14-736: DISTRIBUTED SYSTEMS

LECTURE 6 * CONCURRENCY CONTROL * SPRING 2019 (KESDEN)

CLASSICAL CONCURRENCY CONTROL

- Semaphores, Mutexes, ConditionVariables, Monitors, etc.
- Requirement: Efficient, shared, consistent, memory
- Do we have that in distributed systems?
- Why or why not?

DISTRIBUTED MUTUAL EXCLUSION: REQUIREMENTS

- Correct: At most one user
- Progress: If the resource is available and a participant wants it, it can be put to use.
- Some level of fairness
  - FCFS vs. tightly bounded wait vs eventual
  - Must assume processes hold lock for bounded or non-infinite time.
- Fairness guarantees implies deadlock free

DISTRIBUTED MUTUAL EXCLUSION: ADDITIONAL GOALS

- Lets participants join and leave
- Tolerates failure
- Low message overhead
- Tolerates communications anomalies
  - Loss, Delay, Reordering
  - Obviously can't work with total failure

CENTRAL APPROACH

- Single coordination server maintains a local queue
  - Participant: REQUEST message
  - Coordinator: GRANT message
  - Participant: RELEASE message
  - 6 Messages per access, regardless of number of hosts.
    - Note: All messages are ACKed, so 2x the 3 messages
- Simple, effective

CENTRAL APPROACH

- Single point of failure
- Can use primary-backup arrangement, but it isn't necessarily easy.
  - Example: backup gets all requests, but only primary replies. backup takes over if primary fails heartbeat.
  - But, what happens if there is a partitioning?
- What happens if server reboots?

LAMPORT MUTUAL EXCLUSION

- Big Idea
  - Maintain a distributed queue, with a copy at each participant
  - When done with critical section send RELEASE messages to each and every participant
    - Each participant pops queue and new head gets to go
    - New head sees for itself that it is at the head of the queue
- Trick:
  - How to keep the queues in sync given that the messages can cross and arrive at different hosts in different orders?

LAMPORT MUTUAL EXCLUSION

- Assumption: Messages between pairs of hosts are delivered in order
  - Such as via TCP
- Messages bear Lamport time stamps
- Queue is ordered by Lamport time stamps
  - Total ordering, such as by hostID
- Requestor can't enter critical section until:
  - It's own request is at the head of its own queue
  - It's request is acknowledged by each and every host.
- Key point: The ACK will have a timestamp from the sender greater than the time at which the request was sent.
  - **Since requests are in-order between pairs of hosts, this means that the requestor has seen any prior request from the ACKing host, so its queue is correctly ordered in this respect.**

LAMPORT PERFORMANCE

- 3(N-1) messages (Request, ACK, Release)
  - No need to send to self, hence "minus 1"
- Robustness?
  - What happens if any 1 of N hosts fail? Ouch!
- Communication problems?
  - Missing? Usual: No release? Ouch. No Request? Ouch. No reply? Ouch
  - Reordered? Okay, except if within sender-receiver pair
  - Upshot: Better rely upon TCP, etc.

RICARTI AND AGRAWALA

- In may ways inspired by Lamport
  - Key observation: Reply and Grant can be combined by delaying reply until it is okay to use the critical section
  - Reply effectively means "Okay with me"

RICARTI AND AGRAWALA

Requestor (Request)

- Send REQUEST to each and every participant

Participant

- If in CS, enqueue request
- If not in CS
  - and don't want in, reply OK
  - and want into the CS, **and** the requestor's time is lower, reply OK (messages crossed, requestor was first)
  - and want into the CS, **and** the requestor's time is greater, enqueue request (messages crossed, participant was first)
- **Requestor (Release)**
  - On exit from CS, reply OK to everyone on queue (and dequeue each)
- **Requestor (When to enter critical section?**
  - Once received OK from everyone, enter CS

RICARTI AND AGRAWALA PERFORMANCE

- 2(N-1) messages) = (N-1)(REQUEST + RELEASE) + OK
  - No need to send to self, hence "minus 1"
- Robustness?
  - What happens if any 1 of N hosts fail? Ouch!
- Communication problems?
  - Missing? Usual: No release? Ouch. No Request? Ouch. No reply? Ouch
  - Reordered? OK
  - Upshot: Better rely upon TCP, etc.

MAJORITY VOTING

When entry into the critical section is desired:

- Ask permission from all other participants via a multicast, broadcast, or collection of individual messages
- Wait until more than 50% respond "OK"
- Enter the critical section

When a request from another participant to enter the critical section is received:

- If you haven't already voted, vote "OK."
- Otherwise enqueue the request.

When a participant exits the critical section:

- It sends RELEASE to those participants that voted for it.

When a participant receives RELEASE from the elected host:

- It dequeues the next request (if any) and votes for it with an "OK."

MAJORITY VOTING: TIES (UT-OH!)

- Imagine M concurrent requests, each getting exactly N/M votes.
  - We're stuck!
- To get unstuck, we use Lamport ordering w/hostID tie-breaking and favor earlier requests
- If a host gets an earlier request after voting for a later one, it asks for its vote back (INQUIRE)
  - If the host to which it gave its vote is in the critical section
    - No fault, no foul. Things happen out of order, but no deadlock is possible as progress was made: DENY
  - If the host to which it gave its vote is not in the critical section
    - Deadlock is possible: RELINQUISH
  - The tie will now be broken in favor of lower ID host (and, even if not tied, no problem)

MAJORITY VOTING PERFORMANCE

- 3(N-1) messages (Request, OK, RELEASE)
  - No need to send to self, hence "minus 1"
  - **PLUS:** Up to 2(N-1) INQUIRE-RELINQUISH
- Robustness?
  - What happens if any hosts fail? <50% okay.
- Communication problems?
  - Missing? Usual: No release? Ouch. No Request? Ouch. No reply? Ouch
  - Upshot: Better rely upon TCP, etc.

VOTING DISTRICTS

- What we need is a way to reduce the number of hosts involved in making decisions.
- This way, fewer hosts need to vote, and fewer hosts need to reorganize their votes in the event of a misvote.
- In order to address to reduce the number of messages required to win an election we are going to organize the participating systems into voting districts called *coteries* (pronounced, "koh-tarz" or "koh-tErz"), such that winning an election within a single district implies winning the election across all districts.

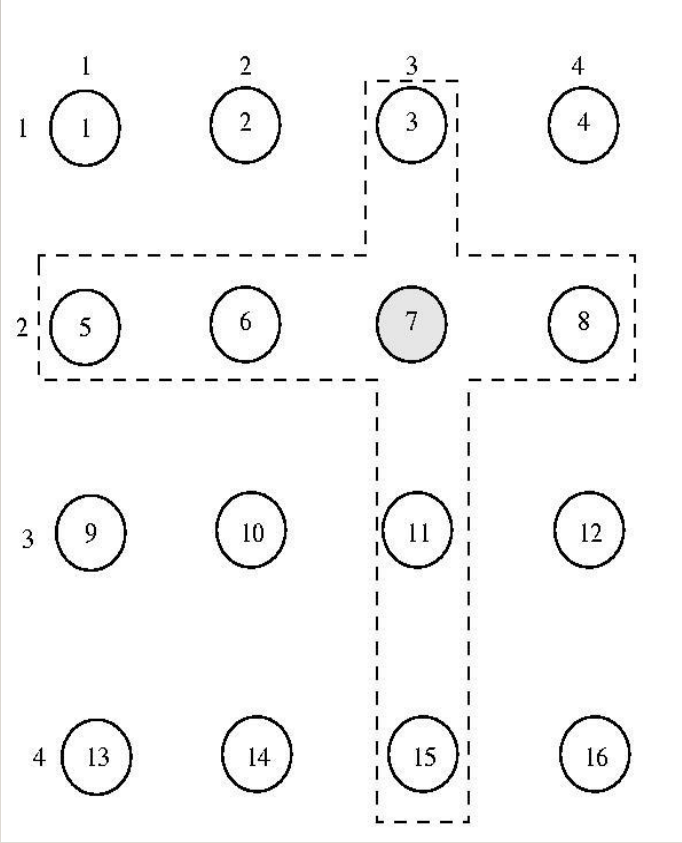
GERRYMANDERING: A US HISTORY MOMENT!

- *Gerrymandering* is a term that was coined by Federalists in the Massachusetts election of 1812. Governor Elbridge Gerry, a Republican, won a very narrow victory over his Federalist rival in the election of 1810. In order to improve their party's chances in the election of 1812, he and his Republican conspirators in the legislator redrew the electoral districts in an attempt to concentrate much of the Federalist vote into very few districts, while creating narrow, but majority, Republican suport in the others. The resulting districts were very irregular in shape. One Federalist commented that one among the new districts looked like a salamander. Another among his cohorts corrected him and declared that it was, in fact, a "Gerrymander." The term Gerrymandering, used to describe the process of contriving political districts to affect the outcome of an election, was born.
- Incidentally, it didn't work and the Republicans lost the election. He was subsequently appointed as Vice-President of the U.S. He served in that role for two years. Since that time both federal law and judge-made law have made Gerrymandering illegal.

BACK TO…
VOTING DISTRICTS

- The method of Gerrymandering disticts that we'll study was developed by Maekawa and published in 1985.
- Using this method, processor's are organized into a grid.
  - Each processor's voting district contains all processors on the same row as the processor and all processors on the same column.
  - That is to say that the voting district of a particular processor are all of those systems that form a perpendicular cross through the processor within the grid.
  - Given N nodes, (2*SQRT(n) – 1) nodes will compose each voting district.

VOTING DISTRICT EXAMPLE



Request:

- •Send a REQUEST to each and every member of host's district
- •Wait until each and every member of district votes YES (Unanimous YES)
- •Enter the critical section

Release

- •Upon exit from the CS, send RELEASE to each and every member of host's district

Received a REQUEST?

- •If already voted in an outstanding election, enqueue the request.
- •Otherwise send YES

•Receive a RELEASE?

- •Dequeue oldest request from its queue, if any. Send a YES vote to this node, if any.

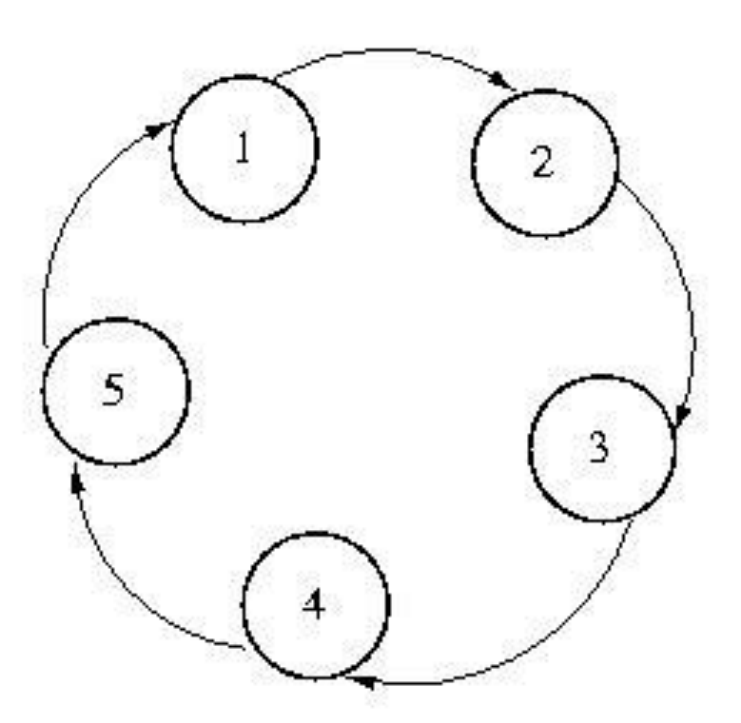
VOTING DISTRICTS: TIES

- Just as before…
- Imagine M concurrent requests, each getting exactly N/M votes.
  - We're stuck!
- To get unstuck, we use Lamport ordering w/hostID tie-breaking and favor earlier requests
- If a host gets an earlier request after voting for a later one, it asks for its vote back (INQUIRE)
  - If the host to which it gave its vote is in the critical section
    - No fault, no foul. Things happen out of order, but no deadlock is possible as progress was made: DENY
  - If the host to which it gave its vote is not in the critical section
    - Deadlock is possible: RELINQUISH
  - The tie will now be broken in favor of lower ID host (and, even if not tied, no problem)

VOTING DISTRICT PERFORMANCE

- 3*(2*SQRT(N)-1) messages (Request, YES, RELEASE)
  - No need to send to self, hence "minus 1"
  - **PLUS:** Up to (2*SQRT(N)-1) INQUIRE-RELINQUISH
- Robustness?
  - What happens if any host fail? Intersecting districts can't get critical section. Ouch!
- Communication problems?
  - Missing? Usual: No release? Ouch. No Request? Ouch. No reply? Ouch
  - Upshot: Better rely upon TCP, etc.

TOKEN RING APPROACH



- Pass token around ring. Host with it has access to CS.
- High contention? 1 message/request
- Low contention? (N-1) messages/request

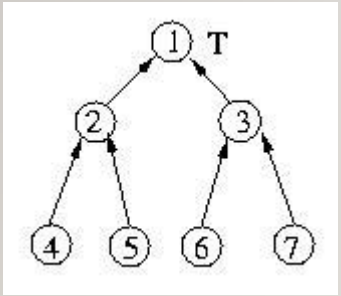
TOKEN RING: FAILURE

- Failed host or link to host
  - Pass token to next logical node
- Failure within CS
  - Minimally, no worse than any other situation
  - With max CS time, can use timer to regenerate
    - At time out, start ROSTER message around ring
      - If sent ROSTER and receive ROSTER only propagate if from higher hostID.
      - Highest host gets ROSTER will all hosts and generates new token
      - Many regeneration messages can be sent.
    - Can result in multiple tokens if partitioning
      - Okay if can make parallel progress slower, or if only one partition can reach resources needed for progress
      - Can require majority to generate token, preventing multiple (also possibly preventing regeneration)

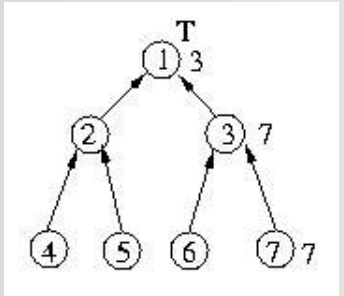
RAYMOND'S ALGORITHM

- Another way of approaching token-based mutual exclusion is to organize the hosts into a tree instead of a ring.
- This organization allows the token to travel from host-to-host, traversing far fewer unnecessary hosts.
- Raymond's algorithm is one such approach.
  - It organizes all of the nodes into an unrooted n-ary tree.
  - When the system is initialized, one node is given the token
  - The other nodes are organized so that they form a tree.
  - The edges of this tree are directional -- they must always point in the direction of the token.

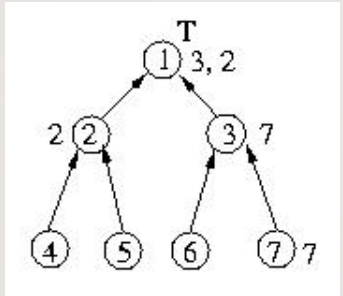
RAYMOND'S ALGORITHM EXAMPLE



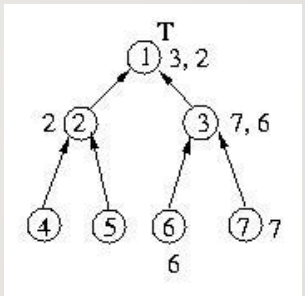
Initial



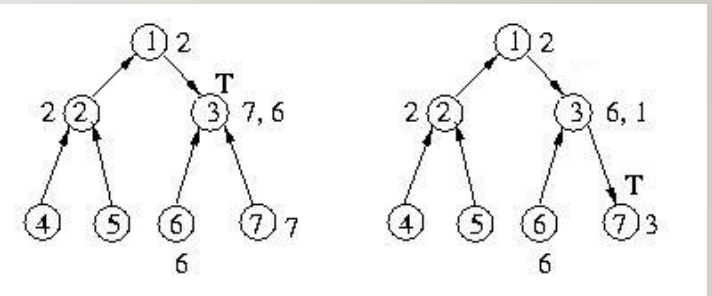
7 Makes Request



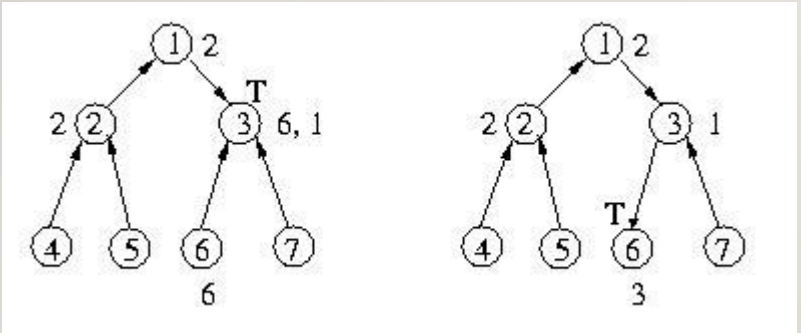
2 Makes Request



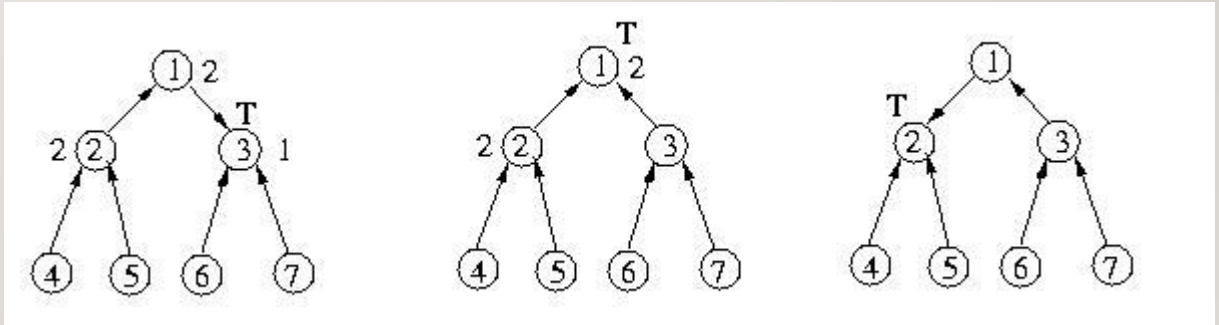
6 Makes Request



1 Exits



7 Exits



6 Exits

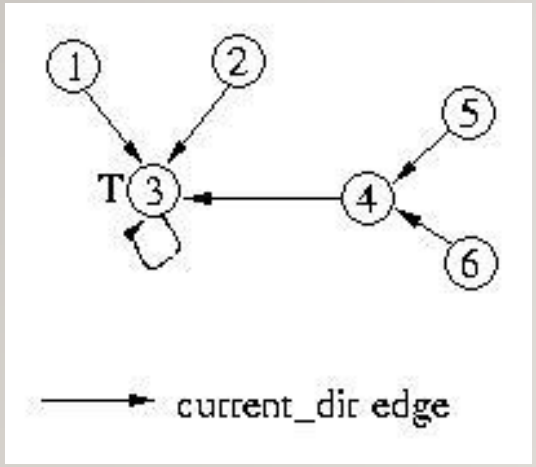
RAYMOND'S ALGORITHM PERFORMANCE

- Obviously not robust to failure – breaks path
- Worst case number of messages?
  - Up and down, 2*log(N)
- Not FIFO/FCFS
  - But, bounded wait
  - May handle local request first, but then goes up before coming back down

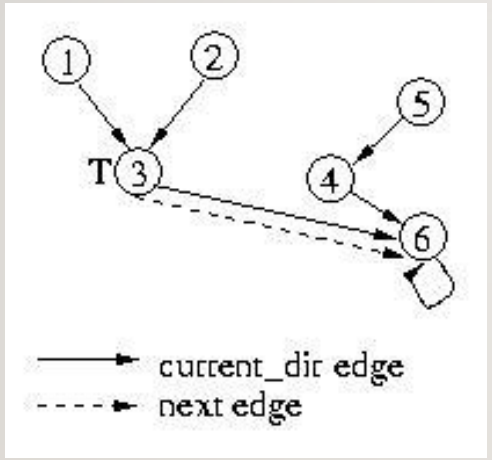
PATH COMPRESSION (LI AND HUDAK)

- Originally developed to pass memory objects
  - But, memory object can be a simple token
- Based on a queue of pending requests. The queue is maintained implicitly by two different types of edges among the nodes:
  - Each node's *current_dir* edge leads to its best guess of the node that is at "the end of the line" of hosts waiting for access to the critical section.
    - The node "at the end of the line" has this pointer set to itself.
    - Current edges may be out of date. This is because a node may not be aware of the fact that additional nodes have been enqueued. But this is okay. A request can follow the current edge to the "old" end of the queue. This node will in turn lead to a node farther back in the list. Eventually, the request will come to the end of the list.
    - Once a request reaches the end of the current edge chain, it will also be at the back of the queue maintained by the next pointers, so it can "get in line" and take its place at the end of the queue.
- The *next* edge is only valid for those nodes that either have the token or have requested the token.
  - If there is a next edge from node A to node B, this indicates that node A will pass the token to node B, once it has exited the critical section.
  - Nodes which have not requested the critical section, or have no requests enqueued after them, have a null next pointer.
  - In this way, the next pointer forms the queue of requests. The next pointer from the token holder to the next node forms the head of the list. The next pointer from that point forward indicates the order in which the nodes that have made requests for the critical section will get the token.
- How do the current edges get updated?
  - As a request is percolating through the nodes via the current edges, each node's current edge is adjusted to point to the requesting node. Why? Well the requesting node is the most recent request and is (or will soon be) at the end of the request queue.

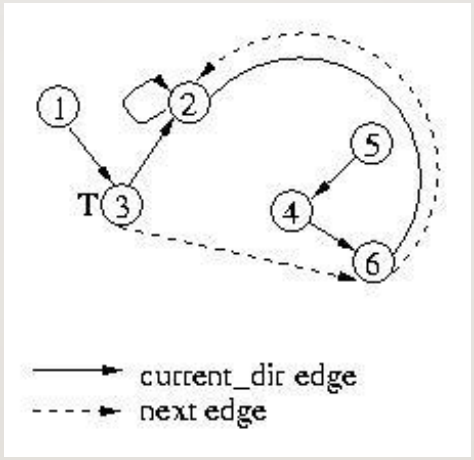
PATH COMPRESSION (LI AND HUDAK)



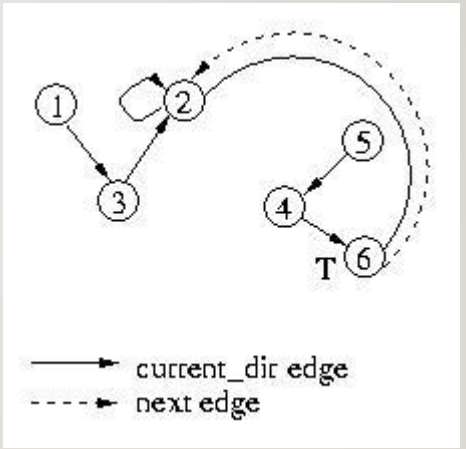
Start



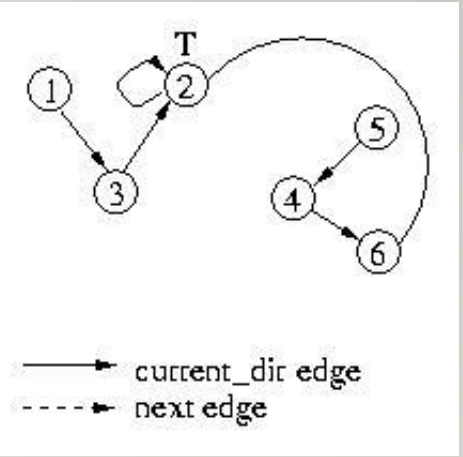
6 requests token



2 requests token



3 releases token



6 releases token

- Fewer messages/request than Raymond's
- Still not robust to failure.

LEASES (GENERAL NOTE)

- Each approach has a problem:
  - What happens if a client dies/walks-away/hogs a resource?
    - Everything stops
- When possible it is most often preferable to "Lease" resources than to give them away
  - Permission to use resource limited in time
  - Not honored after that
  - If needed longer, can request renewal
    - Decision to renew may be nearly automatic (DHCP prefers to renew IP addresses to keep them stable)
    - Or, it may be nearly automatic (Deny – get back in line)
    - Or, it may be based upon demand, etc.