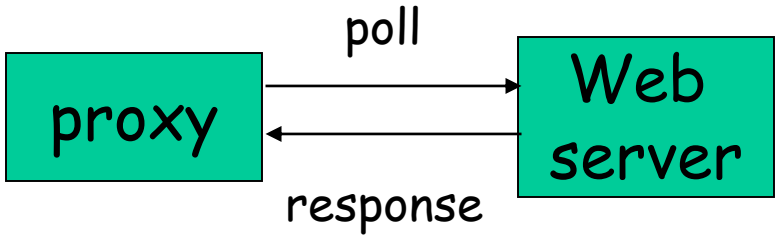
- Server tracks all proxies that have requested objects
- If a web page is modified, notify each proxy
- Notification types
  - Indicate object has changed [invalidate]
  - Send new version of object [update]
- How to decide between invalidate and updates?
  - Pros and cons?
  - One approach: send updates for more frequent objects, invalidate for rest



# Push-based Approaches

- Advantages
  - Provide tight consistency [minimal stale data]
  - Proxies can be passive
- Disadvantages
  - Need to maintain state at the server
    - Recall that HTTP is stateless
    - Need mechanisms beyond HTTP
  - State may need to be maintained indefinitely
    - Not resilient to server crashes

# Pull-based Approaches




- Proxy is entirely responsible for maintaining consistency
- Proxy periodically polls the server to see if object has changed
  - Use if-modified-since HTTP messages
- Key question: when should a proxy poll?
  - Server-assigned *Time-to-Live (TTL)* values
    - No guarantee if the object will change in the interim

# Pull-based Approach: Intelligent Polling

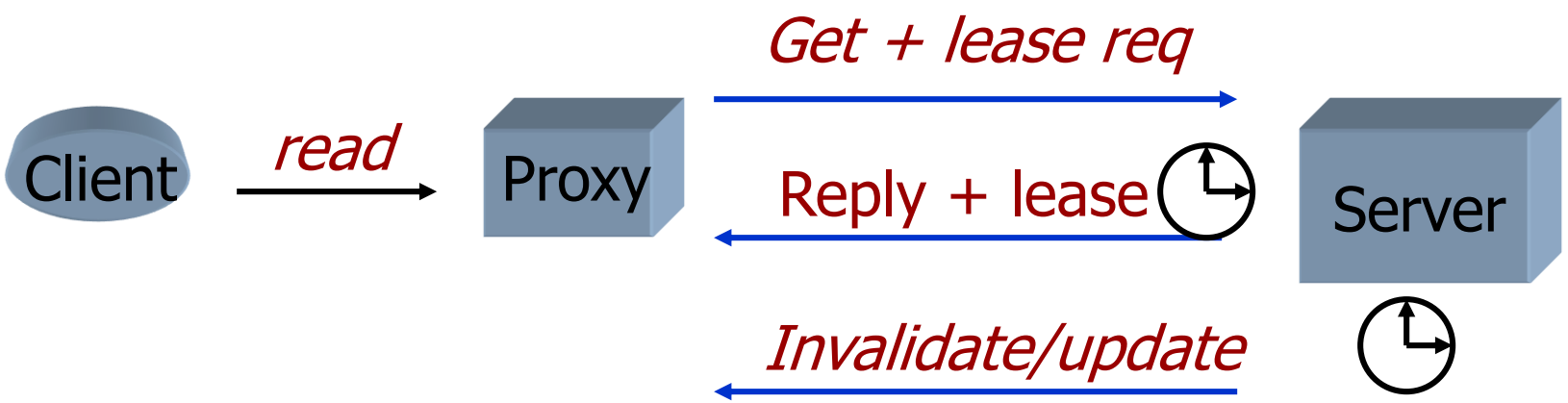
- Proxy can dynamically determine the refresh interval
  - Compute based on past observations
    - Start with a conservative refresh interval
    - Increase interval if object has not changed between two successive polls
    - Decrease interval if object is updated between two polls
    - Adaptive: No prior knowledge of object characteristics needed

# Pull-based Approach

- Advantages
  - Implementation using HTTP (If-modified-Since)
  - Server remains stateless
  - Resilient to both server and proxy failures
- Disadvantages
  - Weaker consistency guarantees (objects can change between two polls and proxy will contain stale data until next poll)
    - Strong consistency only if poll before every HTTP response
  - More sophisticated proxies required
  - High message overhead

# A Hybrid Approach: Leases

- Lease: duration of time for which server agrees to notify proxy of modification
- Issue lease on first request, send notification until expiry
  - Need to renew lease upon expiry
- Smooth tradeoff between state and messages exchanged
  - Zero duration => polling, Infinite leases => server-push
- Efficiency depends on the *lease duration*



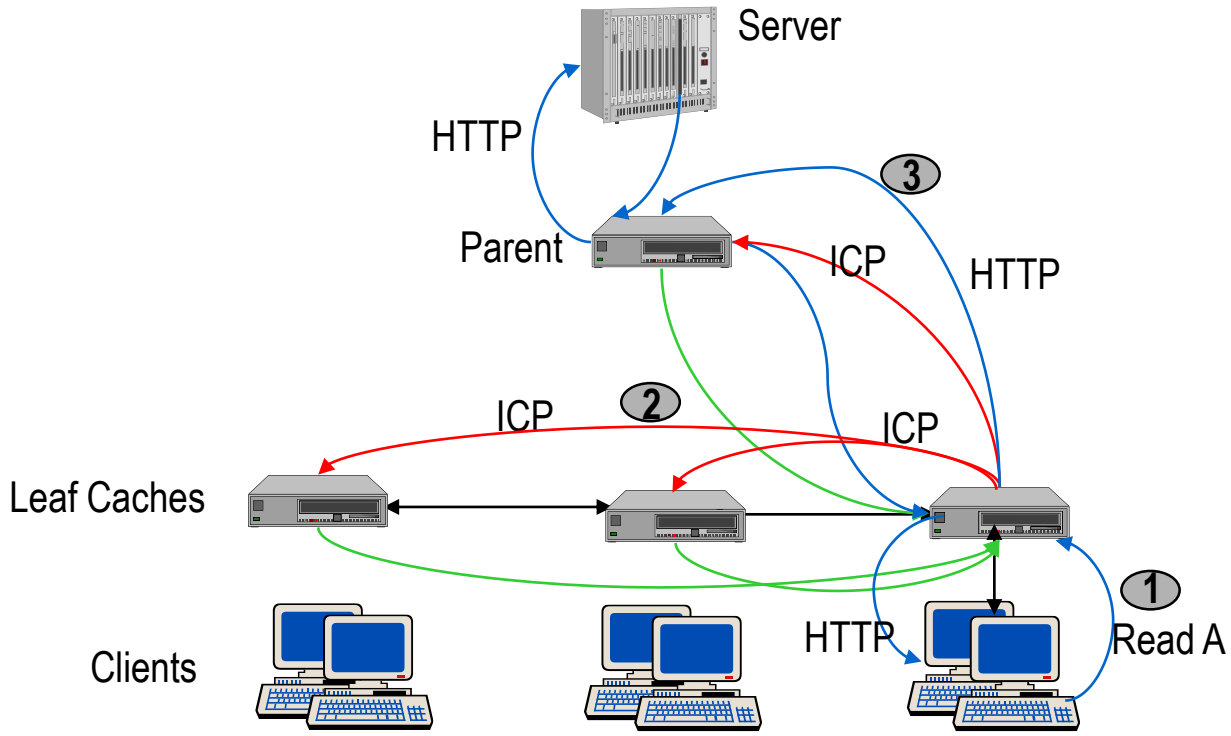
# Policies for Leases Duration

- Age-based lease
  - Based on bi-modal nature of object lifetimes
  - Larger the expected lifetime longer the lease
- Renewal-frequency based
  - Based on skewed popularity
  - Proxy at which objects is popular gets longer lease
- Server load based
  - Based on adaptively controlling  the state space
  - Shorter leases during heavy load

# Cooperative Caching

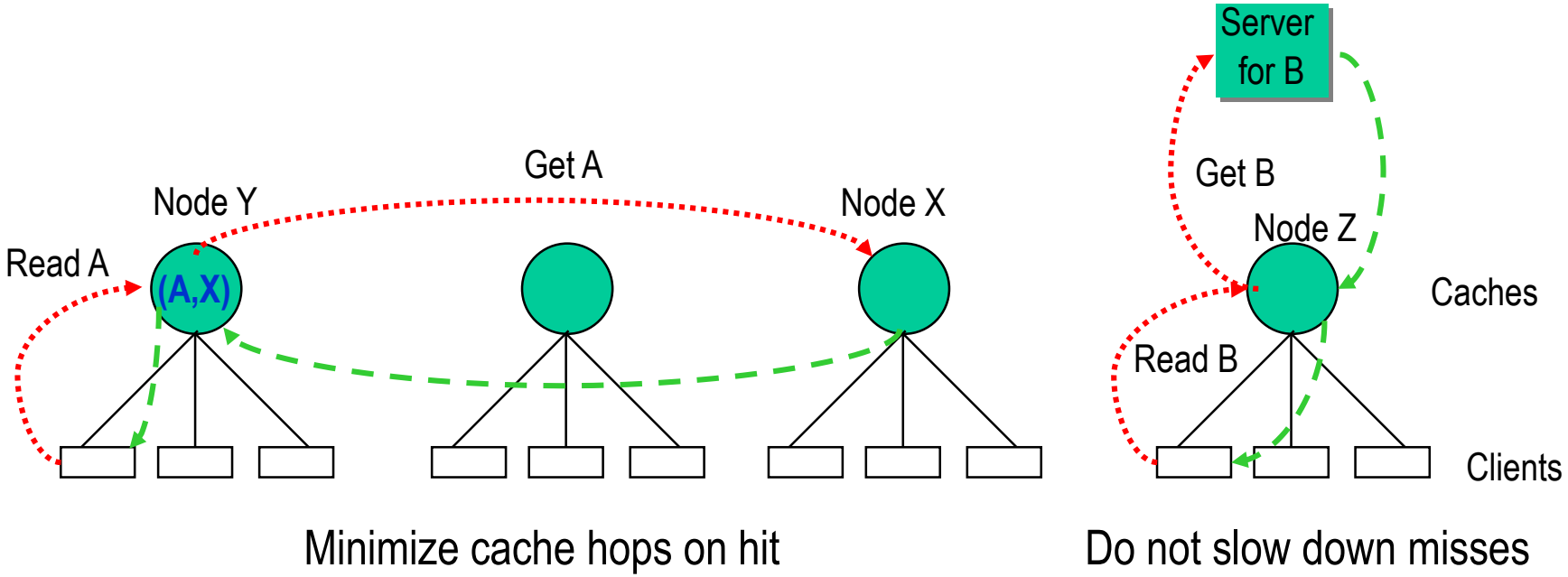
- Caching infrastructure can have multiple web proxies
  - Proxies can be arranged in a hierarchy or other structures
    - Overlay network of proxies: content distribution network
  - Proxies can cooperate with one another
    - Answer client requests
    - Propagate server notifications

# Hierarchical Proxy Caching




Examples: Squid, Harvest

# Locating and Accessing Data




## Properties

- Lookup is local
- Hit at most 2 hops
- Miss at most 2 hops (1 extra on wrong hint)

# CDN Issues

- Which proxy answers a client request?
  - Ideally the “closest” proxy
  - Akamai uses a DNS-based approach
- Propagating notifications
  - Can use multicast or application level multicast to reduce overheads (in push-based approaches)
- Active area of research
  - Numerous research papers available