Today's Objectives

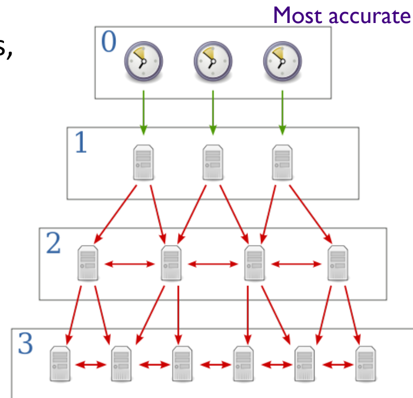
- Wrap up Timing
- Coordination
- Consensus

Review

- What is NTP?
  - What is the motivation?
  - Describe its design
  - What are the benefits?

Review: NTP Clock Strata

- Stratum 0: atomic clocks, GPS clocks, radio clocks w/ UTC
- Stratum 1: Time servers (primary), attached directly to Stratum 0 devices
- Stratum 2: Send requests to one or more Stratum 1 time servers
- Stratum 3: Send requests to one or more Stratum 2 computers
- And so on…
- Up to 256(!) strata levels supported in current version of NTP

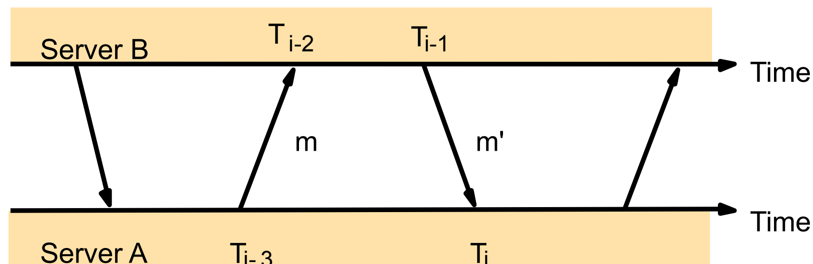


Lowest leaf:
users' workstations
Reconfigurable in
response to failures

https://en.wikipedia.org/wiki/Network_Time_Protocol#/media/File:Network_Time_Protocol_servers_and_clients.svg

Synchronizing Servers

- All messages sent using UDP
- Each message bears timestamps of recent events:
  - Local times of *Send* and *Receive* of previous message
  - Local times of *Send* of current message
- Recipient notes the time of receipt T_i
  - Have T_{i-3}, T_{i-2}, T_{i-1}, T_i

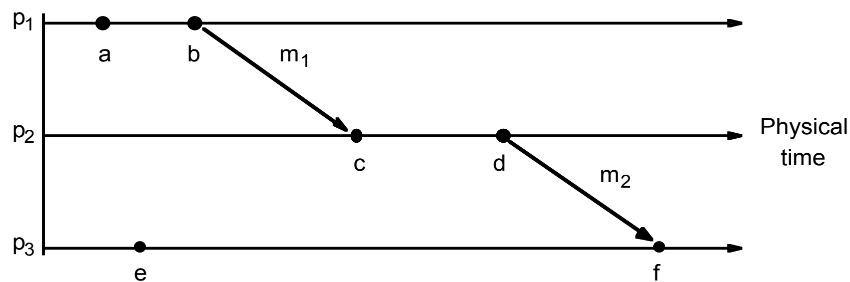


Review: Logical Clocks

- What is the motivation for logical clocks?

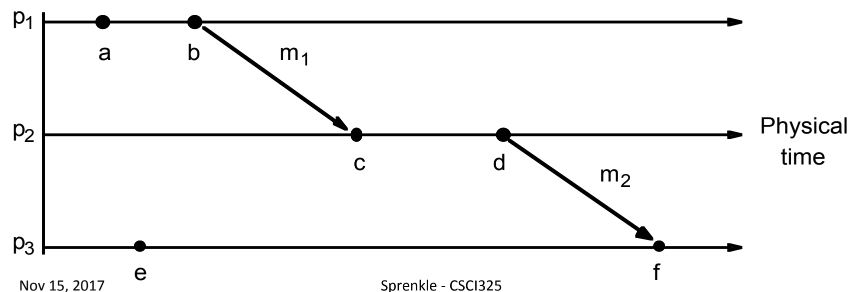
Logical Time and Logical Clocks

- Instead of synchronizing clocks, event ordering can be used
- Rules:
  1. If two events occurred at the same process p_i (i = 1, 2, … N) then they occurred in the order observed by p_i, that is $\rightarrow_i$
  2. When a message *m* is sent between two processes, *send(m)* happened before *receive(m)*
  3. The happened-before relation is transitive



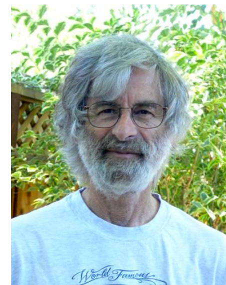
Happened Before Relation

- What do we know about events a, b, c, d, f?
  - Rule 1: a → b (at p_1), c → d (at p_2)
  - Rule 2: b → c (by m_1), d → f (by of m_2)
  - Rule 3: a → b → c → d → f = a → f
- What do we know about a and e?
  - No relation → they are concurrent: a || e



Lamport's Logical Clocks

- A logical clock is a monotonically increasing software counter
  - Need not relate to a physical clock

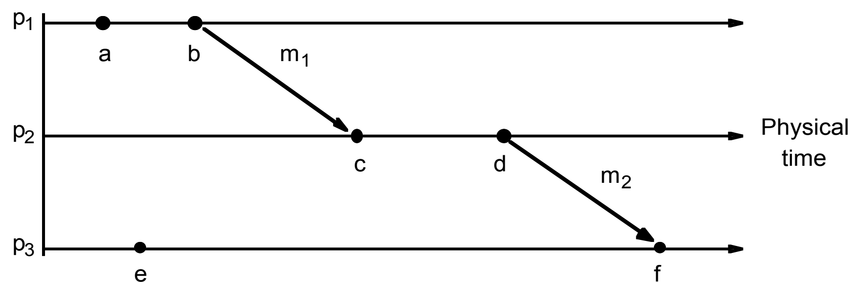


Leslie Lamport

LaTeX

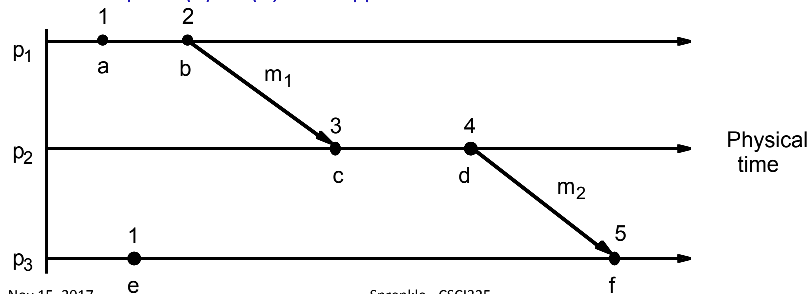
Lamport's Logical Clocks

- Each process p_i has a logical clock, L_i
  - Can be used to apply *logical timestamps* to events using rules:
    - LC1: L_i is incremented by 1 before each event at process p_i, $L_i = L_i + 1$
    - LC2:
      a) when process p_i sends message m, it piggybacks on m the value $t = L_i$
      b) when p_j receives (m,t) it sets $L_j := max(L_j, t)$ and applies LC1 before timestamping the event *receive* (m)



Lamport's Logical Clocks

- Each of p_1, p_2, p_3 has its logical clock initialized to zero
- The clock values on events are those immediately *after* the event
  - e.g., 1 for a, 2 for b.
- For m_1, t = 2 is piggybacked and c gets L_2 = max(0,2)+1 = 3
- Note that e → e' implies L(e) < L(e')
- Does L(e) < L(e') imply e → e' ?
  - No! The converse is not true: L(e) < L(e') does not imply e → e'
  - Example: L(e) < L(b) but b || e

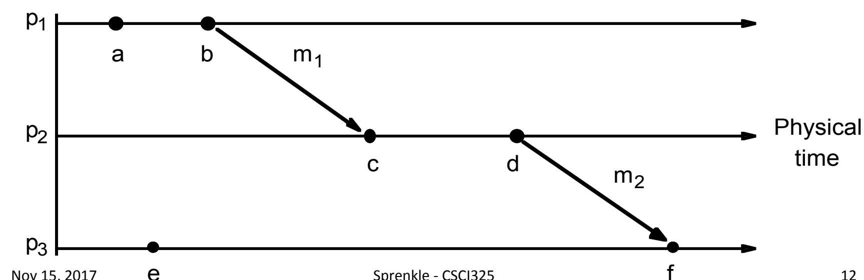


Lamport Clocks → Vector Clocks

- Limitation of Lamport clocks:
  - L(e) < L(e') does not imply e happened before e'
  - If L(e) < L(e'), we want to know *for sure* that e happened before e'
- How can we overcome the limitation?
- Solution: **Vector clocks**
  - ***Vector*** timestamps (rather than a single number) are used to timestamp local events
  - Vector clock $V_i[i]$ is the number of events that p_i has timestamped
  - $V_i[j]$ ($j \neq i$) is the number of events at p_j that p_i has been affected by
- Vector clocks are used in many schemes for replication of data to ensure consistency

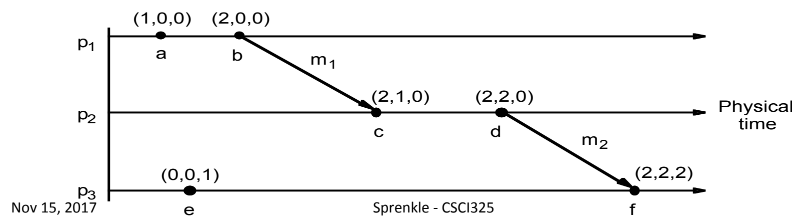
Vector Clocks

- Vector clock V_i at process p_i is an array of N integers
- Rules for determining vector clocks:
  - VC1: Initially $V_i[j] = 0$ for $i, j = 1, 2, \ldots N$
  - VC2: Before p_i timestamps an event, it sets $V_i[i] = V_i[i] + 1$
  - VC3: p_i piggybacks $t = V_i$ on every message it sends
  - VC4: **Merge**: When p_i receives (m,t) it sets $V_i[j] := \max(V_i[j], t[j])$ $j = 1, 2, \ldots N$



Vector Clocks

- At p_1: a(1,0,0), b(2,0,0), piggyback (2,0,0) on m_1
- At p_2: On receipt of m_1 get max ((0,0,0), (2,0,0)) = (2,0,0), and add 1 to own element in clock = (2,1,0) for event c
- At p_3: On receipt of m_2 get max ((0,0,1), (2,2,0)) = (2,2,1) and add 1 to own element in clock
- Vector timestamp operations: =, <=, max, etc.
  - Compare elements pairwise
- Note that e → e' still implies L(e) < L(e')
- And now the converse is also true (L(e) < L(e') implies e → e')
- Can you see a pair of parallel events?
  - c || e because neither V(c) <= V(e) nor V(e) <= V(c)



Summary: Time and Clocks in Distributed Systems

- Accurate timekeeping is important for distributed systems
- Algorithms (e.g., Cristian's and NTP) synchronize clocks in spite of their drift and the variability of message delays
- For ordering an arbitrary pair of events at different computers, clock synchronization is not always practical
- The *happened-before relation* is a partial order on events that reflects a flow of information between them
- **Lamport clocks** are counters that are updated according to happened-before relationship between events
- **Vector clocks** are an improvement on Lamport clocks
  - By comparing vector timestamps, can tell whether two events are ordered by happened-before or are concurrent
  - Applied in schemes for replication of data, e.g. Gossip, Coda

COORDINATION

Coordination

- Distributed processes often need to coordinate their activities
- If the processes share a resource or collection of resources, then mutual exclusion is required to ensure consistency
  - Often called the ***critical section*** problem
  - Discussed in detail in OS courses
- In this class, we need **distributed** mutual exclusion
  - Mutual exclusion based solely on message passing

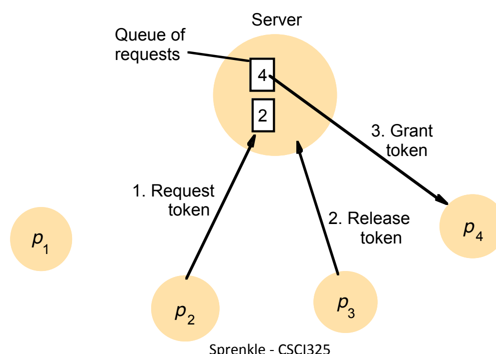
Mutual Exclusion Algorithms

- Assumptions
  - N processes share a resource in a single critical section
  - Asynchronous systems
  - Processes do not fail
  - Message delivery is reliable
- Requirements
  - **Safety**: At most one process may execute in critical section (CS) at a time
  - **Liveness**: Requests to enter and exit CS eventually succeed
  - **Happens-before Ordering**: If one request to enter CS happened-before another, entry is granted in order

Central Server Approach

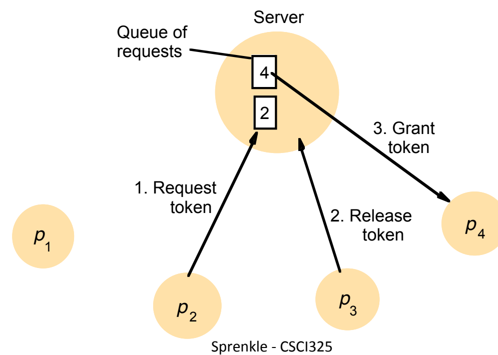
- All processes contact central server to obtain permission to enter critical section (CS)

Pros and Cons?



Central Server Approach

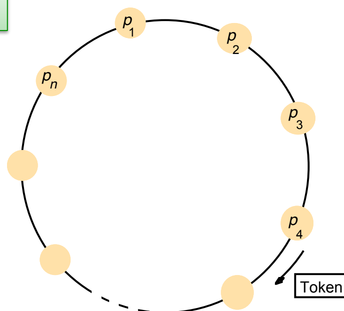
- All processes contact central server to obtain permission to enter critical section (CS)
- Pros: Simple to implement
- Cons: Can be slow (time to transmit release and grant messages); central server is bottleneck



Ring-Based Approach

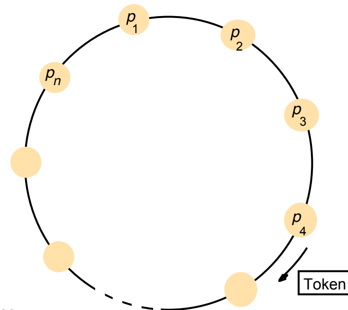
- Arrange processes in logical ring
- Each process has communication channel to the next process
- Pass "token" around ring; token grants access to CS

Pros and Cons?



Ring-Based Approach

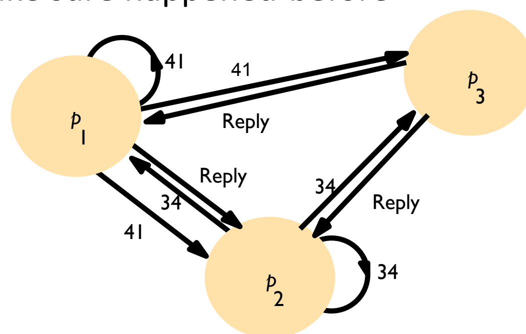
- Arrange processes in logical ring
- Each process has communication channel to the next process
- Pass "token" around ring; token grants access to CS
- Pros: Simple, no central bottleneck
- Cons: Potentially large delay; wastes bandwidth



Multicast & Logical Clocks

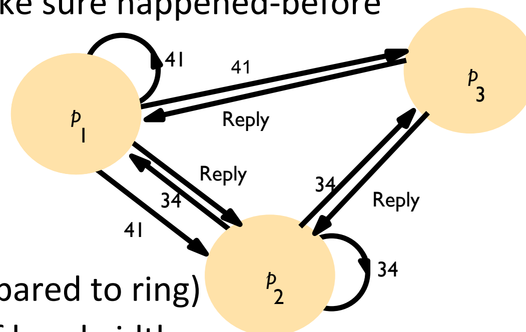
- Ricart and Agrawala developed approach based on multicast and Lamport clocks
- Multicast request for access to other processes; wait for reply
- Logical timestamps make sure happened-before requirement is met

Pros and Cons?



Multicast & Logical Clocks

- Ricart and Agrawala developed approach based on multicast and Lamport clocks
- Multicast request for access to other processes; wait for reply
- Logical timestamps make sure happened-before requirement is met



- Pros: Short delay (compared to ring)
- Cons: Consumes lots of bandwidth

Voting Algorithm

- Not necessary for all processes to grant access, only need subset of all processes
  - Each process maintains a "voting set"
  - All voting sets are the same size
- Make sure subsets used by any two processes overlap
  - For all voting sets, $V_i \cap V_j \neq \emptyset$
- Pros: Requires less bandwidth than previous approach
- Cons: determining optimal voting sets; can cause deadlock!

Questions

- What about fault tolerance?

- What happens when messages are lost?
- What happens when a process crashes?

CONSENSUS

Agreement

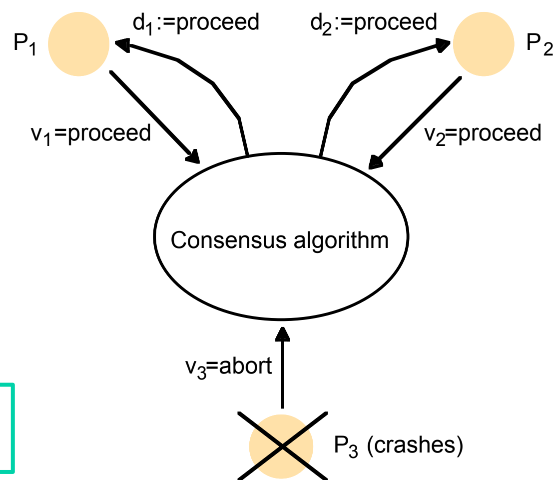
Goal: get processes to agree on some value after one or more processes propose that value

- ...even in the presence of faults!
- Often referred to as the ***consensus problem***

Consensus

- Every process begins in an undecided state and proposes a value
- Processes communicate, deciding which value to accept
  - One option: majority rules
- Requirements:
  - **Termination** - Eventually each process sets its decision variable
  - **Agreement** - The decision value of each process is the same
  - **Integrity** - If the correct processes all proposed the same value, then any correct process in decided state has chosen that value

Consensus



v = value
d = decision

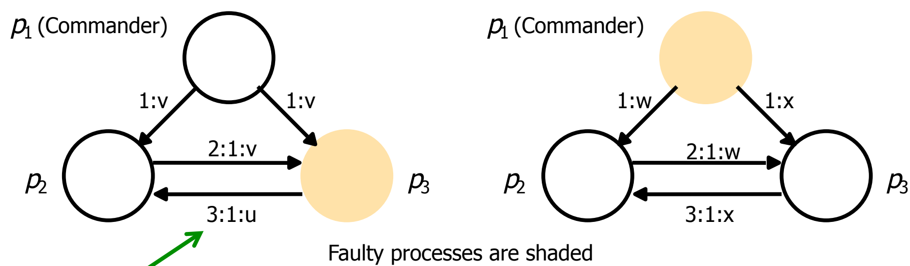
Byzantine Generals Problem

- Problem initially proposed by Lamport in 1982
- Three or more generals (N) agree to attack or retreat
- Commander issues the order
- Others (N-1) must decide to attack or retreat
  - Slightly different than normal consensus since there is a "distinguished process" deciding initial value
- One or more general may be "treacherous" or faulty (f)
  - He lies! He says "attack" to one general and "retreat" to another
  - Why lie? Think about security protocols
- How does each general decide what to do?
- Assume a synchronous system

Byzantine Generals Requirements

- Termination
  - Each "correct" process must eventually make a decision
- Agreement
  - The decision value of all correct processes must be the same
- Integrity
  - If the commander isn't faulty (not always true!), the other correct processes should decide on commander's value (and follow it)

Three Byzantine Generals



"3 says 1 says u"

The goal is for p_2 to determine that p_1 says v. But p_2 doesn't have enough info!

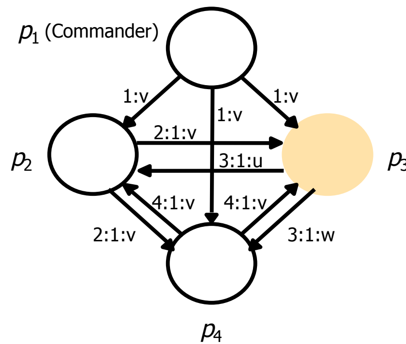
p_2 once again has conflicting info. Can't distinguish between faulty p_3 and faulty commander!

Since we can't distinguish between these two scenarios, no solution exists!

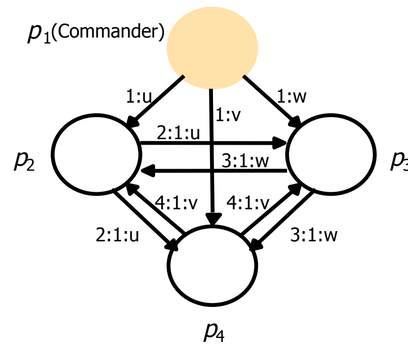
Byzantine Generals

- No solution exists if N ≤ 3f, where *f* is the number of treacherous (faulty) generals
- But if N ≥ 3f + 1, a solution exists!
- Consider N=4 generals, f=1
  - 3f + 1 = 4 ≥ N
- No solution exists in asynchronous systems for all N and f

Four Byzantine Generals



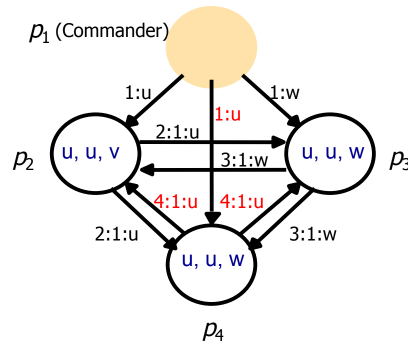
p2 and p4 should correctly determine that "1 says v." Using simple "majority rules" consensus, this works!



p2, p3, and p4 all receive u, v, w. Thus they know that the commander is faulty, and reach "no action" consensus.

- Within two rounds, non-faulty generals reach consensus
  - which may mean "take no action"

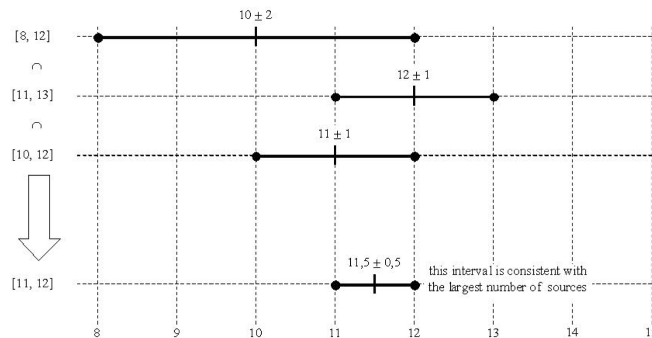
Four Byzantine Generals



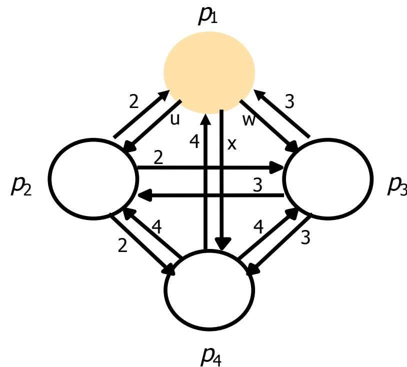
- What now?
  - They'd all pick u!
  - But this commander isn't really truly faulty
    - Faulty processes ALWAYS lie and don't propose a majority of anything

Marzullo's Algorithm

- NTP servers filter pairs <o_i, d_i>, estimating reliability from variation, allowing selection of "good" peers
- NTP servers use a variation of an algorithm developed by Keith Marzullo to choose a time value given a bunch of varying samples



Another Variation of Byzantine Generals



Round 1: Send value to all other processes
1 Got (1, 2, 3, 4)
2 Got (u, 2, 3, 4)
3 Got (w, 2, 3, 4)
4 Got (x, 2, 3, 4)

Round 2: Exchange vectors

2 Got	3 Got	4 Got
(a, b, c, d)	(e, f, g, h)	(i, j, k, l)
(w, 2, 3, 4)	(u, 2, 3, 4)	(u, 2, 3, 4)
(x, 2, 3, 4)	(x, 2, 3, 4)	(w, 2, 3, 4)

- Byzantine Agreement
  - Here p2, p3, and p4 reach an agreement on their respective values, which is all that matters since p1 is faulty

Looking Ahead

- Inverted Index Team Evaluation
- Exam - Due Friday
- Final Project Proposal
  - One-page paper
  - Due Monday after Thanksgiving Break
  - Check in with me beforehand if you're not sure if your project will fit in.