CSCE 310J
Data Structures & Algorithms

Distributed Synchronization

Dr. Steve Goddard
goddard@cse.unl.edu

http://www.cse.unl.edu/~goddard/Courses/CSCE310J

Synchronization

- Coordinating processes to achieve common goal
  - process precedence
  - critical sections
- Synchronization on centralized machines
  - semaphores, monitors, etc.
    - all rely on shared memory
  - event ordering (also used for synchronization)
    - just use kernel's clock

Distributed Synchronization

- Memory is not shared
- Clock is not shared
- Decisions are usually based on local information
- Centralized solutions undesirable (single point of failure, performance bottleneck)

Global Clock Synchronization

- Generally impossible to synchronize clocks
  - clock skew - all crystals run at slightly different rates
    - not a problem for centralized systems
  - 'make' example in book
  - can periodically synchronize clocks
    - but how long does it take to transmit the synch message?
    - what if it has to be re-transmitted?
- Lamport: clock synchronization does not have to be exact
  - synchronization not needed if there is no interaction between machines
  - synchronization only needed when machines communicate
  - i.e. must only agree on ordering of interacting events

Event Ordering

- <u>Happened-before</u> relation
  - denoted by $\rightarrow$
- Partial orders
  - e_i and e_j, are two events
  - if e_i and e_j are in the same process
    - if $e_i \rightarrow e_j$, then e_i occurs before e_j
  - if e_i is the transmission of a message, and e_j is its reception
    - then $e_i \rightarrow e_j$
  - transitivity holds
    - $(e_i \rightarrow e_j)$ and $(e_j \rightarrow e_k) \Rightarrow e_i \rightarrow e_k$

Logical Clocks

- Substitute synchronized clocks with a global ordering of events
  - LC_i is a local clock: contains increasing values
    - each process i has own LC_i
  - increment LC_i on each event occurrence
  - $e_i \rightarrow e_j \Rightarrow LC(e_i) < LC(e_j)$
  - within same process i, if e_j occurs before e_k
    - $LC_i(e_j) < LC_i(e_k)$
  - if e_s is a send event and e_r receives that send, then
    - $LC_i(e_s) < LC_j(e_r)$

Logical Clocks (cont.)

- Timestamp
  - » each event is given a timestamp t
  - » if e_s is a send message m from p_i, then $t = LC_i(e_s)$
  - » when p_j receives m, set LC_j value as follows
    - if $t < LC_j$, increment LC_j by one
      - message regarded as next event on j
    - if $t \geq LC_j$, set LC_j to $t + 1$

Logical Clocks (cont.)

- Achieves clock synchronization across processes
  - » all that matters is when the processes need to synchronize - messages are required
  - » Two cases:
    - $t < LC_j$
      - $LC_j = LC_j + 1$
    - $t \geq LC_j$
      - $LC_j = t + 1$

Physical Clocks

- Must be synchronized with real world
- In a distributed system, they must be synchronized with each other as well!
- Universal Coordinated Time (UTC)
  - » Based on International Atomic Time (TAI)
    - which is based on transitions of a cesium 133 atom
  - » Broadcast by
    - NIST out of Fort Collins, CO on WWV (Short Wave)
    - Geostationary Environment Operation Satellite(GEOS)

Clock Synchronization Algorithms

- Goal
  - » Keep all clocks as synchronized as possible
  - » $dC/dt = 1$
- Reality
  - » Clocks drift with maximum drift rate ρ
  - » $1-\rho \leq dC/dt \leq 1+\rho$
  - » Must synch at least every $\delta/2\rho$ time units to keep all clocks with δ time units of each other

Cristian's Algorithm

- Periodically, clients ask a Time Server for the correct time, C_{UTC}
  - » Let time of
    - request be T_0, time of reply be T_1, server interrupt handling time be I
  - » $C_p = C_{UTC} + (T_1 - T_0 - I)/2$
    - Problem:
      - time cannot go backwards
      - slow down or speed up gradually
- Improve accuracy with a series of requests/measurements

Berkeley Algorithm

- Time server (daemon) is active
  - » sends clients its time periodically
  - » clients send back delta
  - » server averages responses
  - » tells each client how to adjust its clock
- Can be used with or without a WWV receiver
- Highly centralized (as is Cristian's algorithm)

Decentralized Averaging Algorithms

- Divide time into quanta
- At the end of each quantum
  - » Each machine broadcasts its current time
  - » Each machine averages all of the responses and sets its own clock accordingly
  - » Can discard highest and lowest *m* values to
- Variation account for propagation delay.

Using Synchronized Clocks
Implementing at-most-once semantics

- Traditional approach
  - » each message has unique message id
  - » server maintains list of id's
  - » can lose message numbers on server crash
  - » how long does server keep id's?
- With globally synchronized clocks
  - » sender assigns a timestamp to message
  - » server keeps most recent timestamp for each connection
    - reject any message with lower timestamp (is a duplicate)
  - » removing old timestamps
    - G = CurrentTime - MaxLifeTime - MaxClockSkew
    - timestamps older than G are removed

At-Most-Once Semantics (cont.)

- After a server crash
  - » CurrentTime is recomputed
    - using global synchronization of time
  - » all messages older than G are rejected
  - » meaning all messages before crash are rejected as duplicate
    - some new messages may be wrongfully rejected
    - but at-most-once semantics is guaranteed

Using Synchronized Clocks
Cache Consistency

- Problem if two simultaneously update
  - » solution: distinguish between caching for read or write
    - readers must invalidate cache if writer is present
    - server must verify that all readers have invalidated their cache
    - even if cache is very old
- Clock-based cache consistency
  - » clients given a "lease"
    - specifies how long cache is valid
    - clients can renew leases without re-caching
  - » server invalidates caches whose leases have not expired
    - if there is a client crash, just wait for lease to expire
  - » global clock ensures agreement of lease time
    - even in the face of crashes

Mutual Exclusion in Distributed Systems

- Centralized mutex
  - » choose a coordinator
    - all critical region (CR) requests go to coordinator
    - coordinator grants or denies permission
- Request/reply model
  - » p1 requests, CR is available
    - coordinator sends a reply
    - reply indicates permission to enter CR
  - » queue subsequent requests
    - do not send a reply
  - » when p1 finished, send a reply to first in queue

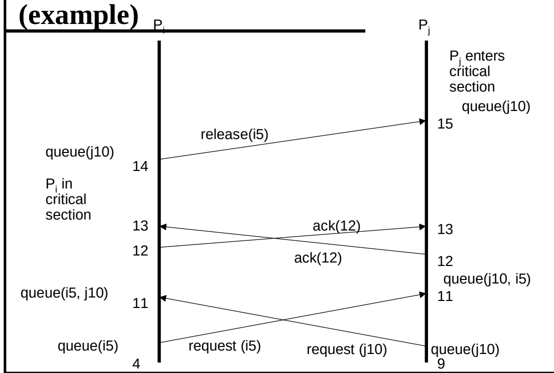
Mutual Exclusion (cont.)

- Request/grant or deny model
  - » send 'permission denied' when CR is busy
  - » two possibilities
    - send 'grant' message when process given CR
    - let requesting process decide what to do - polling
- Problems with centralized approach
  - » single point of failure, bottleneck (the usual...)
- Distributed algorithm (Lamport)
  - » use logical clocks to achieve mutual exclusion
  - » each process has a <u>request queue</u>
  - » decisions made locally, global exclusion maintained

Lamport's Distributed Mutex Alg. Using Logical Clocks

- Suppose P_i wants access to critical region
  - » P_i sends message with T_m to every process
  - » P_j receives message, places it on request queue, sends ack with T_r
  - » P_i gets resource when:
    - 1) T_m in P_i's request queue < all other time stamps
    - 2) P_i receives ack messages from all other processes timestamped later than T_m
    - note that control is local to P_i
  - » when i finished with CR
    - P_i removes T_m from message queue, sends timestamped "P_i releases resource" message
    - P_j's receiving the message remove T_m's from queue

Lamport's Algorithm (example)



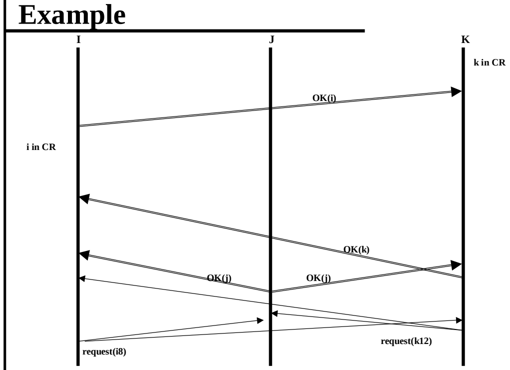
Ricart and Agrawala

- Lamport's algorithm
  - » requires 3(N-1) messages per critical section request
    - broadcast mediums reduce to 3 messages
- Ricart and Agrawala's algorithm
  - » requires only a <u>request</u> and <u>reply</u> message
  - » (no *release* required)
  - » therefore 2(N-1) messages per CS request

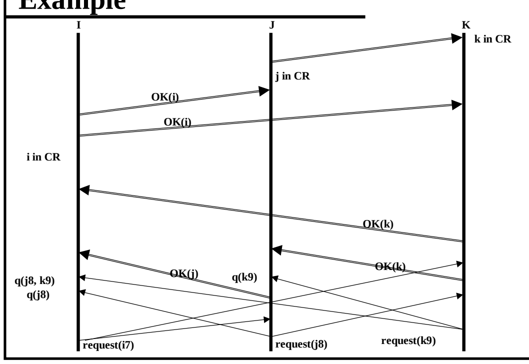
Richart and Agrawala's Algorithm

- When receiving a request from process P_i:
  - » receiver is not in and does not want CR
    - send OK to P_i
  - » receiver already in CR
    - queue the request
  - » receiver wants CR, but has not been granted
    - if timestamp > P_i's, send OK to P_i
    - otherwise, queue request
- When finished with CR, process sends OK to <u>all</u> processes in queue
- P_i enters critical section after receiving OK replies from all other processes in group

Richart and Agrawala Example



Richart and Agrawala Example



Problems with Both Algorithms

- No single point of failure
  - » each process makes independent decisions
  - » But what if one process doesn't send an OK?
    - a form of deadlock
  - » now there are *n* points of failure
- Group communication is needed
  - » must maintain a list of group members
  - » either each process...
  - » or use primitives discussed in Chapter 2
- All processes are involved in all decisions
  - » increases the overall system load

Token Passing Mutex

- General structure
  - » one token per CR
  - » only process with token allowed in CR
  - » token passed from process to process
    - logical ring
- Mutex
  - » pass token to process i + 1 mod N
  - » received token gives permission to enter CR
    - hold token while in CR
  - » must pass token after exiting CR
  - » fairness ensured: each process waits at most n-1 entries to get CR

Token Passing Mutex

- Difficulties with token passing mutex
  - » lost tokens: electing a new token generator
  - » duplicate tokens: ensure by not generating more than one token

Mutex Comparison

- Centralized
  - » simplest, most efficient
  - » centralized coordinator crashes
    - need to choose a new coordinator
- Distributed
  - » 2(n-1) messages per entry/exit (Ricart & Agrawala)
  - » if any process crashes with a non-empty queue, algorithm won't work
- Token Ring
  - » if there are lots of CR requests, between 0 and unbounded # of messages per entry/request
    - if CR requests rare, unbounded number of messages
  - » need methods for re-generating a lost token

Election Algorithms

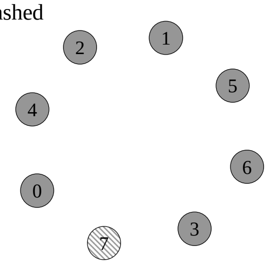
- Centralized approaches often necessary
  - » best choice in mutex, for example
  - » but need method of electing a new coordinator when it fails
- General assumptions
  - » give processes unique system/global numbers
  - » elect (live) process with highest process number
  - » processes know process number of members
  - » all processes agree on new coordinator

The Bully Algorithm

- Suppose the coordinator doesn't respond to p1's request
  - » p1 holds an election by sending an *election* message <u>to all processes with higher numbers</u>
  - » if p1 receives no responses, p1 is the new coordinator
  - » if any higher numbered process responds, p1 ends its election
- If a process with a higher number receives an election request
  - » reply to the sender
    - to tell sender that it has lost the election
  - » hold an election of its own
  - » eventually all give up but highest surviving process

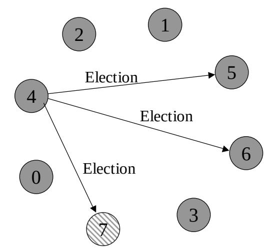
The Bully Algorithm (cont.)

- Example: processes 0-7, 4 detects that 7 has crashed



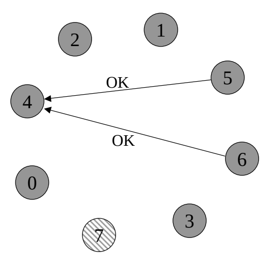
The Bully Algorithm (cont.)

- Example: process 4 holds an election



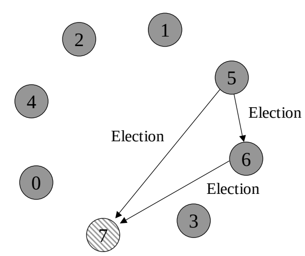
The Bully Algorithm (cont.)

- Example: processes 5 and 6 respond with OK



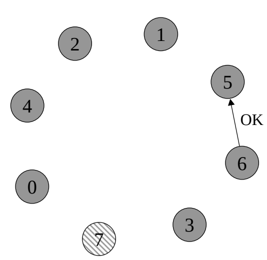
The Bully Algorithm (cont.)

- Example: Processes 5 and 6 hold elections



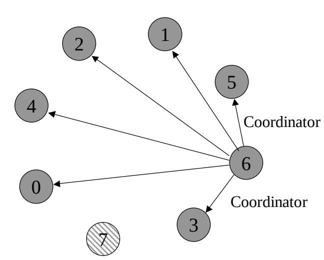
The Bully Algorithm (cont.)

- Example: process 6 sends OK



The Bully Algorithm (cont.)

- Example: process 6 is the new Coordinator



Ring Algorithm

- Processes are ordered
  - » each process knows its successor
  - » no token involved
- Any process noticing that the coordinator is not responding
  - » sends an *election* message to its successor
    - ❖ if successor is down, send to next member
    - ❖ therefore each process has full knowledge of the ring
  - » receiving process adds its number to the message and passes it along
- When message gets back to election initiator
  - » change message to coordinator
  - » circulate to all members
    - ❖ note that members now have complete (and ordered) list of members
  - » coordinator is highest process number

Ring Algorithm (cont.)

- What if more than one process detects a crashed coordinator?
  - » more than one election will be produced
  - » all messages will contain the same information
    - ❖ member process numbers
    - ❖ order of members
  - » same coordinator is chosen (highest number)