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

Inherent Limitations of a Distributed System (cont.)

- Absence of shared memory means “state” is distributed throughout system
- One process can get either:
  - a coherent but partial view of the system,
  - or an incoherent but complete (global) view of the system
  - where coherent means:
    - all processes make their observations at the same time
  - where complete (or global) includes:
    - all local views of the state, plus
    - any messages that are in transit
- It is very difficult for every process to get a complete and coherent view of the global state
  - Example: one person has two bank accounts, and is in process of transferring $50 between the two accounts

Why Do We Care About “Time” in a Distributed System?

- May need to know the time of day some event happened on a specific computer
  - Need to synchronize that computer’s clock with some external authoritative source of time (external clock synchronization)
    - How hard is this to do?
- May need to know the time interval, or relative order, between two events that happened on different computers
  - If their clocks are synchronized to each other to some known degree of accuracy (called internal clock synchronization), we can measure time relative to a local clock
    - Is this always consistent?
- Will ignore relativistic effects
  - Cannot ignore network’s unpredictability

Physical Clocks

- 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
Physical Clocks in a Distributed System

- Does this work?
  - Synchronize all the clocks to some known high degree of accuracy, and then
  - measure time relative to each local clock to determine order between two events

- Well, there are some problems…
  - It’s difficult to synchronize the clocks
  - Crystal-based clocks tend to drift over time — count time at different rates, and diverge from each other
    - Physical variations in the crystals, temperature variations, etc.
    - Drift is small, but adds up over time
    - For quartz crystal clocks, typical drift rate is about one second every $10^6$ seconds = 11.6 days
    - Best atomic clocks have drift rate of one second in $10^{13}$ seconds = 300,000 years

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

- Centralized algorithms
  - Use a time server with a UTC receiver, and synchronize everyone to this time
  - Client sets time to $T_{server} + D_{trans}$
    - $T_{server}$ = server’s time
    - $D_{trans}$ = transmission delay
      - Unpredictable due to network traffic
  - Cristian’s algorithm (1989):
    - Nodes send request to time server, measure time $D_{trans}$ to receive reply $T_{server}$
    - Nodes set local time to $T_{server} + (D_{trans} / 2)$
      - Accuracy is $\pm (D_{trans} / 2) - D_{min}$
      - Improvement: make several requests, take average $T_{server}$ 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

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
Synchronizing Physical Clocks (cont.)

- **Distributed algorithms**
  - All nodes have a UTC receiver, but internal synchronization may still be desirable.
  - Global averaging:
    - Each node periodically broadcasts its time.
    - Each node collects times broadcast by other nodes, recording when it received each broadcast and the difference between its clock and theirs.
      - Then it takes a fault-tolerant average of the differences, and sets its local clock accordingly.
  - Problem:
    - A lot of network traffic.
  - Localized averaging:
    - Structure the nodes in some way (ring, tree, etc.) such that each node only averages values with a small subset of the total number of nodes.

Synchronizing Physical Clocks — Network Time Protocol (NTP)

- **Provides time service on the Internet**
- **Hierarchical network of servers:**
  - Primary servers (100s) — connected directly to a time source.
  - Secondary servers (1000s) — connected to primary servers in hierarchical fashion.
    - ns.mcs.kent.edu runs a time server.
  - Servers at higher levels are presumed to be more accurate than at lower levels.
- **Several synchronization modes:**
  - Multicast — for LANs, low accuracy.
  - Procedure call — similar to Cristian’s algorithm, higher accuracy (file servers).
  - Symmetric mode — exchange detailed messages, maintain history.
- All built on top of UDP (connectionless).

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.