Week 11

* Clock Synchronization
  + Distributed systems: each node has
* Physical vs. Logical Clocks
  + Physical clocks keep time of day
    - Consistent across systems
  + Logical clock keeps track of event ordering
    - Among related events
* Physical Clocks Expanded
  + Clocks are implemented as an oscillator and a counter
    - Two clocks hardly ever agree, they oscillate at slightly different frequencies
    - Results in clock drift
  + Coordinated universal time
    - UTC
    - International standard on atomic time
      * Sometimes add leap seconds to be consistent with astronomical time
      * Perceive them as minor adjustments
  + Synchronize machines clocks with a time server (master) or with another one
* Christian’s Algorithm
  + Synchronize machines to a time server with a UTC receiver
  + Compensate for network delays (assumes symmetry)
    - Client sends request at T = 0
    - Server replies with the current clock value T of server
    - Client receives response at T 1
    - See notebook or slides for diagram
* Network Time Protocol
  + Widely used standard – based on Christian’s Algorithm
    - Improves accuracy by making a series of measurements
  + Goal: allow clients to accurately sync time considering delays
  + Collects eight pars of delays from A to B and from B to A
    - Chooses time offset for which the delay was minimal
    - See notebook or slides for diagram
* Berkeley Algorithm
  + Assumes NO machine as an accurate time source
    - Used in systems without UTC receiver
  + Synchronize clocks with one another
    - One computer elected time server (master)
    - Time server polls times from workers
      * Average it and return differences
      * One machines clock might be disregarded if it is an outlier
* To Summarize Physical Clocks
  + Cristian’s algorithm and NTP
    - Set clock from server
    - Accounts for delats
  + Berkely
    - Syncs clocks together from an average
* Logical Clocks
  + Lamport [1978]
    - Clock synchronization need not to be absolute
    - If two machines do not interact, no need to synchronize them
    - Processes need to agree on the order in which events occur rather than the time at which they occur
* Event Ordering
  + Problem: define a total ordering of all events that occur in a system
  + Single processor machine events are totally ordered
  + Distributed system
    - No global clock, local clocks may be unsynced
  + Key ideas
    - Processes exchange messages
    - Message must be sent before received
    - Send/receive used to order events (and synch clocks)
  + To sync logical clocks, lamport defined a relation called “happens before”
* Lamport Logical Clocks
  + Happened before relation
  + Ex:
    - If A and B are events in the same process and A executed before B, then A->B
    - If A represents sending of a message and B is the receipt of this message, then A-> B
    - Transitive:
      * A->B and B->C then A->C
  + The problem
    - How do we maintain a global view on the system’s behavior that is consistent with the happened-before relation?
      * The answer here is using a timestamp
      * Ex:
        + TC(x) is given to event x
* Lamport Timestamps
  + Each process Pi maintains a local counter TCi

Thursday

* DS: Coordination (cont.)
  + Mutual exclusion
  + Leader election
  + Global states and snapshots
* Mutual Exclusion – Process Synchronization
  + Techniques to coordinate execution among processes
    - One process may have to wait for another
    - Shared resource may require exclusive access
  + Examples
    - Modify a shared file
    - Modify contents that are replicated on multiple servers
* Mutual Exclusion: general
  + Problem
    - Multiple processes in a DS want exclusive access to some resource
  + Key idea
    - Have an algorithm that allows a process to request and obtain exclusive access to a resource available on the network
* Mutual Exclusion Algorithms
  + Categories
    - Permission based: process needs permission from other processes to access a resource or enter a critical section
      * Centralized
        + A process can access a resource because a central coordinator allowed it to do so
      * Distributed
        + A process can access a resource via distributed agreement
    - Token based:
      * A token is passed between processes
* Centralized Algorithm
  + One process elected as a coordinator
  + Steps:
    - Request resource
    - Wait for response
    - Receive grant (ok)
    - Assess resource
    - Release resource
  + If another process claimed resource
    - Coordinator does not reply until release
    - Maintain queue: service requests in FIFO order
  + Additional notes
    - All requests processed in order
    - Other processors do not know group members (only coordinator)
* Distributed Algorithm
  + Ricart & agrawala algorithm
  + Based on event ordering and time stamps
  + Assumes total ordering of events in the system (Lamport’s clock)
  + Process wants to enter a critical section:
    - Generate new time stamp TS = TS + 1
    - Send (multicast) request(process ID, TS) all other n – 1 processes
    - Wait until reply (ok) received from all other processes
    - Enter critical section
  + Process receives request
    - If receiving process has no interest in shared resource
      * Send ok reply to sender
    - If receiving process is in the critical section
      * Do not reply, add request to queue
    - If receiving process is waiting for the resource”
      * Compare timestamps
      * If receiver has lower priority
        + Send okay
      * If receiver has higher priority
        + When done with critical section: send ok to all queued requests
  + Additional notes
    - Two comparing processes will not send OK to each other
    - N-1 request messages and N-1 ok messages
    - Lots of messaging traffic
    - N points of failure
* Token Base Mutual Exclusion Algorithm
  + Idea
    - Organize processes in a logical ring, and let a token be passed between them
    - Process that holds the token is allowed to enter the critical region
  + Initialization
    - Process 0 creates a token for resource R
  + Token circulates around ring
  + When process acquires token
    - Checks to see if it needs to enter critical section
    - If no, send token to neighbor
    - If yes, access resource
      * Hold token until done
  + Additional notes
    - Only one process at a time has token
      * Mutual exclusion guaranteed
    - Order well-defined (but not necessarily first-come, first-served)
    - Request 0 … N – 1 messages and release 1 message
* Leader Election Algorithms
  + Leader Election:
    - Many distributed algorithms need one process to act as coordinator
      * Ex. Take over the role of a failed process, pick a master in Berkeley clock synchronization algorithm
    - Election algorithms
      * Technique to pick a unique coordinator (aka leader election)
* Election Algorithms
  + Assumptions
    - Each process P has a unique identifier ID
    - Every process knows the identifier of every other process
  + Two election algorithms
    - Bully
    - Ring
* Bully Algorithm
  + Each process has a unique numerical ID, and knows the ID’s and address of every other process
  + Communication is assumed reliable
  + Key idea: select process with highest ID
  + A process P initiates election if it just recovered from failure or if a coordinator failed
  + 3 message types: election, OK, I won
  + Several processes can initiate an election simultaneously
    - Need consistent result
* Bull Algorithm Details
  + Any process P can initial an election
  + P sends election messages to all process with higher ID’s and awaits Ok messages
    - If no response, P wins election and becomes coordinator and sends “I won” messages to all process with lower ID’s
    - If P receives an OK, it drops out and waits for an “I won”
  + If a process receives an election msg
    - It returns an OK and starts an election
  + If a process receives an I Won
    - Considers sender the coordinator
* Ring Based Election Algorithm