Why remote procedure calls?
- Simple way to pass control and data
- Elegant transparent way to distribute application

Goal: provide transparency (or hide complexity)

Hard to provide true transparency
- Failures
- Performance
- Memory access
- Etc.

What did we do in the end? give up and let programmer deal with it
Today

- Understanding time in distributed systems
- Synchronizing physical clocks (real clocks)
- Logical Time/Lamport Clock
Time Example

- 3 Professors having dinner.
- 3 counters in different locations.
- They sneak out and pay at different counters.
- Who made the first payment?
Skew between computer clocks in a distributed system

- Computer clocks are not generally in perfect agreement
  - **Skew**: the difference between the times on two clocks (at any instant)

- Computer clocks are subject to clock drift (they count time at different rates)
  - **Clock drift rate**: the difference per unit of time from some ideal reference clock
  - Ordinary quartz clocks drift by about 1 sec in 11-12 days. \((10^{-6} \text{ secs/sec})\).
  - High precision quartz clocks drift rate is about \(10^{-7}\) or \(10^{-8}\) secs/sec
Coordinated Universal Time

- UTC is an international standard for time keeping.
- International Atomic Time is based on very accurate physical clocks (drift rate $10^{-13}$).
- It is based on atomic time, but occasionally adjusted to astronomical time.
- It is broadcast from radio stations on land and satellite (e.g. GPS).
- Computers with receivers can synchronize their clocks with these timing signals.
Coordinated Universal Time (UTC)

- Signals from land-based stations are accurate to about 0.1-10 millisecond
- Signals from GPS are accurate to about 1 microsecond
  - Why can't we put GPS receivers on all our computers?
Clock Synchronization Algorithms

- The relation between clock time and UTC when clocks tick at different rates.
When each machine has its own clock, an event that occurred after another event may nevertheless be assigned an earlier time.
Need for Precision Time

- Social networking services
- Stock market buy and sell orders
- Secure document timestamps (with cryptographic certification)
- Aviation traffic control and position reporting
- Radio and TV programming launch and monitoring
- Intruder detection, location and reporting
- Multimedia synchronization for real-time teleconferencing
- Interactive simulation event synchronization and ordering
- Network monitoring, measurement and control
- Early detection of failing network infrastructure devices and air conditioning equipment
- Differentiated services traffic engineering
- Distributed network gaming and training
Today

- Understanding time in distributed systems
- Synchronizing physical clocks (real clocks)
- Logical Time/Lamport Clock
Perfect networks

- Messages always arrive, with propagation delay exactly $d$
- Sender sends $T$
- Receiver sets clock to $T+d$
  - Synchronization is exact
Synchronous networks

- Messages always arrive, with propagation delay \textit{at most} $D$
- Sender sends time $T$
- Receiver sets clock to $T + D/2$
  - Synchronization error is at most $D/2$
Synchronization in the real world

- Real networks are asynchronous
  - Propagation delays are arbitrary
- Real networks are unreliable
  - Messages don’t always arrive
A time server $S$ receives signals from a UTC source

- Process $p$ requests time in $m_r$ and receives $t$ in $m_t$ from $S$
- $p$ sets its clock to $t + T_{round}/2$
- Window: $[t + \text{min}, t + T_{round} - \text{min}]$
- Size: $T_{round} - 2 \text{ min}$
- Accuracy: $\pm (T_{round}/2 - \text{min})$
- Min is the minimum transmission time of a message.

$T_{round}$ is the round trip time recorded by $p$

$\text{min}$ is an estimated minimum round trip time
Berkeley algorithm

- Cristian’s algorithm
  - A single time server might fail. Must use a group of synchronized servers.
  - Assumes the server’s clock is always perfect.

- Berkeley algorithm (also 1989)
  - An algorithm for internal synchronization of a group of computers
  - Master uses Cristian’s algorithm to get time from many slaves
  - It takes an average (can discard outliers).
  - It sends the required adjustment to the slaves
  - If master fails, can elect a new master to take over (not in bounded time)
The Berkeley Algorithm (1)

- The time daemon asks all the other machines for their clock values.

- Master uses Cristian’s algorithm to get time from many slaves.
The Berkeley Algorithm (2)

- The machines answer.
- Master uses Cristian's algorithm to get time from many slaves.
The time daemon tells everyone how to adjust their clock.
The Network Time Protocol (NTP)

Note: Arrows denote synchronization control, numbers denote strata.
NTP Protocol

- All modes use UDP
- Each message bears timestamps of recent events:
  - Local times of Send and Receive of previous message
  - Local times of Send of current message
- Recipient notes the time of receipt $T_i$ (we have $T_{i-3}$, $T_{i-2}$, $T_{i-1}$, $T_i$)
- In symmetric mode there can be a non-negligible delay between messages
For each pair of messages between two servers, NTP estimates an offset \( o \), between the two clocks and a delay \( d_i \) (total time for the two messages, which take \( t \) and \( t' \))

\[
T_{i-2} = T_{i-3} + t + o \quad \text{and} \quad T_i = T_{i-1} + t' - o
\]

This gives us (by adding the equations):

\[
d_i = t + t' = T_{i-2} - T_{i-3} + T_i - T_{i-1}
\]

Also (by subtracting the equations)

\[
o = o_i + (t' - t)/2 \quad \text{where} \quad o_i = (T_{i-2} - T_{i-3} + T_{i-1} - T_i)/2
\]

Using the fact that \( t, t'>0 \) it can be shown that

\[
o_i - d_i/2 \leq o \leq o_i + d_i/2
\]

Thus \( o_i \) is an estimate of the offset and \( d_i \) is a measure of the accuracy.

NTP servers filter pairs \( <o_i, d_i> \), estimating reliability from variation, allowing them to select peers. Select \( o_i \) that corresponds to the minimum value \( d_i \) to estimate \( o \).

Accuracy of 10s of milliseconds over Internet paths (1 on LANs)
How To Change Time

- Can’t just change time
  - Why not?

- Change the update rate for the clock
  - Changes time in a more gradual fashion
  - Prevents inconsistent local timestamps
Real synchronization is imperfect.
Clocks never exactly synchronized.
Often inadequate for distributed systems
  might need totally-ordered events
  might need millionth-of-a-second precision
Today

- Understanding time in distributed systems
- Synchronizing physical clocks (real clocks)
- Logical Time/Lamport Clock
Lamport’s Logical Clocks (1)

The "happens-before" relation $\rightarrow$ can be observed directly in two situations:

- If $a$ and $b$ are events in the same process, and $a$ occurs before $b$, then $a \rightarrow b$ is true.

- If $a$ is the event of a message being sent by one process, and $b$ is the event of the message being received by another process, then $a \rightarrow b$
Global logical time

Definition ($\rightarrow$): We define $e \rightarrow e'$ using the following rules:

- Local ordering: $e \rightarrow e'$ if $e \rightarrow_i e'$ for any process $i$
- Messages: $\text{send}(m) \rightarrow \text{receive}(m)$ for any message $m$
- Transitivity: $e \rightarrow e''$ if $e \rightarrow e'$ and $e' \rightarrow e''$

We say $e$ “happens before” $e'$ if $e \rightarrow e'$
Concurrency

- → is only a partial-order
  - Some events are unrelated

- Definition (concurrency): We say e is concurrent with e’ (written e || e’) if neither e → e’ nor e’ → e
Lamport logical clocks

- Lamport clock $L$ orders events consistent with logical “happens before” ordering
  - If $e \rightarrow e'$, then $L(e) < L(e')$
- But not the converse
  - $L(e) < L(e')$ does not imply $e \rightarrow e'$
- Similar rules for concurrency
  - $L(e) = L(e')$ implies $e \parallel e'$ (for distinct $e, e'$)
  - $e \parallel e'$ does not imply $L(e) = L(e')$
- i.e., Lamport clocks arbitrarily order some concurrent events
Lamport’s algorithm

- Each process $i$ keeps a local clock, $L_i$
- Three rules:
  1. At process $i$, increment $L_i$ before each event
  2. To send a message $m$ at process $i$, apply rule 1 and then include the current local time in the message: i.e., $send(m,L_i)$
  3. To receive a message $(m,t)$ at process $j$, set $L_j = max(L_j,t)$ and then apply rule 1 before time-stamping the receive event
- The global time $L(e)$ of an event $e$ is just its local time
  - For an event $e$ at process $i$, $L(e) = L_i(e)$
Lamport’s Logical Clocks (2)

- Three processes, each with its own clock.
  The clocks run at different rates.
Lamport’s algorithm corrects the clocks.
Lamport’s Logical Clocks

- The positioning of Lamport’s logical clocks in distributed systems: typically in the middleware layer

![Diagram of Lamport's Logical Clocks]

- Application layer
  - Application sends message
  - Message is delivered to application

- Middleware layer
  - Adjust local clock and timestamp message
  - Adjust local clock
  - Middleware sends message
  - Message is received

- Network layer
Many systems require a total-ordering of events, not a partial-ordering.

Use Lamport’s algorithm, but break ties using the process ID:

\[ L(e) = M \times L_i(e) + i \]

- \( M \) = maximum number of processes
Vector Clocks

- Vector clocks overcome the shortcoming of Lamport logical clocks
  - $L(e) < L(e')$ does not imply $e$ happened before $e'$
- Goal
  - Want ordering that matches causality
  - $V(e) < V(e')$ if and only if $e \rightarrow e'$
- Method
  - Label each event by vector $V(e) \ [c_1, c_2 \ldots, c_n]$
    - $c_i = \# \text{ events in process } i \text{ that causally precede } e$
Vector Clock Algorithm

- Initially, all vectors \([0,0,\ldots,0]\)
- For event on process \(i\), increment own \(c_i\)
- Label message sent with local vector
- When process \(j\) receives message with vector \([d_1, d_2, \ldots, d_n]\):
  - Set local each local entry \(k\) to \(\max(c_k, d_k)\)
  - Increment value of \(c_j\)
Vector Clocks

- Vector clocks overcome the shortcoming of Lamport logical clocks
  - $L(e) < L(e')$ does not imply $e$ happened before $e'$
- Vector timestamps are used to timestamp local events
- They are applied in schemes for replication of data
Vector Clocks

- At $p_1$
  - $a$ occurs at $(1,0,0)$; $b$ occurs at $(2,0,0)$; piggyback $(2,0,0)$ on $m_1$
- At $p_2$ on receipt of $m_1$ use $\text{max} \ ((0,0,0), (2,0,0)) = (2,0,0)$ and add 1 to own element = $(2,1,0)$
- Meaning of $=$, $\leq$, $\text{max}$ etc for vector timestamps
  - compare elements pairwise
Vector Clocks

- Note that $e \rightarrow e'$ implies $L(e) < L(e')$. The converse is also true.

- Can you see a pair of parallel events?
  - $c \parallel e$ (parallel) because neither $V(c) \leq V(e)$ nor $V(e) \leq V(c)$. 
Figure 14.6
Lamport timestamps for the events shown in Figure 14.5
Figure 14.7
Vector timestamps for the events shown in Figure 14.5
Important Lessons

- Clocks on different systems will always behave differently
  - Skew and drift between clocks

- Time disagreement between machines can result in undesirable behavior

- Two paths to solution: synchronize clocks or ensure consistent clocks

- Clock synchronization
  - Rely on a time-stamped network messages
  - Estimate delay for message transmission
  - Can synchronize to UTC or to local source
Announcements

- **PA2**
- **Midterm**
  - In the process of making it.
  - Written exam: take home exam. Honor code. Only textbook and lecture notes. No discussion. No Internet. Will give you about 1.5 days.
  - Programming part: we will give you more time. Can use the Internet, but no discussion.
- **PA3**