CSci 5105

Introduction to Distributed Systems

Synchronization
Last Time

• Naming
• Fundamentals: binding, resolution, naming vs. directory service
Time

• Why is time important in computer systems?

• Why is it hard in Distributed Systems?
Clock Synchronization Problem

- Clocks tick at different rate, skew.
- Interrupt fires every $K$ msec; clock updates in a register
  - may fire $K'$

![Diagram showing clock synchronization with different slopes representing fast, perfect, and slow clocks in relation to UTC time.](image_link)
Clock Synchronization Example

- Makefile example
- What happens?
Solution Options

• Everyone’s clock may be different
• Want notion of a single time
• Two options:
Global time: Network Time Protocol

- Time Server B has correct time
- All other serves get in synch with it
- Server A must take what into account?
  - Delay
    \[ \lambda = \frac{[(T2-T1)+(T4-T3)]}{2} \] (one-way delay)
    \[ \theta = \frac{[(T2-T1) + (T3-T4)]}{2} \] (offset)
NTP clock filter algorithm

The most accurate offset $\theta_0$ is measured at the lowest delay $\delta_0$ (apex of the wedge scattergram)

- The $\delta_0$ is estimated as the minimum of the last eight delay measurements and $(\theta_0, \delta_0)$ becomes the peer update

$$
\theta = \frac{1}{2}[(T_2 - T_1) + (T_3 - T_4)]
$$

$$
\delta = (T_4 - T_1) - (T_3 - T_2)
$$
NTP in practice

• Benefit of filtering algorithm
• Error: 724 \(\mu s\) down to 192 \(\mu s\)
NTP

- Challenge: if A was faster than B
  - A would have to set its clock backward
  - Set clock backward and compile .c file!

- Slowly adjust its clock backward
- 1 interrupt every 10 msec
  + 9 msec to clock
The Berkeley Algorithm

No accurate global clock: select time daemon which computes average (ignoring delay: could add back in); change physical clock value
Do we need global (real) time stamps?

- Real clocks are hard to keep in synchrony
- Absolutely needed for real-time
- Maybe we can live with something weaker ...

Makefile:
- `output.c` (logical time $L$)
- `output.o` (logical time $L'$)
- $L' > L$
Lamport’s Logical Clocks

• The "happens-before" relation → 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 $a \rightarrow b$
Causal Ordering

- The ordering of events is really a partial ordering
  - Transitive $a \rightarrow b$ and $b \rightarrow c$ then $a \rightarrow c$
  - Anti-symmm $a! \rightarrow b$ and $b! \rightarrow a$ then $a$, $b$ concurrent
  - Reflexive $a \rightarrow a$

Total
all $a,b$: either $a \rightarrow b$ or $b \rightarrow a$
Example
Example

Problem?
Quantifying Logical Clocks

• Assign clock values (#s) to events
• If $a \rightarrow b$
  – $C(a) < C(b)$ for all events $a, b$

• Note: Says nothing about order of other events
Time Progression

Computing $C_i$ for process $P_i$

1. Before executing an event $P_i$ sets $C_i \leftarrow C_i + 1$

2. When process $P_i$ sends a message $m$ to $P_j$, it sets $m$’s timestamp $ts(m)$ equal to $C_i$

3. Upon the receipt of a message $m$, process $P_j$ adjusts its own local counter:

   $$C_j \leftarrow \max\{C_j , ts(m)\}$$
Lamport’s Logical Clocks

(a) P₁, P₂, P₃

(b) P₁, P₂, P₃

P₁ adjusts its clock

P₂ adjusts its clock

m₁

m₂

m₃

m₄
Partial vs. Total Order

- Basic lamport clocks give a partial order
  - Many events happen “concurrently”
  - $C(a) < C(b)$ does not imply $a \rightarrow b$

- But sometimes a total order is more convenient
  - e.g., commit operations to a database
  - deposit, withdrawal

- Tie-breakers: concatenate unique PID ($p1 > p2 > ...$)
  - E.g. $C(a, P1) < C(b, P2)$ does imply $a \rightarrow b$
Lamport Clocks

• Cannot solve total order
  – For any two events, \( a \rightarrow b \) or \( b \rightarrow a \)

• BUT can guarantee something weaker
  Everyone acts on messages in the same order
Using Lamport Clocks: Totally Ordered Multicasting

- Problem?

Deposit $100

Update 1

Replicated database

AddInterest 10%

Update 2

Update 1 is performed before update 2

Update 2 is performed before update 1
Can solve ...

• Multicast acknowledgement
• Order events in the queue
• Execute event at top of queue in time order
  – Only if received acknowledgement to the event
Vector Clocks

Want $C(b) < C(a) \Rightarrow a \leftarrow b$ (and vice-versa)
Vector Clocks

Vector clocks are constructed by letting each process $P_i$ maintain a vector $VC_i$ with the following two properties:

1. $VC_i[i]$ is the number of events that have occurred so far at $P_i$. In other words, $VC_i[i]$ is the local logical clock at process $P_i$.

2. If $VC_i[j] = k$ then $P_i$ knows that $k$ events have occurred at $P_j$. It is thus $P_i$’s knowledge of the local time at $P_j$. 
Multicast Using Vector Clocks

1. Before executing an event \( P_i \) executes
   \[ VC_i[i] \leftarrow VC_i[i] + 1 \]

2. When process \( P_i \) sends a message \( m \) to \( P_j \), it sets
   \( m \)'s (vector) timestamp \( ts(m) \) equal to \( VC_i \)

3. Upon the receipt of a message \( m \), process \( P_j \)
   adjusts its own vector by setting
   \[ VC_j[i] \leftarrow \max\{VC_j[i], ts(m)[i]\} \text{ for each } i \]

4. If message is out-of-order, queue \( m \)!
   there exists a \( k, k \neq i \), for which \( VC_j[k] \neq m[k] \)
Enforcement

$VC_0 = (1,0,0)$  $VC_0 = (1,1,0)$

$VC_1 = (1,1,0)$

$VC_2 = (0,0,0)$  $VC_2 = (1,0,0)$  $VC_2 = (1,1,0)$
Next Time

Next topic: Mutual Exclusion

Read Chapter 6 TVS