

Distributed Operating Systems

- Logical clocks
 - Absence of global state
 - Absence of global clock
- Mutual exclusion



Happened-before relation

Happened before: $a \rightarrow b$ (a happened before b).

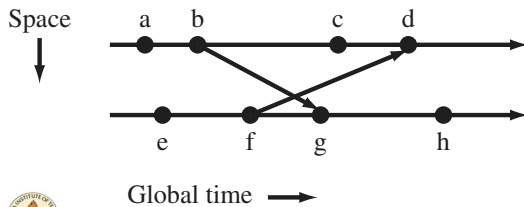
- $a \rightarrow b$ if a and b are in the same process, and a happened before b .
- $a \rightarrow b$ if a is a message send, and b is the corresponding reception.
- if $a \rightarrow b$ and $b \rightarrow c$ then $a \rightarrow c$ (transitivity).

Concurrency: if $\neg(a \rightarrow b)$ and $\neg(b \rightarrow a)$ then $a || b$.



Space-time diagrams

- Each line is a process
- Each process has a sequence of events, listed in global time



Logical clocks (Lamport, 1978)

Each process P_i has a clock C_i that assigns a time $C_i(a)$ to any event in P_i .

Properties:

- C1 for any two events a, b in P_i , if $a \rightarrow b$, then $C(a) < C(b)$.
- C2 if a is a message from P_i to P_j and b is the reception, then $C_i(a) < C_j(b)$.



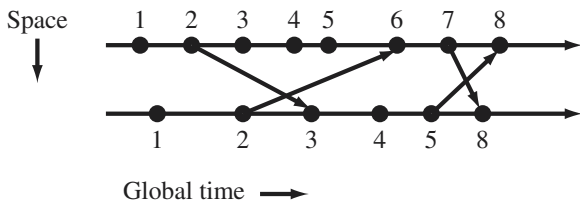
Implementing logical clocks

IR1 Increment between every two events in a process: if $a \rightarrow b$ then $C_i(b) = C_i(a) + d$ for some $d > 0$.

IR2 Send a logical timestamp in each message. When a message m arrives take the max of the local time and the time in the message: $C_j = \max(C_j, t_m + d)$, for $d > 0$.

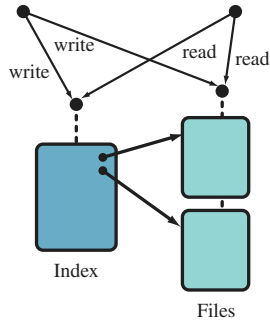


Example



Distributed Mutual Exclusion

- No shared memory, but we still need mutual exclusion to protect shared resources (like files)



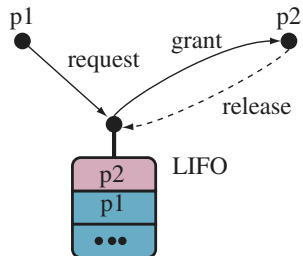
Mutual exclusion requirements

- Freedom from deadlocks
- Freedom from starvation: a site should not wait indefinitely while other sites repeatedly access the CS
- Strict fairness: requests are served in the (logical) order in which they arrive
- Fault tolerance: an algorithm should be able to detect and recover from failures



Server-based mutual exclusion

- Server maintains a queue of CS requests
- Grants them in order



Central server

- Advantages
 - High performance (3 messages per CS execution)
- Disadvantages
 - Central point of failure
 - Not scalable
 - Low throughput (server does not grant the CS until it gets a release)



Lamport's algorithm

- Each site S_i keeps a *request queue* of requests ordered by logical timestamp
- Requesting the critical section
 - Send a $REQUEST(C_i, i)$ to each other site, and place in the request queue
 - When S_j receives the $REQUEST(C_i, i)$ it returns a timestamped $REPLY$ to S_i and places the $REQUEST$ in the request queue

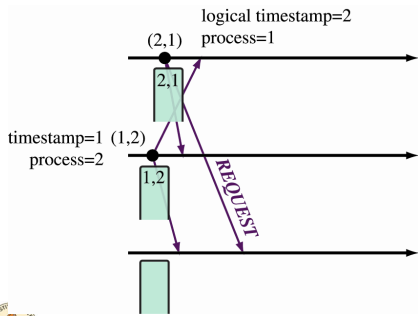


Executing/releasing the CS

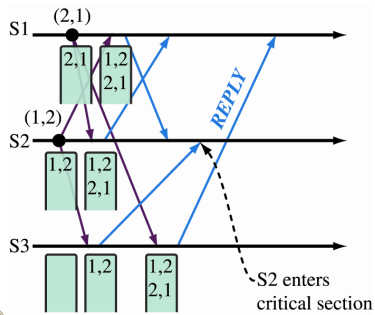
- Conditions for entering CS
 - L1: S_i has received a message with a timestamp larger than (C_i, i) from all other sites
 - L2: S_i 's request is at the top of the request queue
- Releasing the CS
 - Remove (C_i, i) from the request queue, and send a $RELEASE$ to all sites in the request set
 - When S_j receives $RELEASE$, it removes the request from the request queue



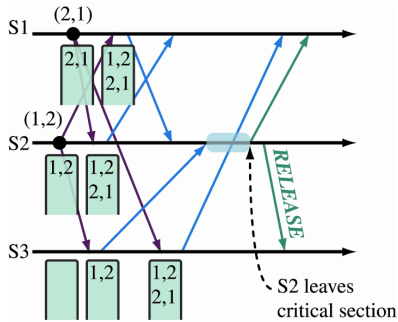
Step 1



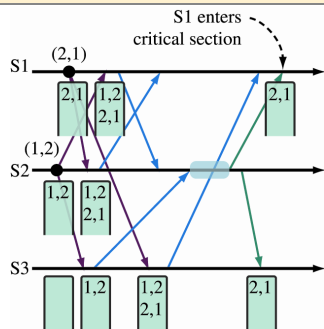
Step 2



Step 3



Step 4



Lamport's algorithm: correctness

- Assume two processes S1, S2 in CS simultaneously
 - [L1, L2] hold in both processes (both have received messages with larger timestamps from all others, their requests are at the top of their queues)
 - WLOG, assume S1's request is earlier
 - So, S1's request is in S2's queue, but S2's request is at the top of the queue—contradiction.



Lamport evaluation

- Deadlock: processes do not wait forever
- Fairness: CS requests are granted in order of logical clock, which is fair
- Fault tolerance
 - If a process fails before a request, fine
 - If a process fails after a few requests...
 - One method
 - Use a probabilistic failure detector (pings)
 - If a process times out, remove it from all queues
 - If a process fails in critical section...
- Performance
 - $3(N - 1)$ messages per CS execution
 - Sync delay is T (average message latency)

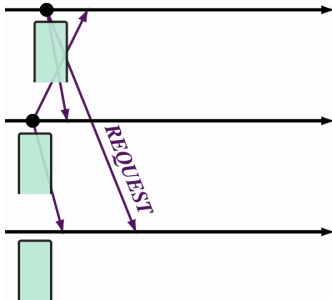


Ricart-Agrawala

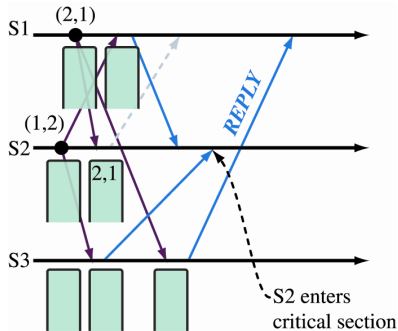
- An optimization of Lamport
- Request when S_i wants to enter
 - Broadcast timestamped REQUEST message
 - When S_j receives, send a REPLY iff:
 - S_j is not requesting or executing CS
 - Or if S_j requesting, but S_i 's timestamp is smaller
- Entering CS for S_i
 - Enter CS on receiving REPLY from everyone
- Releasing CS for S_i
 - On exiting, send REPLY for all deferred requests



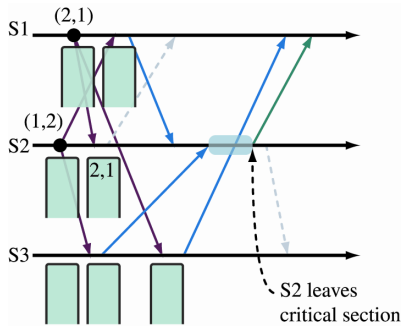
Ricart-Agrawala, step 1



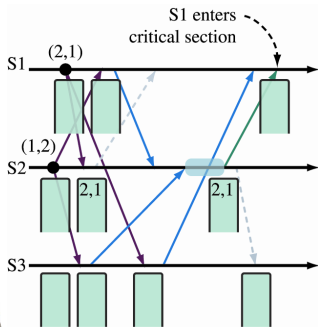
Ricart-Agrawala, step 2



Ricart-Agrawala, step 3



Ricart-Agrawala, step 4



Token-based algorithms

- A unique token is shared among all sites
 - This means a sequence number is used instead of a timestamp
 - A site increments its sequence number every time it requests the token
- This means
 - Issues of liveness (deadlock-freedom, starvation) are more interesting



Suzuki-Kasami (1985)

- When requesting the CS
 - If the site does not have the token, broadcast a REQUEST
 - If it has the token it can enter repeatedly
- When receiving a REQUEST
 - Send the requestor the token after leaving CS



Problems

- Liveness: a process can starve the others...



Suzuki-Kasami: algorithm

- Request
 - If S_i does not have the token, increment sequence number $Rn[i][i]$, and send REQUEST($i, Rn[i]$) to other sites
 - When S_j receives REQUEST(i, sn), set $RN[j][i]$ to $\max(sn, RN[j][i])$; if it has the IDLE token, send it to S_i if $RN[j][i] = LN[j] + 1$
- Entering
 - Site S_i enters when it gets the token
- Releasing (from S_i)
 - $LN[i] < Rn[i][i]$
 - For each S_j not in the token queue, append ID to queue if $RN[i][j] = LN[j] + 1$
 - If token queue is nonempty after update, delete top ID and send to site indicated by ID



Suzuki-Kasami: evaluation

- Performance
 - Either O or N messages per CS execution
 - Sync delay is either O or T



Summary

- It has been fun!
 - *It is a lot of work...*
 - *"Systems" development can be hard*
 - *But you can do a lot of cool things!*
 - *Systems affect nearly everyone*
- Can we make CS systems easy?
 - *Yes, but we also have to understand PL*
 - *More on this next term...*