Last Class: Naming

- Name distribution: use hierarchies
- DNS
- X.500 and LDAP

 Canonical Problems in Distributed Systems

- Time ordering and clock synchronization
- Leader election
- Mutual exclusion
- Distributed transactions
- Deadlock detection
Clock Synchronization

- Time in unambiguous in centralized systems
  - System clock keeps time, all entities use this for time
- Distributed systems: each node has own system clock
  - Crystal-based clocks are less accurate (1 part in million)
  - Problem: An event that occurred after another may be assigned an earlier time

Physical Clocks: A Primer

- Accurate clocks are atomic oscillators (one part in $10^{13}$)
- Most clocks are less accurate (e.g., mechanical watches)
  - Computers use crystal-based blocks (one part in million)
  - Results in clock drift
- How do you tell time?
  - Use astronomical metrics (solar day)
- Coordinated universal time (UTC) – international standard based on atomic time
  - Add leap seconds to be consistent with astronomical time
  - UTC broadcast on radio (satellite and earth)
  - Receivers accurate to 0.1 – 10 ms
- Need to synchronize machines with a master or with one another
Clock Synchronization

- Each clock has a maximum drift rate $\rho$
  - $1 - \rho \leq \frac{dC}{dt} \leq 1 + \rho$
  - Two clocks may drift by $2\rho \Delta t$ in time $\Delta t$
  - To limit drift to $\delta \Rightarrow$ resynchronize every $\frac{\delta}{2\rho}$ seconds

Cristian’s Algorithm

- Synchronize machines to a time server with a UTC receiver
- Machine P requests time from server every $\frac{\delta}{2\rho}$ seconds
  - Receives time $t$ from server, P sets clock to $t + t_{\text{reply}}$ where $t_{\text{reply}}$ is the time to send reply to P
  - Use $(t_{\text{req}} + t_{\text{reply}})/2$ as an estimate of $t_{\text{reply}}$
  - Improve accuracy by making a series of measurements
Berkeley Algorithm

- Used in systems without UTC receiver
  - Keep clocks synchronized with one another
  - One computer is *master*, other are *slaves*
  - Master periodically polls slaves for their times
    - Average times and return differences to slaves
    - Communication delays compensated as in Cristian’s algo
  - Failure of master => election of a new master

---

**Berkeley Algorithm**

a) The time daemon asks all the other machines for their clock values
b) The machines answer
c) The time daemon tells everyone how to adjust their clock
Distributed Approaches

- Both approaches studied thus far are centralized
- Decentralized algorithms: use resync intervals
  - Broadcast time at the start of the interval
  - Collect all other broadcast that arrive in a period $S$
  - Use average value of all reported times
  - Can throw away few highest and lowest values
- Approaches in use today
  - `rdate`: synchronizes a machine with a specified machine
  - Network Time Protocol (NTP)
    - Uses advanced techniques for accuracies of 1-50 ms

Logical Clocks

- For many problems, internal consistency of clocks is important
  - Absolute time is less important
  - Use *logical* clocks
- Key idea:
  - Clock synchronization need not be absolute
  - If two machines do not interact, no need to synchronize them
  - More importantly, processes need to agree on the *order* in which events occur rather than the *time* at which they occurred
Event Ordering

• **Problem:** define a total ordering of all events that occur in a system

• Events in a single processor machine are totally ordered

• In a distributed system:
  – No global clock, local clocks may be unsynchronized
  – Can not order events on different machines using local times

• Key idea [Lamport ]
  – Processes exchange messages
  – Message must be sent before received
  – Send/receive used to order events (and synchronize clocks)

Happened Before Relation

• If A and B are events in the same process and A executed before B, then  $A \rightarrow B$

• If A represents sending of a message and B is the receipt of this message, then A -> B

• Relation is transitive:
  – A -> B and B -> C => A -> C

• Relation is undefined across processes that do not exchange messages
  – Partial ordering on events
Event Ordering Using HB

- Goal: define the notion of time of an event such that
  - If A-> B then C(A) < C(B)
  - If A and B are concurrent, then C(A) <, = or > C(B)
- Solution:
  - Each processor maintains a logical clock LC_i
  - Whenever an event occurs locally at i, LC_i = LC_i + 1
  - When i sends message to j, piggyback Lc_i
  - When j receives message from i
    - If LC_j < LC_i then LC_j = LC_i + 1 else do nothing
  - Claim: this algorithm meets the above goals

Lamport’s Logical Clocks

(a) (b)
Example: Totally-Ordered Multicasting

- Update 1 is performed before update 2.
- Update 2 is performed before update 1.

Replicated database

- Update 1
- Update 2