

Distributed Systems

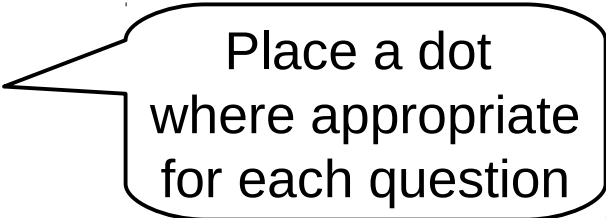
Lec 8: Distributed Mutual Exclusion

Slide acks: Dave Andersen

(<http://www.cs.cmu.edu/~dga/15-440/F10/lectures/Distributed-Mutual-Exclusion-slides.pdf>)

Congrats on finishing HW 2

- **Poll:** How hard was it intellectually / coding-wise (ignore time)?
 - a) Easy
 - b) Medium
 - c) Hard
- **Poll:** How was the time budget?
 - a) Too little time
 - b) Sufficient time, but I wish I had started earlier
 - c) Insufficient time
- **Poll:** How much did you learn?
 - a) Little
 - b) Medium
 - c) A lot



Place a dot
where appropriate
for each question

Last Time: Time & Synchronization

- Synchronizing real, distributed clocks
 - Why is it hard?
 - What are some algorithms? Describe 'em.
- Logical time
 - What is that?
 - What are its goals?
 - How do Lamport clocks work?
 - How to get global ordering for Lamport clocks?

Last Time: Time & Synchronization

- Synchronizing real, distributed clocks
 - Why is it hard? Asynchronous, unreliable networks
 - What are some algorithms?
 - Cristian's algorithm: request remote time, measure RTT, and set local time to remote time + $RTT/2$ for bounded error
 - NTP: uses modified Cristian's algo, but distributes time servers into three layers of decreased accuracy
- Logical time
 - What is that?
 - Discreet assignment of sequence numbers to events, which preserve "happens-before" orders
 - How do Lamport clocks work?
 - Processes increment their clocks upon receiving/sending new messages and based on other processes' clocks

Today: Distributed Mutual Exclusion

- We'll look at five algorithms (more like seven)
 - Centralized algorithm
 - Token algorithms
 - Distributed algorithms

Today: Distributed Mutual Exclusion

- We'll look at five algorithms (more like seven)
 - Centralized algorithm
 - Token algorithms
 - Distributed algorithms
- A word of warning:
 - None of the algorithms is perfect, all have **tradeoffs**
 - So, don't expect a natural progression to some "great" algorithm
 - **The goal is to understand several algorithms, so you get used to the idea of distributed algorithms, logical clocks, voting, etc.**

Distributed Mutual Exclusion

- Maintain mutual exclusion among **n distributed processes**
 - Terminology: use process/processor/machine/server/node to denote the processing unit in a distributed system

Distributed Mutual Exclusion

- Maintain mutual exclusion among **n distributed processes**
 - Terminology: use process/processor/machine/server/node to denote the processing unit in a distributed system
- Model: Each process executes loop of form:

```
while true:  
    Perform local operations  
    Acquire()  
    Execute critical section  
    Release()
```


Distributed Mutual Exclusion

- Maintain mutual exclusion among **n distributed processes**
 - Terminology: use process/processor/machine/server/node to denote the processing unit in a distributed system
- Model: Each process executes loop of form:

```
while true:  
    Perform local operations  
    Acquire()  
    Execute critical section  
    Release()
```

- During critical section, process interacts with remote processes or directly with shared resource
 - Example: **send a message to a shared file server asking it to write something to a file**

Goals of Distributed Mutual Exclusion

- Much like regular mutual exclusion
 - **Safety**: at most one process holds the lock at any time
 - **Liveness**: progress (if no one holds the lock, a processor requesting it will get it)
 - **Fairness**: bounded wait and in-order

Goals of Distributed Mutual Exclusion

- Much like regular mutual exclusion
 - **Safety**: at most one process holds the lock at any time
 - **Liveness**: progress (if no one holds the lock, a processor requesting it will get it)
 - **Fairness**: bounded wait and in-order



in logical time

Goals of Distributed Mutual Exclusion

- Much like regular mutual exclusion
 - **Safety**: at most one process holds the lock at any time
 - **Liveness**: progress (if no one holds the lock, a processor requesting it will get it)
 - **Fairness**: bounded wait and in-order
- Other goals:
 - Minimize **message traffic**
 - Minimize **synchronization delay**
 - Switch quickly between processes waiting for lock
 - i.e., if no one has the lock and you ask for it, you should quickly get it



in logical time

Distributed Mutual Exclusion Is Different

- Regular mutual exclusion solved using **shared state**
 - E.g., atomic test-and-set of shared variable
- We solve distributed mutual exclusion with **message passing**

Distributed Mutual Exclusion Is Different

- Regular mutual exclusion solved using **shared state**
 - E.g., atomic test-and-set of shared variable
- We solve distributed mutual exclusion with **message passing**
- **Assumptions** for this lecture:
 - The network is **reliable** (all messages sent get to their destinations at some point in time)
 - Network is **asynchronous** (messages may take long time)
 - Processes may **fail** at any time

Distributed Mutual Exclusion Protocols

- Key ideas:
 - Before entering critical section, processor must get permission from other processors
 - When exiting critical section, processor must let the others know that he's finished
 - For fairness, processors allow other processors who have asked for permission earlier than them to proceed
- We'll give examples of five such protocols (+ two variations)
 - We'll compare them from a **liveness**, **message overhead**, **synchronization delay** perspective

Solution 1: Centralized Lock Server

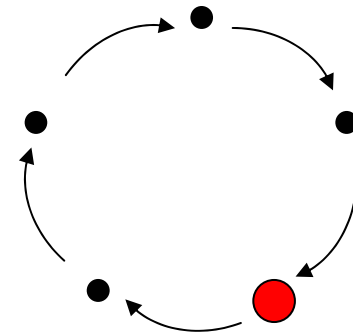
- To enter critical section:
 - send REQUEST to central server
 - wait for permission from server
- To leave critical section:
 - send RELEASE to central server
- Server:
 - Has an internal queue of all REQUESTs it's received but to which it hasn't yet sent OK
 - Delays sending OK back to process until process is at head of queue
 - Removes process from the queue after it gets RELEASE

Solution 1: Centralized Lock Server

- **Advantages:**
 - Simple (we like simple!)
 - Only 3 messages required per sync session (enter&exit)
- **Disadvantages:**
 - Single point of failure
 - Single performance bottleneck
 - With an asynchronous network, doesn't achieve in-order fairness (even for logical time order)
 - Must select (or *elect*) a central server

Solution 2: A ring-based algorithm

- Pass a token around a ring
 - Can enter critical section only if you hold the token
- Problems:
 - Not in-order
 - Long synchronization delay
 - Need to wait for up to $N-1$ messages, for N processors
 - Very unreliable
 - Any process failure breaks the ring



2': A fair ring-based algorithm

- Token contains the time t of the earliest known outstanding request
- To enter critical section:
 - Stamp your request with the current time T_r , wait for token
- When you get token with time t while waiting with request from time T_r , compare T_r to t :
 - If $T_r = t$: hold token, run critical section
 - If $T_r > t$: pass token
 - If t not set or $T_r < t$: set token-time to T_r , pass token, wait for token
- To leave critical section:
 - Set token-time to null (i.e., unset it), pass token

Solution 3: A shared priority queue

- By Lamport, using Lamport clocks
- Each process i locally maintains Q_i , part of a shared priority queue
- To run critical section, must have replies from all other processes AND be at the front of Q_i
 - When you have all replies:
 - #1: All other processes are aware of your request
 - #2: You are aware of any earlier requests for the mutex

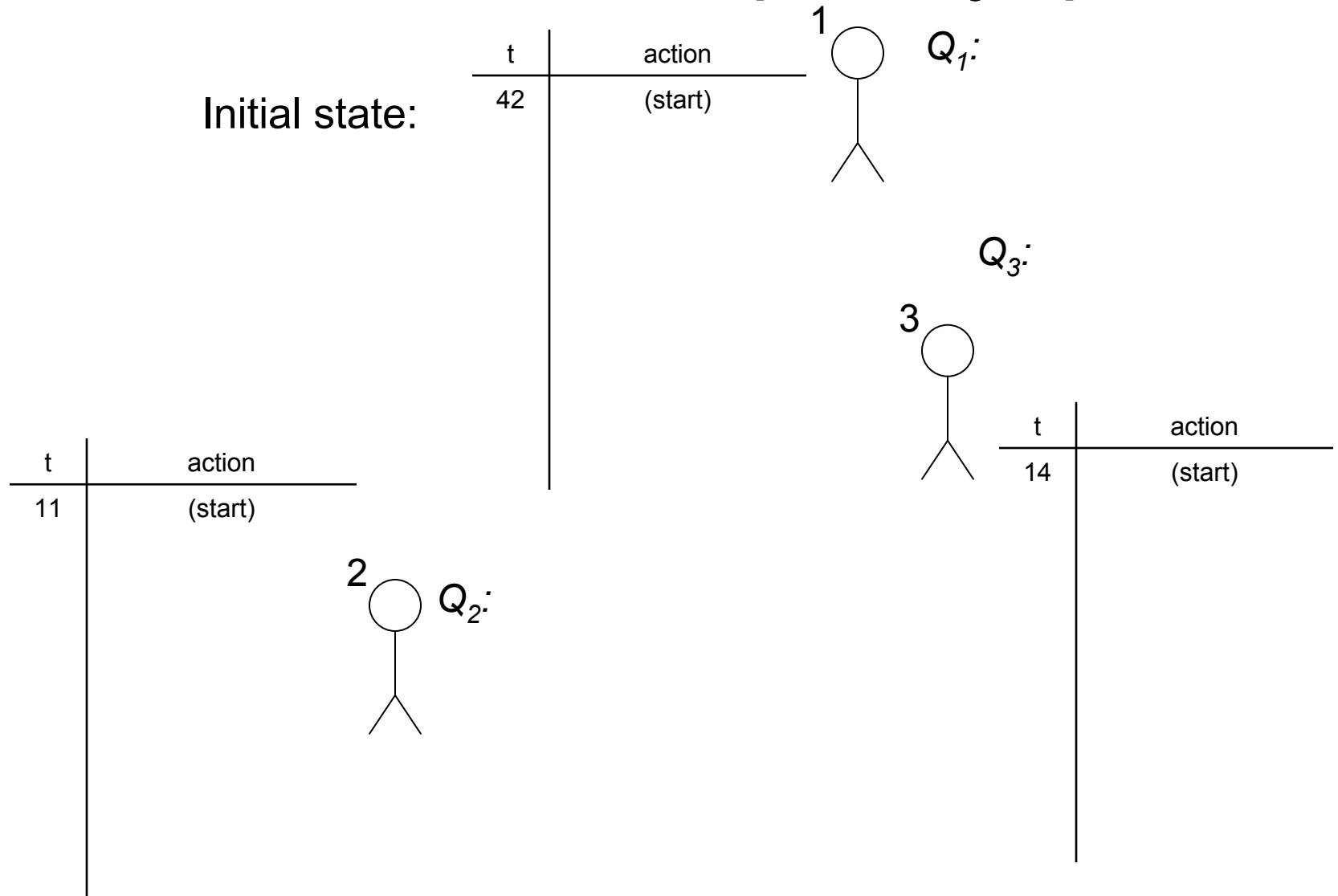
Solution 3: A shared priority queue

- To enter critical section at process i :
 - Stamp your request with the current time T
 - Add request to Q_i
 - Broadcast REQUEST(T) to all processes
 - Wait for all replies and for T to reach front of Q_i
- To leave:
 - Pop head of Q_i , Broadcast RELEASE to all processes
- On receipt of REQUEST(T') from process j :
 - Add T' to Q_i
 - If waiting for REPLY from j for an earlier request T , wait until j replies to you
 - Otherwise REPLY
- On receipt of RELEASE
 - Pop head of Q_i

This delay
enforces
property #2

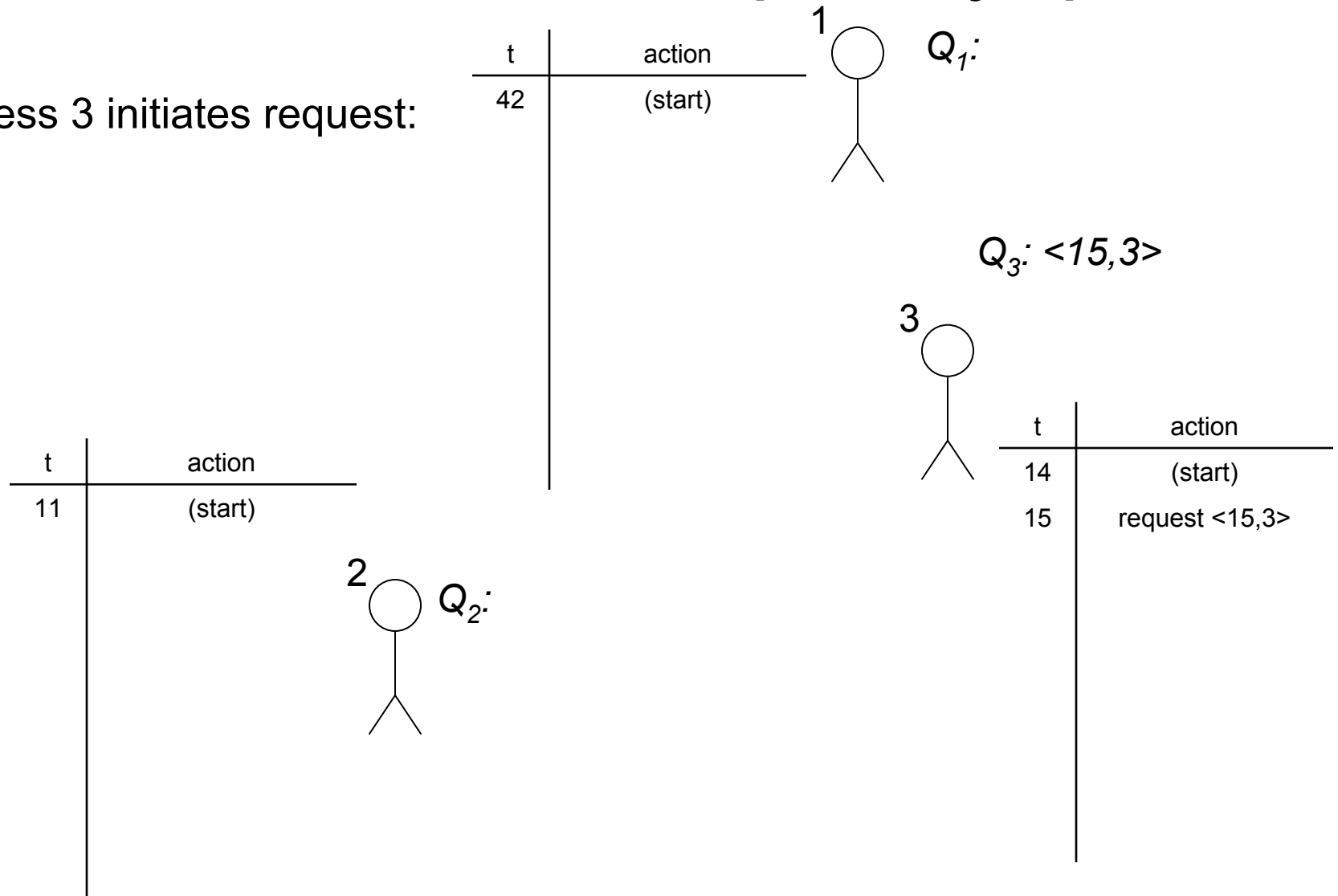


Solution 3: A shared priority queue



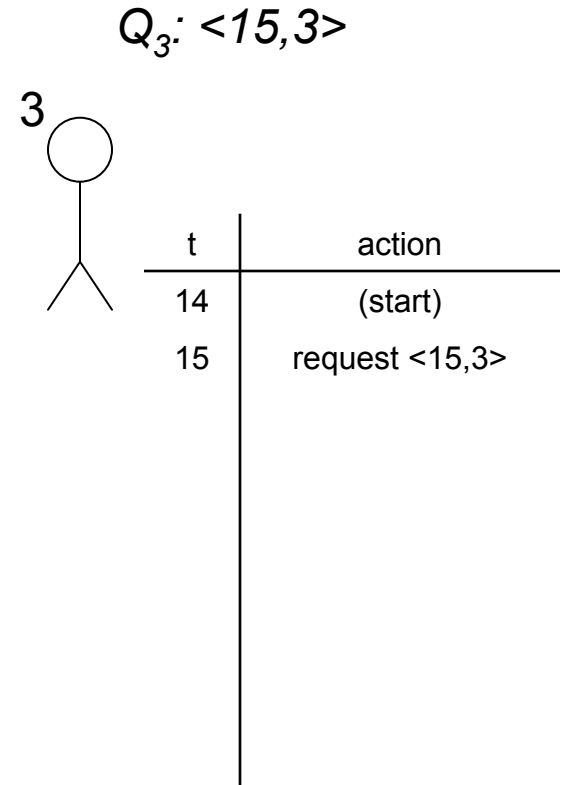
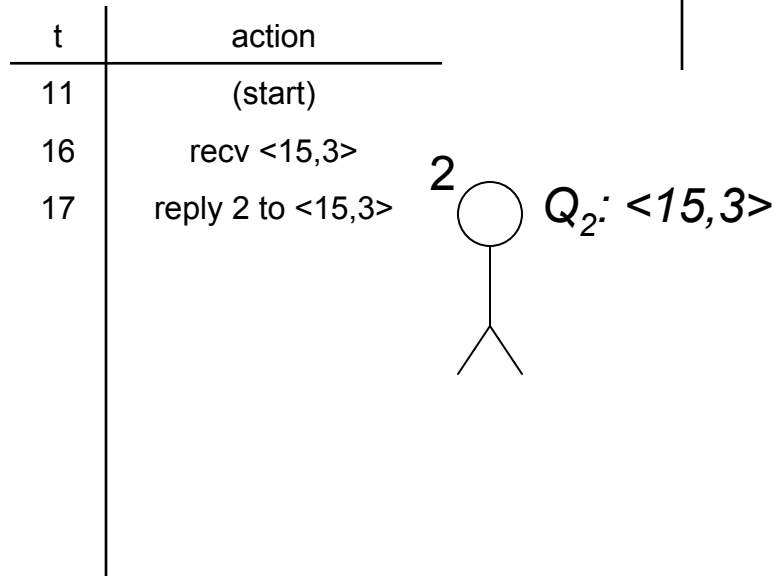
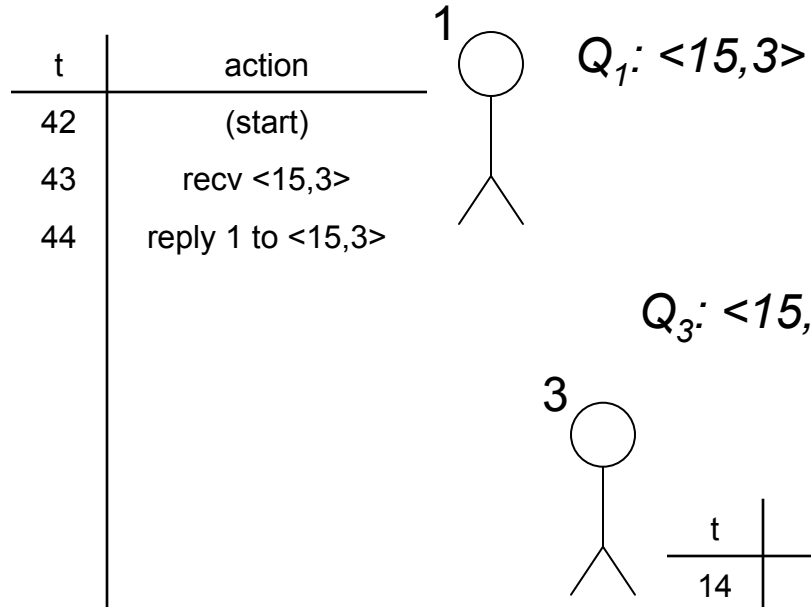
Solution 3: A shared priority queue

Process 3 initiates request:



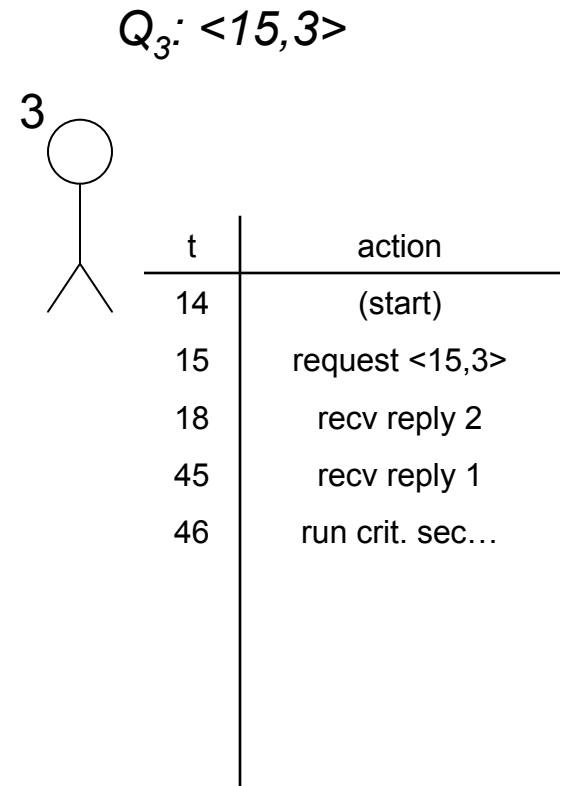
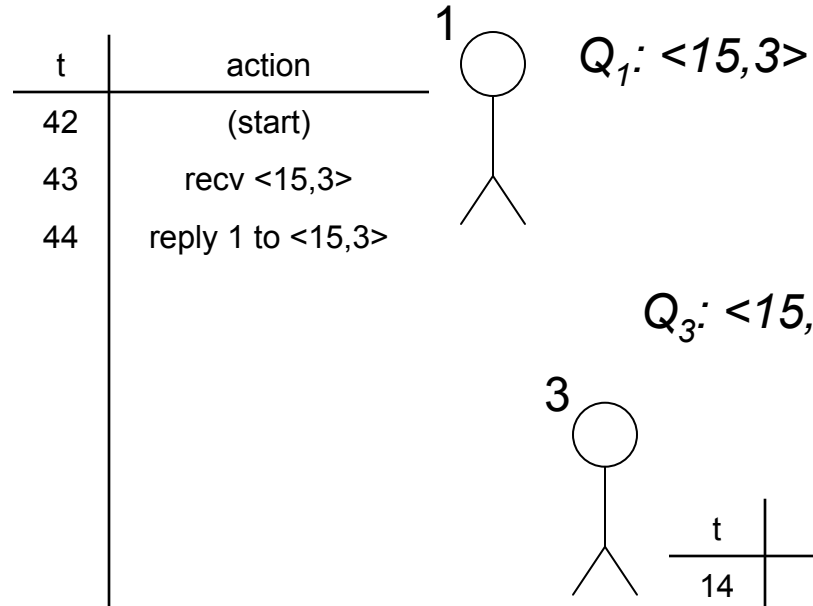
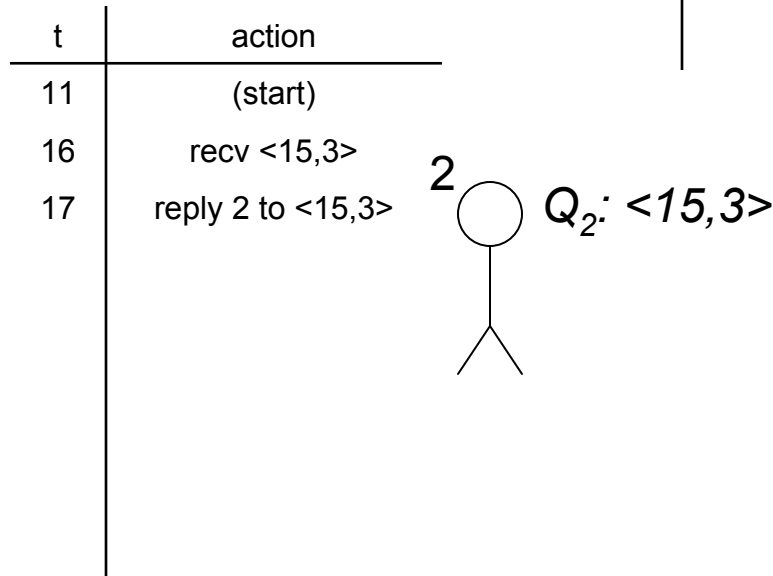
Solution 3: A shared priority queue

1 & 2 receive and reply



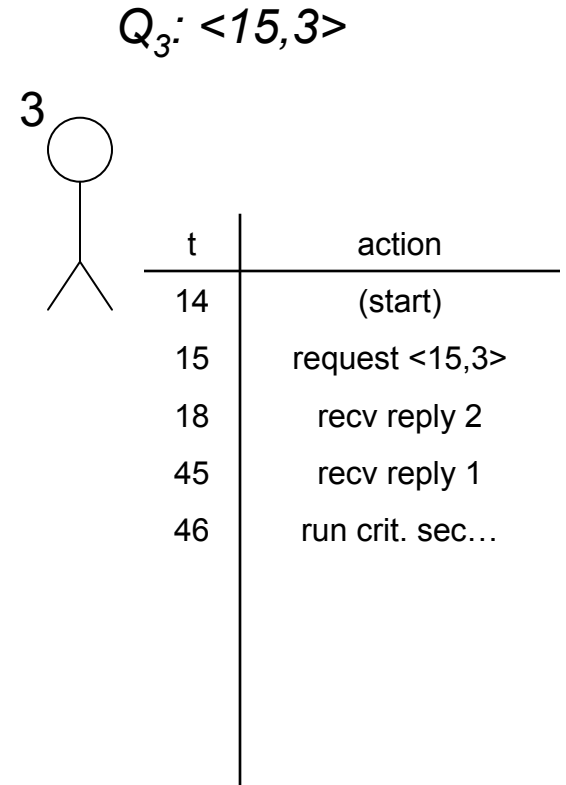
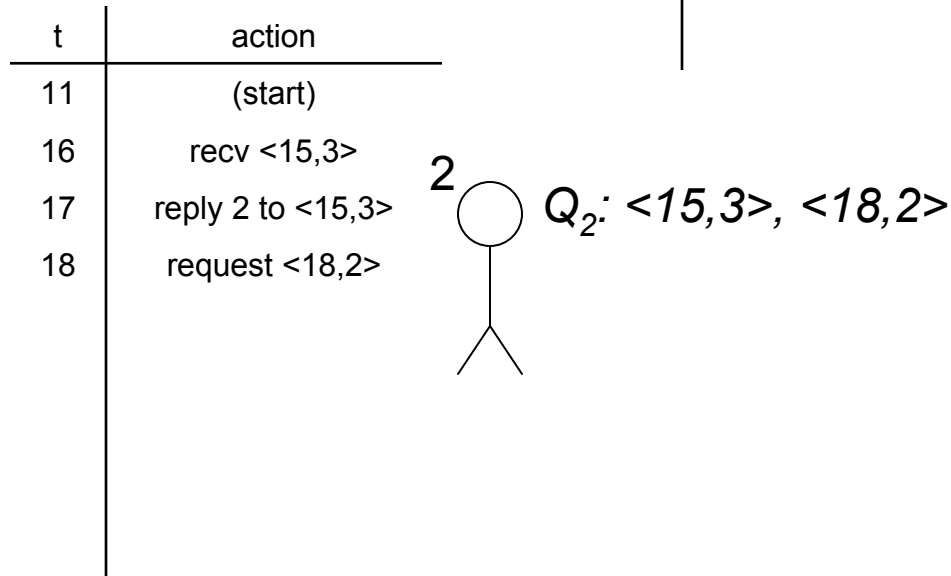
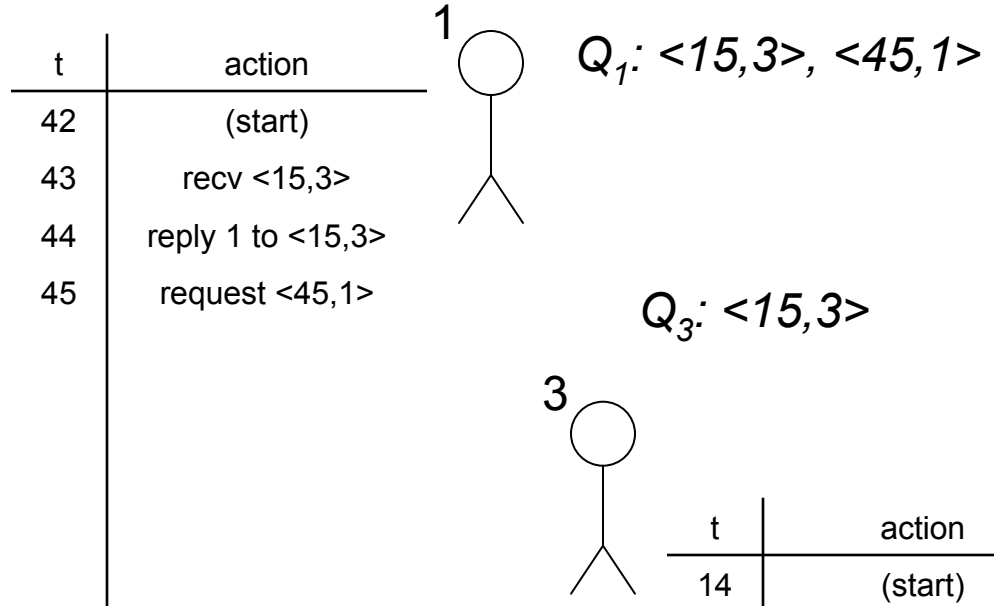
Solution 3: A shared priority queue

3 gets replies, is on front of queue, can run crit. section:



Solution 3: A shared priority queue

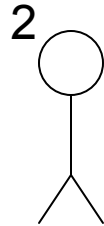
Processes 1 and 2
concurrently initiate
requests:



Solution 3: A shared priority queue

Process 3 gets requests and replies:

t	action
11	(start)
16	recv <15,3>
17	reply 2 to <15,3>
18	request <18,2>
51	recv reply 3

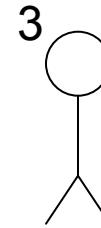


$Q_2: \langle 15,3 \rangle, \langle 18,2 \rangle$

t	action
42	(start)
43	recv <15,3>
44	reply 1 to <15,3>
45	request <45,1>
49	recv reply 3



$Q_1: \langle 15,3 \rangle, \langle 45,1 \rangle$



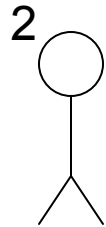
$Q_3: \langle 15,3 \rangle, \langle 18,2 \rangle, \langle 45,1 \rangle$

t	action
14	(start)
15	request <15,3>
18	recv reply 2
45	recv reply 1
46	run crit. sec...
47	recv <45,1>
48	reply 3 to <45,1>
49	recv <18,2>
50	reply 3 to <18,2>

Solution 3: A shared priority queue

Process 2 gets request $\langle 45,1 \rangle$, delays reply because $\langle 18,2 \rangle$ is an earlier request to which Process 1 has not replied

t	action
11	(start)
16	recv $\langle 15,3 \rangle$
17	reply 2 to $\langle 15,3 \rangle$
18	request $\langle 18,2 \rangle$
51	recv reply 3
52	recv $\langle 45,1 \rangle$

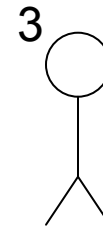


$Q_2: \langle 15,3 \rangle, \langle 18,2 \rangle, \langle 45,1 \rangle$

t	action
42	(start)
43	recv $\langle 15,3 \rangle$
44	reply 1 to $\langle 15,3 \rangle$
45	request $\langle 45,1 \rangle$
49	recv reply 3



$Q_1: \langle 15,3 \rangle, \langle 45,1 \rangle$



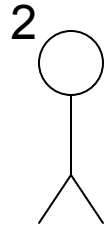
$Q_3: \langle 15,3 \rangle, \langle 18,2 \rangle, \langle 45,1 \rangle$

t	action
14	(start)
15	request $\langle 15,3 \rangle$
18	recv reply 2
45	recv reply 1
46	run crit. sec...
47	recv $\langle 45,1 \rangle$
48	reply 3 to $\langle 45,1 \rangle$
49	recv $\langle 18,2 \rangle$
50	reply 3 to $\langle 18,2 \rangle$

Solution 3: A shared priority queue

Process 1 gets request $\langle 18,2 \rangle$, replies

t	action
11	(start)
16	recv $\langle 15,3 \rangle$
17	reply 2 to $\langle 15,3 \rangle$
18	request $\langle 18,2 \rangle$
51	recv reply 3
52	recv $\langle 45,1 \rangle$

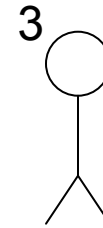


$Q_2: \langle 15,3 \rangle, \langle 18,2 \rangle, \langle 45,1 \rangle$

t	action
42	(start)
43	recv $\langle 15,3 \rangle$
44	reply 1 to $\langle 15,3 \rangle$
45	request $\langle 45,1 \rangle$
49	recv reply 3
50	recv $\langle 18,2 \rangle$
51	reply 1 to $\langle 18,2 \rangle$



$Q_1: \langle 15,3 \rangle, \langle 18,2 \rangle, \langle 45,1 \rangle$



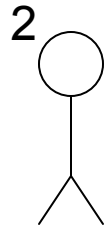
$Q_3: \langle 15,3 \rangle, \langle 18,2 \rangle, \langle 45,1 \rangle$

t	action
14	(start)
15	request $\langle 15,3 \rangle$
18	recv reply 2
45	recv reply 1
46	run crit. sec...
47	recv $\langle 45,1 \rangle$
48	reply 3 to $\langle 45,1 \rangle$
49	recv $\langle 18,2 \rangle$
50	reply 3 to $\langle 18,2 \rangle$

Solution 3: A shared priority queue

Process 2 gets reply from process 1, finally replies to <45,1>

t	action
11	(start)
16	recv <15,3>
17	reply 2 to <15,3>
18	request <18,2>
51	recv reply 3
52	recv <45,1>
53	recv reply 1
54	reply 2 to <45,1>

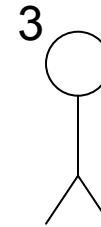


$Q_2: \langle 15,3 \rangle, \langle 18,2 \rangle, \langle 45,1 \rangle$

t	action
42	(start)
43	recv <15,3>
44	reply 1 to <15,3>
45	request <45,1>
49	recv reply 3
50	recv <18,2>
51	reply 1 to <18,2>



$Q_1: \langle 15,3 \rangle, \langle 18,2 \rangle, \langle 45,1 \rangle$



$Q_3: \langle 15,3 \rangle, \langle 18,2 \rangle, \langle 45,1 \rangle$

t	action
14	(start)
15	request <15,3>
18	recv reply 2
45	recv reply 1
46	run crit. sec...
47	recv <45,1>
48	reply 3 to <45,1>
49	recv <18,2>
50	reply 3 to <18,2>

Solution 3: A shared priority queue

- Advantages:
 - Fair
 - Short synchronization delay
- Disadvantages:
 - Very unreliable
 - Any process failure halts progress
 - $3(N-1)$ messages per entry/exit

Solution 4: Ricart and Agrawala

- An improved version of Lamport's shared priority queue
 - Combines function of REPLY and RELEASE messages
- Delay REPLY to any requests later than your own
 - Send all delayed replies after you exit your critical section

Solution 4: Ricart and Agrawala

- To enter critical section at process i :
 - Same as Lamport's algorithm
 - Except you don't need to reach the front of Q_i to run your critical section: you just need all replies
- To leave:
 - Broadcast REPLY to all processes in Q_i
 - Empty Q_i
- On receipt of REQUEST(T'):
 - If waiting for (or in) critical section for an earlier request T , add T' to Q_i
 - Otherwise REPLY immediately

Ricart and Agrawala safety

- Suppose request T_1 is earlier than T_2 . Consider how the process for T_2 collects its reply from process for T_1 :
 - T_1 must have already been time-stamped when request T_2 was received, otherwise the Lamport clock would have been advanced past time T_2
 - But then the process must have delayed reply to T_2 until after request T_1 exited the critical section. Therefore T_2 will not conflict with T_1 .

Solution 4: Ricart and Agrawala

- Advantages:
 - Fair
 - Short synchronization delay
 - Better than Lamport's algorithm
- Disadvantages
 - Very unreliable
 - $2(N-1)$ messages for each entry/exit

Solution 5: Majority rules

- Instead of collecting REPLYs, collect VOTES
 - Each process VOTES for which process can hold the mutex
 - Each process can only VOTE once at any given time
 - You hold the mutex if you have a majority of the VOTES
 - Only possible for one process to have a majority at any given time!

Solution 5: Majority rules

- To enter critical section at process i :
 - Broadcast REQUEST(T), collect VOTES
 - Can enter crit. sec. if collect a majority of VOTES
- To leave:
 - Broadcast RELEASE-VOTE to all processes who VOTEd for you
- On receipt of REQUEST(T') from process j :
 - If you have not VOTEd, VOTE for T'
 - Otherwise, add T' to Q_i
- On receipt of RELEASE-VOTE:
 - If Q_i not empty, VOTE for pop(Q_i)

Solution 5: Majority rules

- Advantages:
 - Can progress with as many as $N/2 - 1$ failed processes
- Disadvantages:
 - Not fair
 - Deadlock!
 - No guarantee that anyone receives a majority of votes

Solution 5': Dealing with deadlock

- Allow processes to ask for their vote back
 - If already VOTEd for T' and get a request for an earlier request T , RESCIND-VOTE for T'
 - If receive RESCIND-VOTE request and not in critical section, RELEASE-VOTE and re-REQUEST
- Guarantees that some process will eventually get a majority of VOTES
- Still not fair...

Algorithm Comparison

Algorithm	Messages per entry/exit	Synchronization delay (in RTTs)	Liveness
Central server	3	1 RTT	Bad: coordinator crash prevents progress
Token ring	N	$\leq \text{sum}(\text{RTTs})/2$	Horrible: any process' failure prevents progress
Lamport	$3*(N-1)$	max(RTT) across processes	Horrible: any process' failure prevents progress
Ricart & Agrawal	$2*(N-1)$	max(RTT) across processes	Horrible: any process' failure prevents progress
Voting	$\geq 2*(N-1)$ (might have vote recalls, too)	max(RTT) between the fastest $N/2+1$ processes	Great: can tolerate up to $N/2-1$ failures

/

You want the lock; no one else has it; how long till you get it?

So, Who Wins?

- Well, none of the algorithms we've looked at thus far
- But the closest one to industrial standards is...

So, Who Wins?

- Well, none of the algorithms we've looked at thus far
- But the closest one to industrial standards is...
 - **The centralized model** (e.g., Google's Chubby, Yahoo's ZooKeeper)

So, Who Wins?

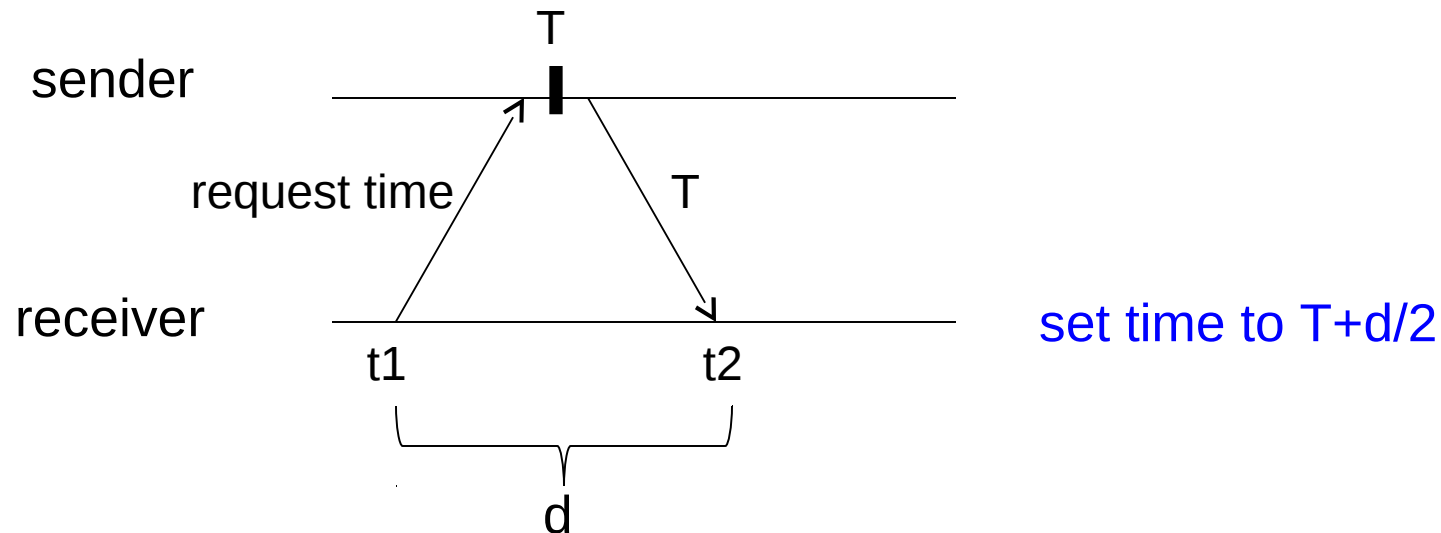
- Well, none of the algorithms we've looked at thus far
- But the closest one to industrial standards is...
 - **The centralized model** (e.g., Google's Chubby, Yahoo's ZooKeeper)
 - But **replicate** it for fault-tolerance across a few machines
 - Replicas coordinate closely via mechanisms similar to the ones we've shown for the distributed algorithms (e.g., voting) – we'll talk later about generalized voting alg.
 - For manageable load, app writers must **avoid using the centralized lock service** as much as humanly possible!

Take-Aways

- Lamport algorithm demonstrates how distributed processes can maintain consistent replicas of a data structure (the priority queue)!
- Lamport and Ricart & Agrawala's algorithms demonstrate utility of logical clocks
- If you build your distributed system wrong, then you get worse properties from distribution than if you didn't distribute at all
- None of these algorithms can tolerate dropped messages

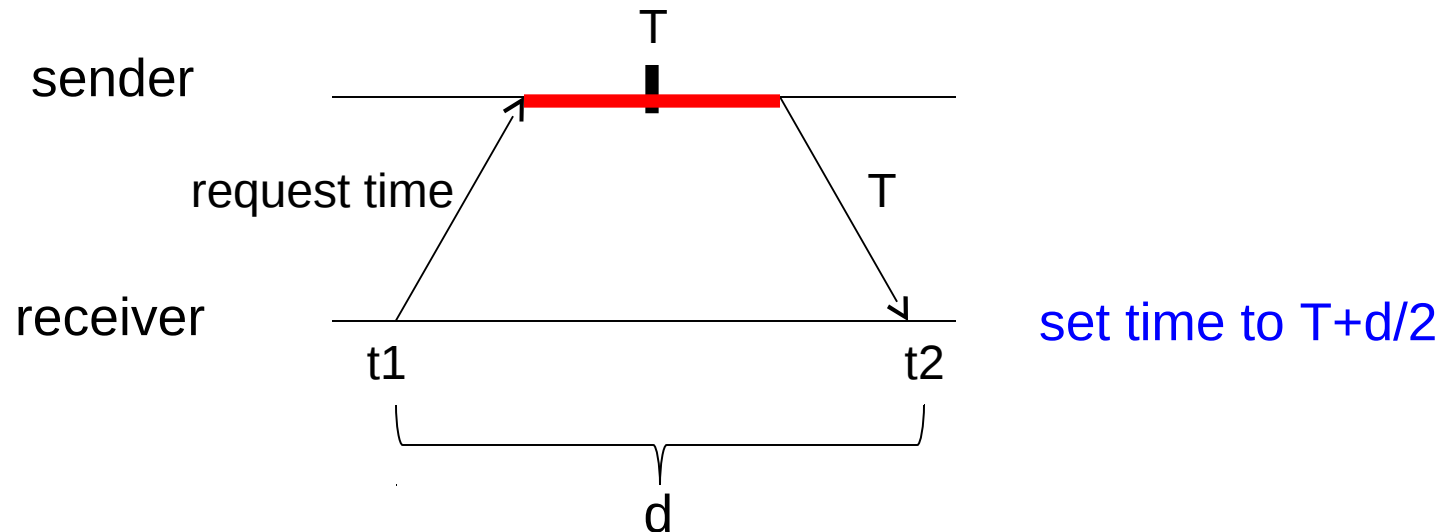
Clarification for Last Lecture: The NTP Protocol

- Uses a hierarchy of time servers
- Synchronization similar to Cristian's alg.
 - Modified to use multiple one-way messages instead of immediate round-trip



Clarification for Last Lecture: The NTP Protocol

- Uses a hierarchy of time servers
- Synchronization similar to Cristian's alg.
 - Modified to use multiple one-way messages instead of immediate round-trip



Clarification for Last Lecture: The NTP Protocol

- Uses a hierarchy of time servers
- Synchronization similar to Cristian's alg.
 - Modified to use multiple one-way messages instead of immediate round-trip

