Time in Distributed Systems

- no common clock in distributed system
- physical clock design
- coordinated universal time (UTC)
- synchronizing physical clocks
- Cristian’s algorithm
- Berkeley’s algorithm
- network time protocol (NTP)
- compensating for clock drift

Inherent Limitations of a Distributed System

- A distributed system is a set of computers that communicate over a network, and do not share a common memory or a common clock
- Absence of a common (global) clock
- No concept of global time
- It’s difficult to reason about the temporal ordering of events
- Cooperation between processes (e.g., producer/consumer, client/server)
- Arrival of requests to the OS (e.g., for resources)
- Collecting up-to-date global state
- It’s difficult to design and debug algorithms in a distributed system
- Mutual exclusion
- Synchronization
- Deadlock

Physical clocks in a distributed system

- Every computer contains a physical clock
- A clock (also called a timer) is an electronic device that counts oscillations in a crystal at a particular frequency
- Count is typically divided and stored in a counter register
- Clock can be programmed to generate interrupts at regular intervals (e.g., at time interval required by a CPU scheduler)
- Counter can be scaled to get time of day
- This value can be used to timestamp an event on that computer
- Two events will have different timestamps only if clock resolution is sufficiently small
- Many applications are interested only in the order of the events, not the exact time of day at which they occurred, so this scaling is often not necessary

Coordinated universal time

- The output of the atomic clocks is called International Atomic Time
- Coordinated Universal Time (UTC) is an international standard based on atomic time, with an occasional leap second added or deleted
- UTC signals are synchronized and broadcast regularly by various radio stations (e.g., WWV in the US) and satellites (e.g., GEOS, GPS)
- Have propagation delay due to speed of light, distance from broadcast source, atmospheric conditions, etc.
- Received value is only accurate to 0.1–10 milliseconds
- Unfortunately, most workstations and PCs don’t have UTC receivers
Synchronizing physical clocks

- Use a time server with a UTC receiver
- Centralized algorithms
  - Client sets time to $T_{\text{server}} + D_{\text{trans}}$
  - $D_{\text{trans}}$ = transmission delay
  - Unpredictable due to network traffic
- Cristian's algorithm (1989):
  - Send request to time server, measure time $D_{\text{trans}}$ taken to receive reply $T_{\text{server}}$
  - Set local time to $T_{\text{server}} + \frac{D_{\text{trans}}}{2}$
  - Accuracy is $\pm \left(\frac{D_{\text{trans}}}{2} - D_{\text{min}}\right)$
  - Improvement: make several requests, take average $T_{\text{server}}$ value
- Assumptions:
  - Network delay is fairly consistent
  - Request & reply take equal amount of time
- Problems:
  - Doesn't work if time server fails
  - Not secure against malfunctioning time server, or malicious impostor time server

Synchronizing physical clocks (cont.)

- Centralized algorithms (cont.)
  - Berkeley (Gusella & Zatti) algorithm (1989):
    - Choose a coordinator computer to act as the master
    - Master periodically polls the slaves — the other computers whose clocks should be synchronized to the master
    - Slaves send their clock value to master
    - Master observes transmission delays, and estimates their local clock times
    - Master averages everyone's clock times (including its own)
      - Master takes a fault-tolerant average — it ignores readings from clocks that have drifted badly, or that have failed and are producing readings far outside the range of the other clocks
    - Master sends to each slave the amount (positive or negative) by which it should adjust its clock
- Distributed algorithms (see text...)

Compensating for clock drift

- Compare time $T_s$ provided by time server to time $T_c$ at computer $C$
  - If $T_s > T_c$ (e.g., 9:07am vs 9:05am)
    - Could advance C's time to $T_s$
    - May miss some clock ticks; probably OK
  - If $T_s < T_c$ (e.g., 9:07am vs 9:10am)
    - Can't roll back C's time to $T_s$
    - Many applications (e.g., make) assume that time always advances!
    - Can cause C's clock to run slowly until it resynchronizes with the time server
      - Can't change the clock oscillator rate, so have to change the software interpreting the clock's counter register
      - $T_{\text{software}} = a T_{\text{hardware}} + b$
      - Can determine constants $a$ and $b$

Is It Enough to Synchronize Physical Clocks?

- Summary:
  - In a distributed system, there is no common clock, so we have to:
    - Use atomic clocks to minimize clock drift
    - Synchronize with time servers that have UTC receivers, trying to compensate for unpredictable network delay
  - Is this sufficient?
    - Value received from UTC receiver is only accurate to within 0.1–10 milliseconds
    - At best, we can synchronize clocks to within 10–30 milliseconds of each other
    - We have to synchronize frequently, to avoid local clock drift
    - In 10 ms, a 100 MIPS machine can execute 1 million instructions
    - Accurate enough as time-of-day

*Not sufficiently accurate* to determine the relative order of events on different computers in a distributed system