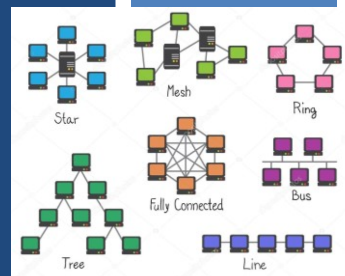
# TCSS 558: APPLIED DISTRIBUTED COMPUTING




## Chapter 6 – Coordination

Wes J. Lloyd
School of Engineering & Technology (SET)
University of Washington - Tacoma

1

## OBJECTIVES – 2/27

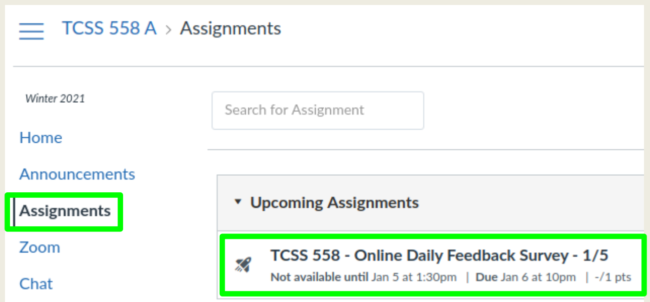
- Questions from 2/22
- Assignment 3: Replicated Key Value Store
- Chapter 4.4 - Review / Finish
- Chapter 6: Coordination
  - Chapter 6.1: Clock Synchronization
  - Chapter 6.2: Logical Clocks
    Vector Clocks
- Class Activity – Total Ordered Multicasting (Thursday)
  - Chapter 6.3: Distributed Mutual Exclusion

| February 27, 2024 | TCSS558: Applied Distributed Computing [Winter 2024]<br>School of Engineering and Technology, University of Washington - Tacoma | L15.2 |
|---|---|---|

2

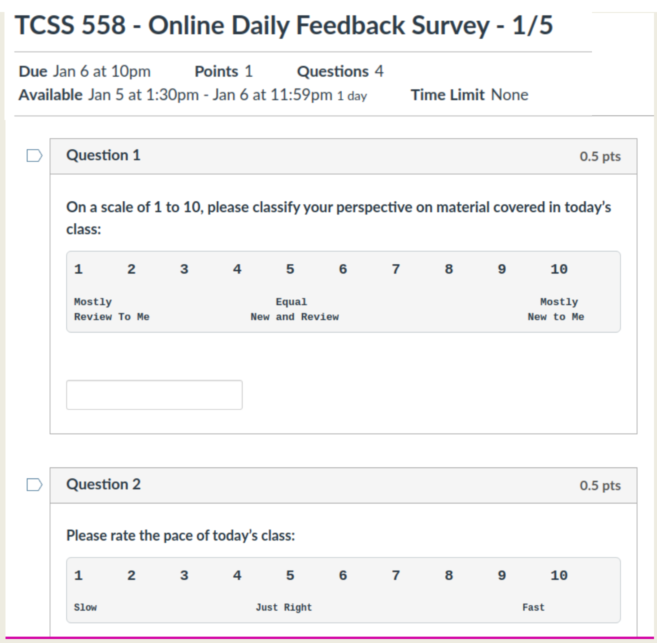
# ONLINE DAILY FEEDBACK SURVEY

- Daily Feedback Quiz in Canvas – Available After Each Class
- Extra credit available for completing surveys ***ON TIME***
- Tuesday surveys: due by ~ Wed @ 10p
- Thursday surveys: due ~ Mon @ 10p




| February 27, 2024 | TCSS558: Applied Distributed Computing [Winter 2024]<br>School of Engineering and Technology, University of Washington - Tacoma | L15.3 |
|---|---|---|

3




February 27, 2024 TCSS558: Applied Distributed Computing [Winter 2024]
School of Engineering and Technology, University of Washington - Tacoma L15.4

4

## MATERIAL / PACE

- Please classify your perspective on material covered in today's class (22 respondents):
- 1-mostly review, 5-equal new/review, 10-mostly new
- **Average – 6.09 (↓ - *previous* 6.60)**

- Please rate the pace of today's class:
- 1-slow, 5-just right, 10-fast
- **Average – 5.36 (↓ - *previous* 5.56)**



5

## CSS TENURE TRACK FACULTY CANDIDATE RESEARCH SEMINARS – EXTRA CREDIT

- **Week 9:**
  - Wednesday February 28 – 1:30pm – JOY 117
  - Thursday February 29 – 1:30pm – MLG 110
  - Friday March 1 – 1:30pm – MLG 301
- **Week 10:**
  - Monday March 4 – 1:30pm - MLG 110
  - Tuesday March 5 – 1:30pm - CP 324
  - Wednesday March 6 – 1:30pm – BHS 106
  - Thursday March 7 – 12:30pm – MLG 110

- Earn up to 33 buffer points added to the Final Exam score
- Earn 3 points for each seminar attended
- Buffer points replace missed points on the Final Exam
- Once the Final Exam score = 100%, additional points do not push the Final Exam score above 100%
- Buffer points will not impact the course curve for the Final Exam
- Any course curve will be applied before buffer points



6

# FEEDBACK FROM 2/22



7

# OBJECTIVES – 2/27

- Questions from 2/22
- **Assignment 3: Replicated Key Value Store**
- Chapter 4.4 - Review / Finish
- Chapter 6: Coordination
  - Chapter 6.1: Clock Synchronization
  - Chapter 6.2: Logical Clocks
    Vector Clocks
- Class Activity – Total Ordered Multicasting (Thursday)
  - Chapter 6.3: Distributed Mutual Exclusion



8

## SHORT-HAND-CODES FOR MEMBERSHIP TRACKING APPROACHES

- Include readme.txt or doc file with instructions in submission
- Must document membership tracking method

***>> please indicate which types to test <<***

| ID | Description |
|---|---|
| F | Static file membership tracking – file is not reread |
| FD | Static file membership tracking DYNAMIC - file is periodically reread to refresh membership list |
| T | TCP membership tracking – servers are configured to refer to central membership server |
| U | UDP membership tracking - automatically discovers nodes with no configuration |



9

## ASSIGNMENT 3

- **Sunday March 10th**
- Goal: Replicated Key Value Store
- Team signup to be posted on Canvas under 'People'
- Build off of Assignment 2 GenericNode
- Focus on TCP client/server w/ replication
- How to track membership for data replication?
  - Can implement multiple types of membership tracking for extra credit
- REQUIREMENT: 'store' command needs to output 1 key-value pair per line using ASCII text (no binary)



10

# OBJECTIVES – 2/27

- Questions from 2/22
- Assignment 3: Replicated Key Value Store
- **Chapter 4.4 - Review / Finish**
- Chapter 6: Coordination
  - Chapter 6.1: Clock Synchronization
  - Chapter 6.2: Logical Clocks
    Vector Clocks
- Class Activity – Total Ordered Multicasting (Thursday)
  - Chapter 6.3: Distributed Mutual Exclusion



11

# RUMOR SPREADING

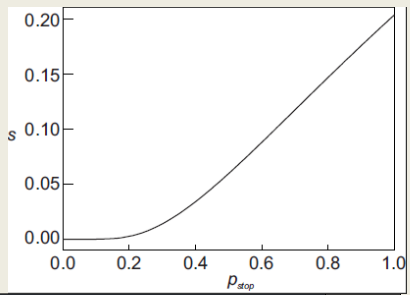
- Variant of epidemic protocols
- Provides an approach to **"stop"** message spreading
- Mimics "gossiping" in real life
- **Rumor spreading:**
- **Node P** receives new data **item X**
- Contacts an arbitrary **node Q** to push update
- **Node Q** reports already receiving **item X** from another node
- **Node P** may loose interest in spreading the rumor with probability = $p_{stop}$, let's say 20% . . . (or 0.20)



12

# RUMOR SPREADING - 2

- $p_{stop}$, is the probability node will stop spreading once contacting a node that already has the message
- Rumor spreading does not guarantee all nodes will be updated
- Fraction of nodes s, that remain susceptible grows relative to the probability that node P stops propagating when finding a node already having the message
- Fraction of nodes not updated remains < 0.20 with high $p_{stop}$
- Susceptible nodes (s) vs. probability of stopping →






13

# REMOVING DATA

- Gossiping is good for spreading data
- **<u>But how can data be removed from the system?</u>**
- Idea is to issue ***"death certificates"***
- Act like data records, which are spread like data
- When death certificate is received, data is deleted
- Certificate is held to prevent data element from reinitializing from gossip from other nodes
- Death certificates time-out after expected time required for data element to clear out of entire system
- A few nodes maintain death certificates forever



14

# DEATH CERTIFICATE EXAMPLE

- **For example:**
- **Node P** keeps death certificates forever
- **Item X** is removed from the system
- **Node P** receives an update request for **Item X**, but ***also*** holds the death certificate for **Item X**
- **Node P** will recirculate the death certificate across the network for **Item X**



# OBJECTIVES – 2/27

- Questions from 2/22
- Assignment 3: Replicated Key Value Store
- Chapter 4.4 - Review / Finish
- Chapter 6: Coordination
  - **Chapter 6.1: Clock Synchronization**
  - Chapter 6.2: Logical Clocks
    Vector Clocks
- Class Activity – Total Ordered Multicasting (Thursday)
  - Chapter 6.3: Distributed Mutual Exclusion

# CHAPTER 6 - COORDINATION

- 6.1 Clock Synchronization
  - Physical clocks
  - Clock synchronization algorithms
- 6.2 Logical clocks
  - Lamport clocks
  - Vector clocks
- 6.3 Mutual exclusion
- 6.4 Election algorithms
- 6.6 Distributed event matching *(light)*
- 6.7 Gossip-based coordination *(light)*



# CH. 6.1: CLOCK SYNCHRONIZATION

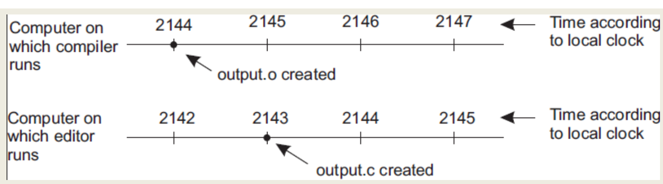
# CLOCK SYNCHRONIZATION

- Example:
- "make" is used to compile source files into binary object and executable files
- As an optimization, make only compiles files when the "last modified time" of source files is more recent than object and executables
- Consider if files are on a shared disk of a distributed system where there is no agreement on time
- Consider if the program has 1,000 source files





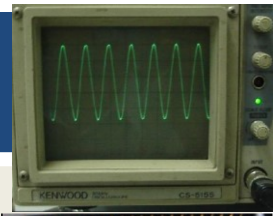
# TIME SYNCHRONIZATION PROBLEM FOR DISTRIBUTED SYSTEMS




- Updates from different machines, may have clocks set to different times
- Make becomes confused with which files to recompile
- Linux commands:

```
date +%s     # seconds since Jan 1, 1970
```

# PHYSICAL CLOCKS



- **Computer timers:** precisely machined quartz crystals
- When under tension, they oscillate at a well defined frequency
- In analog electronics/communications crystals once used to set the frequency of two-way radio transceivers for
- Today, crystals are associated with a counter and holding register on a digital computer.

**1960s ERA radio crystal →**

- Each oscillation decrements a counter by one
- When counter gets to zero, an interrupt fires
- Can program timer to generate interrupt, let's say 60 times a second, or another frequency to track time

| February 27, 2024 | TCSS558: Applied Distributed Computing [Winter 2024]<br>School of Engineering and Technology, University of Washington - Tacoma | L15.21 |
|---|---|---|

21

# COMPUTER CLOCKS

- Digital clock on computer sets base time
- Crystal clock tracks forward progress of time
  - Translation of wave "ticks" to clock pulses
- CMOS battery on motherboard maintains clock on power loss

- **Clock skew**: physical clock crystals are not exactly the same
- Some run at slightly different rates
- Time differences accumulate as clocks drift forward or backward slightly

- In an automobile, where there is no clock synchronization, clock skew may become noticeable over months, years

| February 27, 2024 | TCSS558: Applied Distributed Computing [Winter 2024]<br>School of Engineering and Technology, University of Washington - Tacoma | L15.22 |
|---|---|---|

22

# UNIVERSAL COORDINATED TIME

```
ubuntu@ip-172-31-58-89:~$ date
Thu Nov 16 10:13:39 UTC 2017
```

- Universal Coordinated Time (UTC)
  - Worldwide standard for time keeping
  - Equivalent to Greenwich Mean Time (United Kingdom)
  - 40 shortwave radio stations around the world broadcast a short pulse at the start of each second (WWV)
  - World wide "atomic" clocks powered by constant transitions of the non-radioactive caesium-133 atom
    - 9,162,631,770 transitions per second
- Computers track time using UTC as a base
  - Avoid thinking in local time, which can lead to coordination issues
  - Operating systems may translate to show local time

| February 27, 2024 | TCSS558: Applied Distributed Computing [Winter 2024]<br>School of Engineering and Technology, University of Washington - Tacoma | L15.23 |
|---|---|---|

23

# COMPUTING: CLOCK CHALLENGES

- How do we synchronize computer clocks with real-world clocks?
- How do we synchronize computer clocks with each other?

| February 27, 2024 | TCSS558: Applied Distributed Computing [Winter 2024]<br>School of Engineering and Technology, University of Washington - Tacoma | L15.24 |
|---|---|---|

24

# CLOCK SYNCHRONIZATION

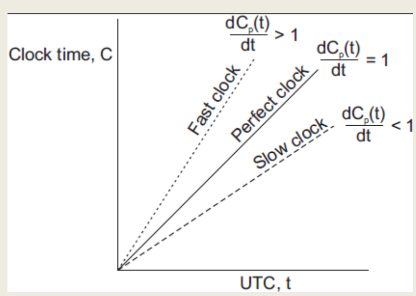
- **UTC services**: use radio and satellite signals to provide time accuracy to 50ns
- **Time servers**: Server computers with UTC receivers that provide accurate time
- **Precision** ($\pi$): how close together a set of clocks may be
- **Accuracy**: how correct to actual time clocks may be
- **Internal synchronization**: Sync local computer clocks
- **External synchronization**: Sync to UTC clocks
- **Clock drift**: clocks on different machines gradually become out of sync due to crystal imperfections, temperature differences, etc.
- **Clock drift rate**: typical is 31.5s per year
- **Maximum clock drift rate** ($\rho$): clock specifications include one

| February 27, 2024 | TCSS558: Applied Distributed Computing [Winter 2024]<br>School of Engineering and Technology, University of Washington - Tacoma | L15.25 |
|---|---|---|

25

# CLOCK SYNCHRONIZATION - 2

- If two clocks drift from UTC in opposite directions, after time $\Delta t$ after synchronization, they may be $2\rho$ apart.
  - $\rho$ - clock drift rate, $\pi$ - clock precision (max 50ns)
- Clocks must be resynchronized every $\pi/2\rho$ seconds
- **Network time protocol**
- Provide coordination of time for servers
- Leverage distributed network of time servers

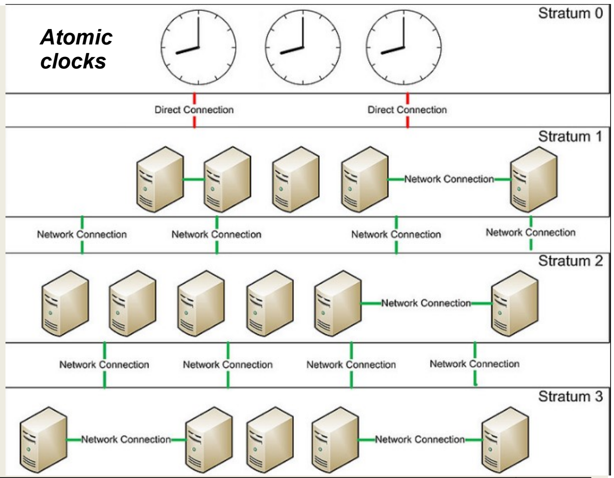



| February 27, 2024 | TCSS558: Applied Distributed Computing [Winter 2024]<br>School of Engineering and Technology, University of Washington - Tacoma | L15.26 |
|---|---|---|

26

## NETWORK TIME PROTOCOL

- Servers organized into stratums
- Stratum-1 servers have UTC receivers and are sync'd with atomic clocks
- Servers connect with closest NTP server for time synchronization
- Servers assume role as NTP server at stratum+1




February 27, 2024 | TCSS558: Applied Distributed Computing [Winter 2024] School of Engineering and Technology, University of Washington - Tacoma | L15.27

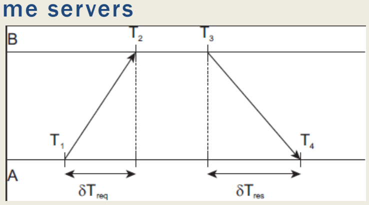
27

## NTP - 2

- Must estimate network delays when synchronizing with remote UTC receiver clocks / time servers

Time server B

Client A




1. A sends message to B, with timestamp T1
2. B records time of receipt T2 (from local clock)
3. B returns response with send time T3, and receipt time T2
4. A records arrival of T4

- Assuming propagation delay of A→B→A is the same
- Estimate propagation delay:
- Add delay to time

$$\theta = T_3 + \frac{(T_2 - T_1) + (T_4 - T_3)}{2} - T_4 = \frac{(T_2 - T_1) + (T_3 - T_4)}{2}$$

February 27, 2024 | TCSS558: Applied Distributed Computing [Winter 2024] School of Engineering and Technology, University of Washington - Tacoma | L15.28

28

## NTP - 3

- Cannot set clocks backwards (recall "make" file example)
- Instead, temporarily slow the progress of time to allow fast clock to align with actual time
- Change rate of clock interrupt routine
- Slow progress of time until synchronized
- NTP accuracy is within 1-50ms

- In Ubuntu Linux, to quickly synchronize time:
  `$apt install ntp ntpdate`
- Specify local timeservers in /etc/ntp.conf
  ```
  server time.u.washington.edu iburst
  server bigben.cac.washington.edu iburst
  ```
- Shutdown service (sudo service ntp stop)
- Run ntpdate: (sudo ntpdate time.u.washington.edu)
- Startup service (sudo service ntp start)

| February 27, 2024 | TCSS558: Applied Distributed Computing [Winter 2024]<br>School of Engineering and Technology, University of Washington - Tacoma | L15.29 |
|---|---|---|

29

## AWS EC2 INSTANCE – TIME SYNCHRONIZATION

- Amazon uses a variant of ntp called **chrony** "chron" is time in Greek
- By default "chrony" is preinstalled on standard AMIs for ec2 instances (i.e. Ubuntu 22.04, Amazon Linux, etc.)
- Installation instructions:
- https://docs.aws.amazon.com/AWSEC2/latest/UserGuide/set-time.html
- Once installed you can monitor clock drift with:
  `watch -n .2 chronyc tracking`
- Can publish clock drift using bash script as a CloudWatch metric:
- https://aws.amazon.com/blogs/mt/manage-amazon-ec2-instance-clock-accuracy-using-amazon-time-sync-service-and-amazon-cloudwatch-part-2/
- Upgrade script to Instance Metadata Service v2:
- https://docs.aws.amazon.com/AWSEC2/latest/UserGuide/instance-metadata-v2-how-it-works.html

| February 27, 2024 | TCSS558: Applied Distributed Computing [Winter 2024]<br>School of Engineering and Technology, University of Washington - Tacoma | L15.30 |
|---|---|---|

30

# WATCH -N .2 CHRONYC TRACKING

```
ubuntu@ip-172-31-47-121: ~
Every 0.2s: chronyc tracking

Reference ID    : A9FEA97B (169.254.169.123)
Stratum         : 4
Ref time (UTC)  : Tue Feb 27 11:39:58 2024
System time     : 0.000001064 seconds fast of NTP time
Last offset     : +0.000000921 seconds
RMS offset      : 0.000000628 seconds
Frequency       : 2.687 ppm fast
Residual freq   : +0.000 ppm
Skew            : 0.019 ppm
Root delay      : 0.000425237 seconds
Root dispersion : 0.000298539 seconds
Update interval : 16.0 seconds
Leap status     : Normal
```



# LINUX CRON

- Linux scheduling facility
  - Cron: background process to run scheduled tasks at specified times
  - Supports running maintenance jobs, scripts at regular intervals
  - Can schedule script to run at specific time of day or interval
  - Highest frequency: once per minute
  - /etc/crontab file captures scheduled tasks
  - By default, runs scripts in /etc/cron.hourly
    /etc/cron.daily
    /etc/cron.weekly
    /etc/cron.monthly

```
# Example of job definition:
# .---------------- minute (0 - 59)
# |  .------------- hour (0 - 23)
# |  |  .---------- day of month (1 - 31)
# |  |  |  .------- month (1 - 12) OR jan,feb,mar,apr ...
# |  |  |  |  .---- day of week (0 - 6) (Sunday=0 or 7)
# |  |  |  |  |
# *  *  *  *  * user-name command to be executed
```

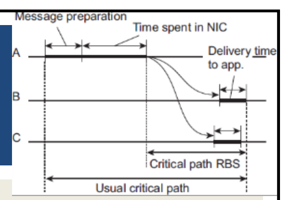
# BERKELEY ALGORITHM

- Berkeley time daemon server actively polls network to determine average time across servers
- Suitable when no machine has a UTC receiver
- Time daemon instructs servers how much to adjust clocks to achieve precision
- Accuracy can not be guaranteed
- Berkeley is an internal clock synchronization algorithm

| February 27, 2024 | TCSS558: Applied Distributed Computing [Winter 2024]<br>School of Engineering and Technology, University of Washington - Tacoma | L15.33 |
|---|---|---|

33

# CLOCK SYNCHRONIZATION IN WIRELESS NETWORKS




- Sensor networks bring unique challenges for clock synchronization
  - ***Address resource constraints***: limited power, multihop routing slow
- **Reference broadcast synchronization (RBS)**
- Provides time precision
- Where no UTC clock available
- RBS sender broadcasts a reference message to allow receivers to adjust clocks
- Assume: NO multi-hop routing
- Assume: Time to propagate a signal to nodes is roughly constant
- RBS: Message propagation time does not consider time spent waiting in NIC for message to send
  - Wireless network resource contention may force waiting before message can be sent – RBS only pays attention to msg receipt time

| February 27, 2024 | TCSS558: Applied Distributed Computing [Winter 2024]<br>School of Engineering and Technology, University of Washington - Tacoma | L15.34 |
|---|---|---|

34

# REFERENCE BROADCAST SYNCHRONIZATION (RBS)

- Node broadcasts reference message k
- Each node p records time Tp,k when k is received
- Tp,k is read from node p's clock
- Two nodes p and q can exchange delivery times to estimate mutual relative offset
- Then calculate relative average offset for each other:

$$Offset[p,q] = \frac{\sum_{k=1}^{M}(T_{p,k} - T_{q,k})}{M}$$

- Where M is the total number of reference messages sent
- To save battery life: nodes store offsets instead of frequently synchronizing clocks to save energy

| February 27, 2024 | TCSS558: Applied Distributed Computing [Winter 2024]<br>School of Engineering and Technology, University of Washington - Tacoma | L15.35 |
|---|---|---|

35

# REFERENCE BROADCAST SYNCHRONIZATION (RBS) - 2

- Cloud skew: over time clocks drift apart
- Averages become less precise
- Elson et al. propose using standard linear regression to predict offsets, rather than calculating them
- IDEA: Use node's history of message times in a simple linear regression to continuously refine a formula with coefficients to predict time offsets:

$$Offset[p,q](t) = \alpha t + \beta$$

- Models the clock drift so time offsets can be inferred

| February 27, 2024 | TCSS558: Applied Distributed Computing [Winter 2024]<br>School of Engineering and Technology, University of Washington - Tacoma | L15.36 |
|---|---|---|

36

# WE WILL RETURN AT 4:55 PM

37

# OBJECTIVES – 2/27

- Questions from 2/22
- Assignment 3: Replicated Key Value Store
- Chapter 4.4 - Review / Finish
- Chapter 6: Coordination
  - Chapter 6.1: Clock Synchronization
  - **Chapter 6.2: Logical Clocks**
    Vector Clocks
- Class Activity – Total Ordered Multicasting (Thursday)
  - Chapter 6.3: Distributed Mutual Exclusion



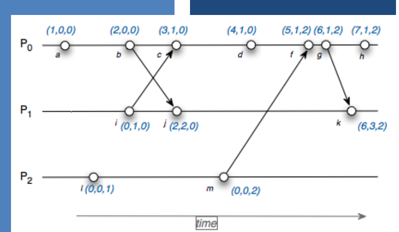
38

# CHAPTER 6 - COORDINATION

- 6.1 Clock Synchronization
  - Physical clocks
  - Clock synchronization algorithms
- 6.2 Logical clocks
  - Lamport clocks
  - Vector clocks
- 6.3 Mutual exclusion
- 6.4 Election algorithms
- 6.6 Distributed event matching *(light)*
- 6.7 Gossip-based coordination *(light)*



39




# CH. 6.2: LOGICAL CLOCKS

L15.40

40

# LOGICAL CLOCKS

- In distributed systems, synchronizing to actual time may not be required…
- It may be sufficient for every node to simply agree on a current time (e.g. logical)
- ***Logical clocks*** provide a mechanism for capturing chronological and ***causal*** relationships in a distributed system
- Think ***counters . . .***
- Leslie Lamport [1978] seminal paper showed that absolute clock synchronization often is not required
- Processes simply need to agree on the order in which events occur

| | | |
|---|---|---|
| **February 27, 2024** | TCSS558: Applied Distributed Computing [Winter 2024]<br>School of Engineering and Technology, University of Washington - Tacoma | L15.41 |

41

# LOGICAL CLOCKS - 2

- **Happens-before relation**
- A→B: **Event A**, happens before **event B**…
- All processes must agree that **event A** occurs first
- Then afterward, **event B**
- Actual time not important. . .
- If **event A** is the event of proc P1 sending a msg to a proc P2, and **event B** is the event of proc P2 receiving the msg, then A→B is also true. . .
- The assumption here is that message delivery takes time
- Happens before is a ***transitive relation***:
- A→B, B→C, therefore A→C

| | | |
|---|---|---|
| **February 27, 2024** | TCSS558: Applied Distributed Computing [Winter 2024]<br>School of Engineering and Technology, University of Washington - Tacoma | L15.42 |

42

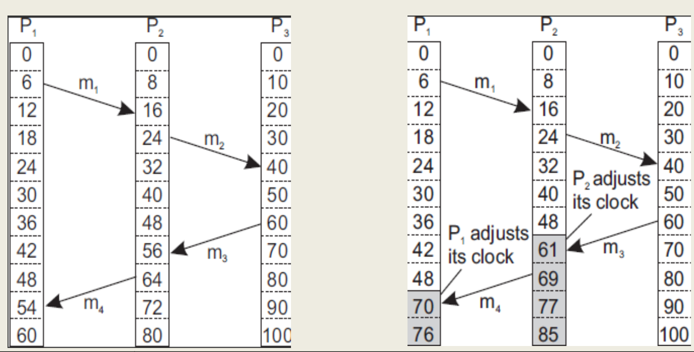
# LOGICAL CLOCKS – 3

- If two events, say event X and event Y do not exchange messages, not even via third parties, then the sequence of X→Y vs. Y→X **can not be determined!!**
- Within the system, these events appear **concurrent**
- **Concurrent:** nothing can be said about when the events happened, or which event occurred first
- Clock time, C, must always go forward (increasing), never backward (decreasing)
- Corrections to time can be made by adding a positive value, but never by subtracting one



# LOGICAL CLOCKS - 4

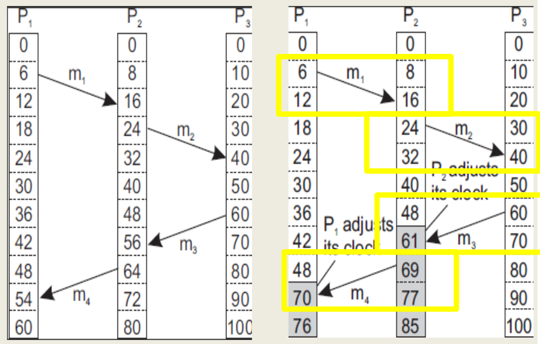
- Three processes each with local clocks
- ***Lamport's algorithm*** corrects process clock values
- Always propagate the most recent known value of logical time



# LOGICAL CLOCKS

- **Events**:

6: P1 send m1 to P2
16: P2 receives m1
24: P2 sends m2 to P3
40: P3 receives m2
60: P3 sends m3 to P2
56: P2 receives m3
56: P2 clock reset=61
69: P2 sends m4 to P1
54: P1 receives m4
70: P1 clock reset=70






45

# LAMPORT LOGICAL CLOCKS - IMPLEMENTATION

- Negative values not possible
- When a message is received, and the local clock is before the timestamp when then message was sent, the local clock is updated to message_sent_time + 1

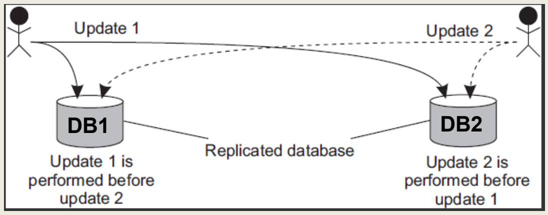
1. Clock is incremented before an event: (*sending-a-message, receiving-a-message, some-other-internal-event*)
   Pi increments Ci: Ci ← Ci + 1
2. When Pi send msg m to Pj, m's timestamp is set to Ci
3. When Pj receives msg m, Pj adjusts its local clock
   Cj ← max{Cj, timestamp(m)}
4. Ties broken by considering Proc ID: i<j; <40,i> < <40,j>
   Both Lamport clocks are = 40
   The winner has a higher alphanumeric Process ID
   J (winner) is greater than i, alphabetically



46

# TOTAL-ORDERED MULTICASTING

- Consider concurrent updates to a replicated database
- Communication latency between DB1 and DB2 is 250ms




- **Initial Account balance**: $1,000
- **Update #1**: Deposit $100
- **Update #2**: Add 1% Interest
- Total Ordered Multicasting needed



# TOTAL-ORDERED MULTICASTING EXAMPLE

- Two messages ($m_1$, $m_2$) must be distributed, to two processes ($p_1$, $p_2$)
- We assume messages have correct lamport clock timestamps
- $m_1$(10, $p_1$, add $100)
- $m_2$(12, $p_2$, add 1% interest)

- Each process maintains a queue of messages
- Arriving messages are placed into queues ordered by the Lamport clock timestamp

- In each queue, each message must be acknowledged by every process in the system before operations can be applied to the local database

# TOTAL-ORDERED MULTICASTING EXAMPLE

- Two messages ($m_1$, $m_2$) must be distributed, to two processes ($p_1$, $p_2$)
- We assume messages have correct lamport clock timestamps
- $m_1$(10, $p_1$, add \$100)

**Key point:**

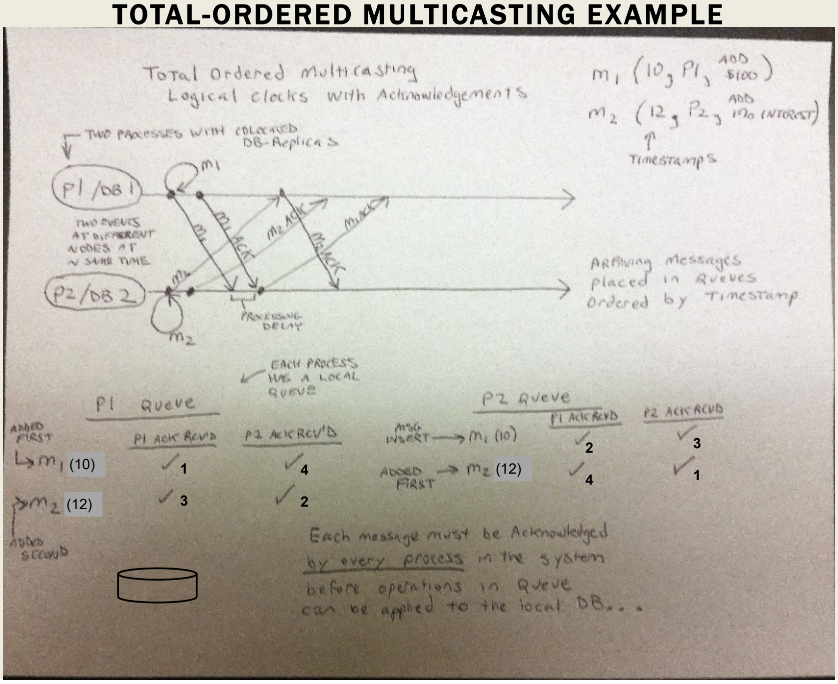
**Multicast messages are also received by the sender (*itself*)**

- Arriving messages are placed into queues ordered by the Lamport clock timestamp
- In each queue, each message must be acknowledged by every process in the system before operations can be applied to the local database



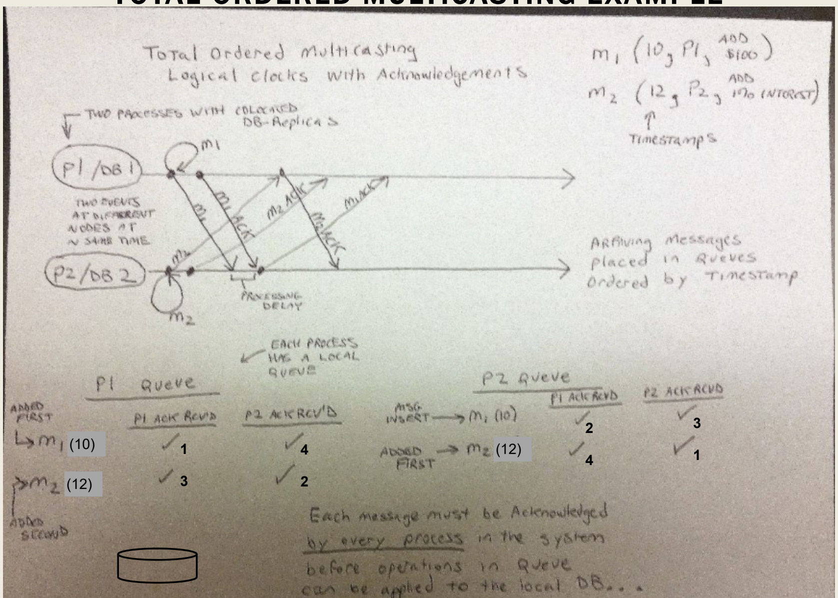
49

# TOTAL-ORDERED MULTICASTING EXAMPLE




50

## TOTAL-ORDERED MULTICASTING EXAMPLE




**What is the final account balance?**

51

## TOTAL-ORDERED MULTICASTING - 2

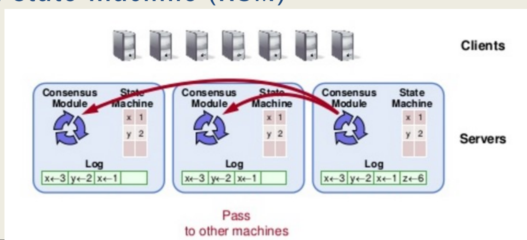
- Each message timestamped with local logical clock of sender
- **Multicast messages are also received by the sender (*itself*)**
- Assumptions:
  - Messages from same sender received in order they were sent
  - No messages are lost
- When messages arrive they are placed in local queue ordered by timestamp
- Receiver ***multicasts*** acknowledgement of message receipt to other processes
  - Time stamp of message receipt is lower the acknowledgement
- This process ***replicates*** queues across sites
- Messages delivered to application (database) only when message at the head of the queue has been acknowledged by ***every*** process in the system



52

# TOTAL-ORDERED MULTICASTING - 3

- Can be used to implement replicated state machines (RSMs)
- Concept is to replicate event queues at each node
- (1) ***Using logical clocks*** and (2) ***exchanging acknowledgement messages***, allows for events to be ***"totally"*** ordered in replicated event queues
- Events can be applied **"in order"** to each (distributed) replicated state machine (RSM)




| February 27, 2024 | TCSS558: Applied Distributed Computing [Winter 2024]<br>School of Engineering and Technology, University of Washington - Tacoma | L15.53 |
|---|---|---|

53

# OBJECTIVES – 2/27

- Questions from 2/22
- Assignment 3: Replicated Key Value Store
- Chapter 4.4 - Review / Finish
- Chapter 6: Coordination
  - Chapter 6.1: Clock Synchronization
  - Chapter 6.2: Logical Clocks
    Vector Clocks
- Class Activity – Total Ordered Multicasting (Thursday)
  - Chapter 6.3: Distributed Mutual Exclusion

| February 27, 2024 | TCSS558: Applied Distributed Computing [Winter 2024]<br>School of Engineering and Technology, University of Washington - Tacoma | L15.54 |
|---|---|---|

54

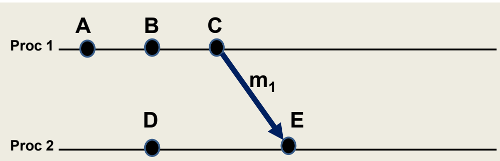
# VECTOR CLOCKS

- Lamport clocks don't help to determine causal ordering of messages
- Vector clocks capture causal histories and can be used as an alternative
- But what is causality? …



55

# WHAT IS CAUSALITY?




- Having a causal relationship between two events (A and E) indicates that event E results from the occurrence of event A.
- When one event results from another, there is a causal relationship between the two events.
- This is also referred to as ***cause and effect***.



56

# CAUSALITY - 2

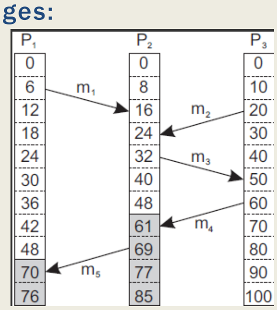
- **Disclaimer:**
- Without knowing actual information contained in messages, it is not possible to state with certainty that there is a causal relationship or perhaps a conflict
- Lamport/Vector clocks can help us suggest possible causality
- But we never know for sure…

| February 27, 2024 | TCSS558: Applied Distributed Computing [Winter 2024]<br>School of Engineering and Technology, University of Washington - Tacoma | L15.57 |
|---|---|---|

57

# CAUSALITY - 3

- Consider the messages:




- P2 receives m1, and subsequently sends m3
- **Causality:** Sending m3 ***may*** depend on what's contained in m1
- P2 receives m2, receiving m2 is ***not*** related to receiving m1
- ***Is sending m3 causally dependent on receiving m2?***

| February 27, 2024 | TCSS558: Applied Distributed Computing [Winter 2024]<br>School of Engineering and Technology, University of Washington - Tacoma | L15.58 |
|---|---|---|

58

# VECTOR CLOCKS

- Vector clocks help keep track of **causal history**
- If two local events happened at process P, then the causal history H(p2) of event p2 is {p1,p2}

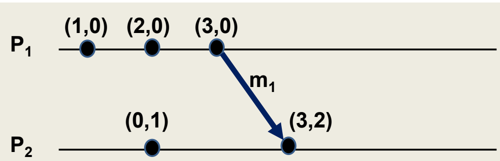
- P sends messages to Q (*event p3*)
- Q previously performed event q1
- Q records arrival of message as q2
- Causal histories merged at Q H(q2)= {p1,p2,p3,q1,q2}

- Fortunately, can simply store history of last event, as a vector clock → H(q2) = (3,2)
- Each entry corresponds to the last event at the process

February 27, 2024 | TCSS558: Applied Distributed Computing [Winter 2024] School of Engineering and Technology, University of Washington - Tacoma | L15.59

59

# VECTOR CLOCKS - 2




- Each process maintains a vector clock which
  - Captures number of events at the local process (e.g. logical clock)
  - Captures number of events at all other processes
- Causality is captured by:
  - For each event at Pi, the vector clock ($VC_i$) is incremented
  - The msg is timestamped with $VC_i$; and sending the msg is recorded as a new event at $P_i$
  - $P_j$ adjusts its $VC_j$ choosing the **max** of: the message timestamp –or- the local vector clock ($VC_j$)

February 27, 2024 | TCSS558: Applied Distributed Computing [Winter 2024] School of Engineering and Technology, University of Washington - Tacoma | L15.60

60

# VECTOR CLOCKS - 3

- Pj knows the # of events at Pi based on the timestamps of the received message
- Pj learns how many events have occurred at other processes based on timestamps in the vector
- These events ***"may be causally dependent"***
- **In other words:** they may have been necessary for the message(s) to be sent…

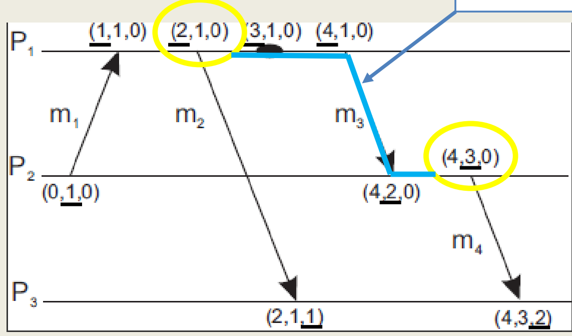
| February 27, 2024 | TCSS558: Applied Distributed Computing [Winter 2024]<br>School of Engineering and Technology, University of Washington - Tacoma | L15.61 |
|---|---|---|

61

# VECTOR CLOCKS EXAMPLE

- Local clock is underlined

CAUSALITY

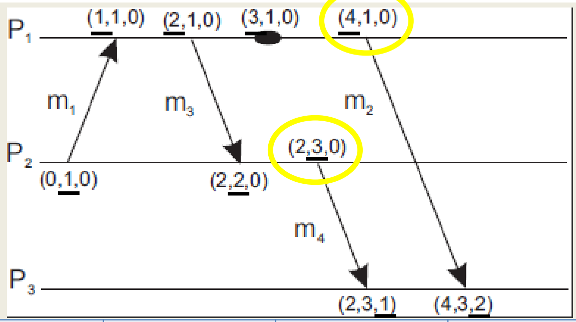



| $m_2$ | $m_4$ | $m_2 < m_4$ | $m_2 > m_4$ | Conclusion |
|---|---|---|---|---|
| (2,1,0) | (4,3,0) | Yes | No | m2 may causally precede m4 |

| February 27, 2024 | TCSS558: Applied Distributed Computing [Winter 2024]<br>School of Engineering and Technology, University of Washington - Tacoma | L15.62 |
|---|---|---|

62

# VECTOR CLOCKS EXAMPLE - 2

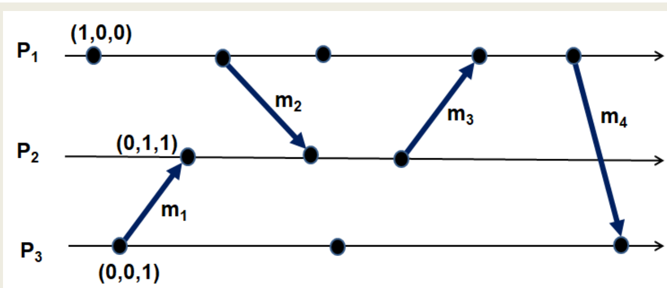



| $m_2$ | $m_4$ | $m_2 < m_4$ | $m_2 > m_4$ | Conclusion |
|---|---|---|---|---|
| (4,1,0) | (2,3,0) | No | No | m2 and m4 may conflict |

- P3 can't determine if m4 may be causally dependent on m2
- **Is m4 causally dependent on m3 ?**

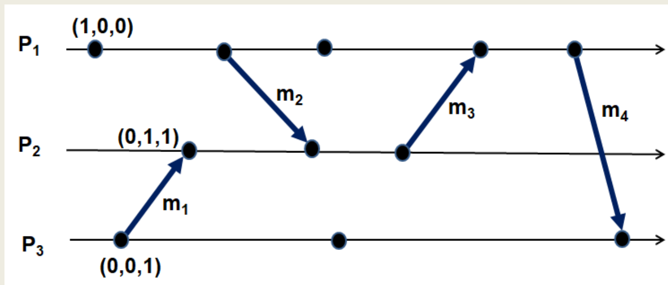


# VECTOR CLOCKS EXAMPLE - 3




- Provide a vector clock label for unlabeled events

# VECTOR CLOCKS EXAMPLE - 4

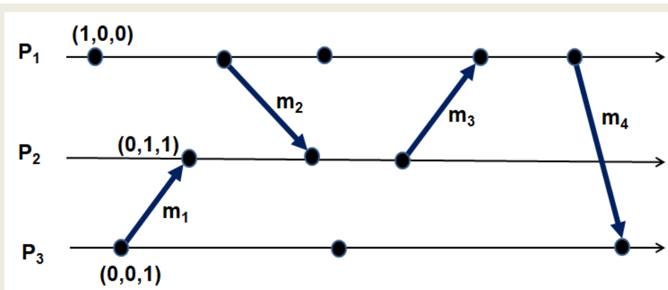



- TRUE/FALSE:
- The sending of message $m_3$ is causally dependent on the sending of message $m_1$.
- The sending of message $m_2$ is causally dependent on the sending of message $m_1$.

| February 27, 2024 | TCSS558: Applied Distributed Computing [Winter 2024]<br>School of Engineering and Technology, University of Washington - Tacoma | L15.65 |
|---|---|---|

65

# VECTOR CLOCKS EXAMPLE - 5




- TRUE/FALSE:
- $P_1$ (1,0,0) and $P_3$ (0,0,1) may be concurrent events.
- $P_2$ (0,1,1) and $P_3$ (0,0,1) may be concurrent events.
- $P_1$ (1,0,0) and $P_2$ (0,1,1) may be concurrent events.

| February 27, 2024 | TCSS558: Applied Distributed Computing [Winter 2024]<br>School of Engineering and Technology, University of Washington - Tacoma | L15.66 |
|---|---|---|

66

# OBJECTIVES – 2/27

- Questions from 2/22
- Assignment 3: Replicated Key Value Store
- Chapter 4.4 - Review / Finish
- Chapter 6: Coordination
  - Chapter 6.1: Clock Synchronization
  - Chapter 6.2: Logical Clocks
    **Vector Clocks**
- Class Activity – Total Ordered Multicasting (Thursday)
  - Chapter 6.3: Distributed Mutual Exclusion



67

# OBJECTIVES – 2/27

- Questions from 2/22
- Assignment 3: Replicated Key Value Store
- Chapter 4.4 - Review / Finish
- Chapter 6: Coordination
  - Chapter 6.1: Clock Synchronization
  - Chapter 6.2: Logical Clocks
    Vector Clocks
- **Class Activity – Total Ordered Multicasting (Thursday)**
  - Chapter 6.3: Distributed Mutual Exclusion



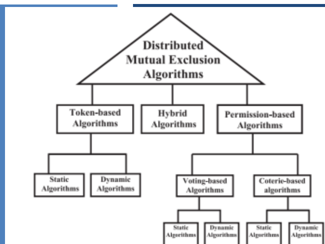
68

# OBJECTIVES – 2/27

- Questions from 2/22
- Assignment 3: Replicated Key Value Store
- Chapter 4.4 - Review / Finish
- Chapter 6: Coordination
  - Chapter 6.1: Clock Synchronization
  - Chapter 6.2: Logical Clocks
    Vector Clocks
- Class Activity – Total Ordered Multicasting (Thursday)
  - **Chapter 6.3: Distributed Mutual Exclusion**








# CH. 6.3: DISTRIBUTED MUTUAL EXCLUSION

## DISTRIBUTED MUTUAL EXCLUSION ALGORITHMS

- Coordinating access among distributed processes to a shared resource requires **Distributed Mutual Exclusion**
- **Algorithms in 6.3**
- Token-ring algorithm
- ***Permission-based algorithms:***
- Centralized algorithm
- Distributed algorithm (Ricart and Agrawala)
- Decentralized voting algorithm (Lin et al.)

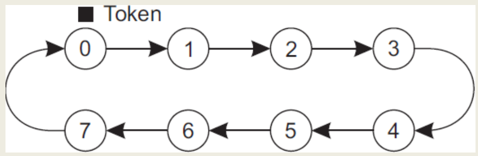


## TOKEN-BASED ALGORITHMS

- Mutual exclusion by passing a "token" between nodes
- Nodes often organized in ring
- Only one token, holder has access to shared resource
- Avoids starvation: ***everyone gets a chance to obtain lock***
- Avoids deadlock: easy to avoid

# TOKEN-RING ALGORITHM

- Construct overlay network
- Establish logical ring among nodes




- Single token circulated around the nodes of the network
- Node having token can access shared resource
- If no node accesses resource, token is constantly circulated around ring

| February 27, 2024 | TCSS558: Applied Distributed Computing [Winter 2024]<br>School of Engineering and Technology, University of Washington - Tacoma | L15.73 |
|---|---|---|

73

# TOKEN-RING CHALLENGES

1. If token is lost, token must be regenerated
   - **Problem**: may accidentally circulate multiple tokens
2. Hard to determine if token is lost
   - What is the difference between token being lost and a node holding the token (***lock***) for a long time?
3. When node crashes, circular network route is broken
   - Dead nodes can be detected by adding a receipt message for when the token passes from node-to-node
   - When no receipt is received, node assumed dead
   - Dead process can be “jumped” in the ring

| February 27, 2024 | TCSS558: Applied Distributed Computing [Winter 2024]<br>School of Engineering and Technology, University of Washington - Tacoma | L15.74 |
|---|---|---|

74

# DISTRIBUTED MUTUAL EXCLUSION ALGORITHMS - 3

- **Permission-based algorithms**
- Processes must require permission from other processes before first acquiring access to the resource
  - CONTRAST: Token-ring did not ask nodes for permission

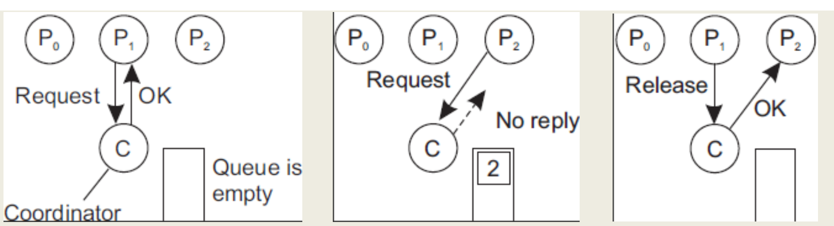
- **Centralized algorithm**
- Elect a single leader node to coordinate access to shared resource(s)
- Manage mutual exclusion on a distributed system similar to how it mutual exclusion is managed for a single system
- Nodes must all interact with leader to obtain ***"the lock"***



75

# CENTRALIZED MUTUAL EXCLUSION

Permission granted from coordinator \/ No response from coordinator




**$P_1$ executes** **$P_2$ blocks** **$P_1$ finishes; $P_2$ executes**

- When resource not available, coordinator can block the requesting process, or respond with a reject message
- P2 must ***poll*** the coordinator if it responds with reject otherwise can wait if simply blocked
- Requests granted permission fairly using FIFO queue
- Just three messages: (request, grant (OK), release)



76

# CENTRALIZED MUTUAL EXCLUSION - 2

- **Issues**
- Coordinator is a single point of failure
- Processes can't distinguish dead coordinator from ***"blocking"*** when resource is unavailable
  - No difference between CRASH and Block (*for a long time*)
- Large systems, coordinator becomes performance bottleneck
  - Scalability: Performance does not scale

- **Benefits**
- Simplicity:
  Easy to implement compared to distributed alternatives



# DISTRIBUTED ALGORITHM

- Ricart and Agrawala [1981], use total ordering of all events
  - Leverages Lamport logical clocks

- Package up resource request message (AKA Lock Request)
- Send to all nodes
- Include:
  - Name of resource
  - Process number
  - Current (logical) time

- Assume messages are sent reliably
  - No messages are lost

# DISTRIBUTED ALGORITHM - 2

- When each node receives a request message they will:
1. Say OK (***if the node doesn't need the resource***)
2. Make **no reply**, queue request (***node is using the resource)***
3. ***If node is also waiting to access the resource:*** perform a timestamp comparison -
   1. Send OK if requester has lower logical clock value
   2. Make **no reply** if requester has higher logical clock value
- Nodes sit back and wait for all nodes to grant permission

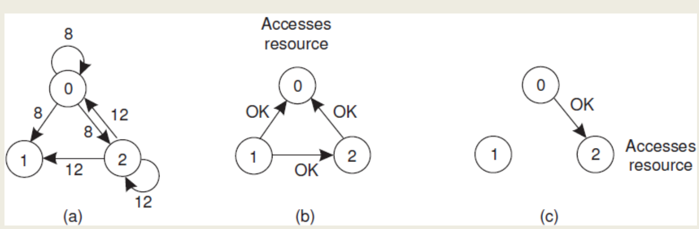
- Requirement: every node must know the entire membership list of the distributed system



79

# DISTRIBUTED ALGORITHM - 3

- Node 0 and Node 2 simultaneously request access to **resource**
- Node 0's time stamp is lower (8) than Node 2 (12)
- Node 1 and Node 2 grant Node 0 access
- Node 1 is not interested in the resource, it OKs both requests




- **In case of conflict, lowest timestamp wins!**
  - Node 2 rejects its own request (1@) in favor of node 0 (8)



80

# CHALLENGES WITH DISTRIBUTED ALGORITHM

- **Problem:** Algorithm has N points of failure !
- Where N = Number of Nodes in the system
- **No Reply Problem:** When node is accessing the resource, it does not respond
  - Lack of response can be confused with **failure**
  - ***Possible Solution:*** When node receives request for resource it is accessing, always send a reply either granting or denying permission (ACK)
  - Enables requester to determine when nodes have died





# CHALLENGES WITH DISTRIBUTED ALGORITHM - 2

- **Problem**: Multicast communication required –or- each node must maintain full group membership
  - Track nodes entering, leaving, crashing…
- **Problem**: Every process is involved in reaching an agreement to grant access to a shared resource
  - This approach ***may not scale*** on resource-constrained systems
- Solution: Can relax total agreement requirement and proceed when a **simple majority** of nodes grant permission
  - *Presumably any one node locking the resource prevents agreement*
  - *If one node gets majority of acknowledges no other can*
  - *Requires every node to know size of system (# of nodes)*
- Distributed algorithm for mutual exclusion works best for:
  - Small groups of processes
  - When memberships rarely change

## DECENTRALIZED ALGORITHM

- Lin et al. [2004], decentralized voting algorithm
- Resource is replicated N times
- Each replica has its own coordinator ...(N coordinators)
- Accessing resource requires majority vote:
  total votes (m) > N/2 coordinators
- **Assumption #1:** When coordinator does not give permission to access a resource (because it is busy) it will inform the requester



83

## DECENTRALIZED ALGORITHM - 2

- **Assumption #2:** When a coordinator crashes, it recovers quickly, but will have forgotten votes before the crash.
- Approach assumes coordinators reset **arbitrarily** at any time
- **Risk**: on crash, coordinator forgets it previously granted permission to the shared resource, and on recovery it errantly grants permission again
- **The Hope**: if coordinator crashes, *upon recovery*, ***the node granted access to the resource has already finished before the restored coordinator grants access again . . .***



84

# DECENTRALIZED ALGORITHM - 3

- With 99.167% coordinator availability (30 sec downtime/hour) chance of violating correctness **is so low** it can be neglected in comparison to other types of failure
- Leverages fact that a new node must obtain a majority vote to access resource, ***which requires time***

| N | m | p | Violation |
|---|---|---|---|
| 8 | 5 | 3 sec/hour | $< 10^{-15}$ |
| 8 | 6 | 3 sec/hour | $< 10^{-18}$ |
| 16 | 9 | 3 sec/hour | $< 10^{-27}$ |
| 16 | 12 | 3 sec/hour | $< 10^{-36}$ |
| 32 | 17 | 3 sec/hour | $< 10^{-52}$ |
| 32 | 24 | 3 sec/hour | $< 10^{-73}$ |

| N | m | p | Violation |
|---|---|---|---|
| 8 | 5 | 30 sec/hour | $< 10^{-10}$ |
| 8 | 6 | 30 sec/hour | $< 10^{-11}$ |
| 16 | 9 | 30 sec/hour | $< 10^{-18}$ |
| 16 | 12 | 30 sec/hour | $< 10^{-24}$ |
| 32 | 17 | 30 sec/hour | $< 10^{-35}$ |
| 32 | 24 | 30 sec/hour | $< 10^{-49}$ |

N = number of resource replicas, m = required “majority” vote
p=seconds per hour coordinator is offline



# DECENTRALIZED ALGORITHM - 4

- **Back-off Polling Approach for *permission-denied:***
- If permission to access a resource is denied via majority vote, process can poll to gain access again with a ***random*** delay (***known as back-off***)
- Node waits for a random amount, retries…
- If too many nodes compete to gain access to a resource, majority vote can lead to low resource utilization
  - ***No one can achieve majority vote to obtain access to the shared resource***
  - *Mimics elections where with too many candidates, where no one candidate can get >50% of the total vote*
- Problem Solution detailed in [Lin et al. 2014]

## DISTRIBUTED MUTUAL EXCLUSION ALGORITHMS REVIEW

- Which algorithm offers the best scalability to support distributed mutual exclusion in a large distributed system?
- (A) Token-ring algorithm
- (B) Centralized algorithm
- (C) Distributed algorithm
- (D) Decentralized voting algorithm



## DISTRIBUTED MUTUAL EXCLUSION ALGORITHMS REVIEW - 2

- Which algorithm(s) involve blocking when a resource is not available?
- (A) Token-ring algorithm
- (B) Centralized algorithm
- (C) Distributed algorithm
- (D) Decentralized voting algorithm

## DISTRIBUTED MUTUAL EXCLUSION ALGORITHMS REVIEW - 3

- Which algorithm(s) involve arriving at a consensus to determine whether a node should be granted access to a resource?
- (A) Token-ring algorithm
- (B) Centralized algorithm
- (C) Distributed algorithm
- (D) Decentralized voting algorithm



## DISTRIBUTED MUTUAL EXCLUSION ALGORITHMS REVIEW - 4

- Which algorithm(s) have N points of failure, where N = Number of Nodes in the system?
- (A) Token-ring algorithm
- (B) Centralized algorithm
- (C) Distributed algorithm
- (D) Decentralized voting algorithm

# QUESTIONS



91