Lecture 17: Time, Clocks and Event Ordering

CSC 469H1F / CSC 2208H1F
Fall 2007
Angela Demke Brown
Time in Distributed Systems

• Each machine maintains its own time
  • No global shared clock

• Consider `make` program

```c
myprogram: myprogram.c
  gcc -o myprogram myprogram.c
```

• Each `target` has a list of files on which it depends
• `make` compares timestamps on target, dependencies
• If target is older than some file that it depends on, then target is re-built
• Unambiguous on single computer
• What if timestamps are assigned on different machines?
Distributed Edit/Make

- myprogram.c saved

- Looks like myprogram doesn’t have to be recompiled
• Typical computer timer is a precisely-machined quartz crystal
  • Oscillates at a well-defined frequency when kept under tension
  • Freq depends on tension, kind of crystal, cut
• 2 associated registers, “counter” and “holding”
  • Counting reg decremented by one on each oscillation
  • When zero, interrupt is generated (tick) and counter is reloaded from holding
• Can’t guarantee that two crystals (in two different machines) oscillate at exactly the same frequency
  • Leads to clock skew over time
Clock synchronization

• Simple algorithm:
  • Time server maintains global notion of time
  • Each machine periodically contacts time server asking for current global time
    • How often depends on maximum drift rate ($\rho$) of local clocks, and maximum allowed difference ($\delta$) between two clocks $\rightarrow \frac{\delta}{2\rho}$ secs
  • Machine updates local time with global time

• Problems?
  • Time can’t run backward
  • Transmission delay
Basic “Message Passing” Model

- A collection of $n$ processes
- A process executes a sequence of events
  - Local computation
  - Sending a message
  - Receiving a message
Logical Time in Distributed Systems

• Time gives us a reference with which to order events
  • Need not be consistent with external “real” time

• How do we define when one event occurs “before” another?

• Intuition: event $A$ occurs before event $B$ if $A$ could have influenced $B$
  • A “causal” definition
The “Happens Before” Relation

- Given two events $A$ and $B$, $A \Rightarrow B$ if
  - $A$ and $B$ are executed at the same process, and $A$ occurs before $B$
  - $A = \text{send}(m)$ and $B = \text{receive}(m)$ for some message $m$
  - There is an event $C$ such that $A \Rightarrow C$ and $C \Rightarrow B$
Observing “Happens Before” Relation

• Associate with each event a *logical timestamp* \( T \) such that:

\[
\text{If } A \Rightarrow B \text{ then } T(A) < T(B).
\]

• Algorithm to achieve it [Lamport]:
  • \( i \)-th process keeps a non-negative integer counter \( T_i \), initially 0
  • When \( i \)-th process performs computation event, \( T_i \leftarrow T_i + 1 \)
  • When \( i \)-th process sends msg \( m \), it computes \( T_i \leftarrow T_i + 1 \) and appends \( T(m) \leftarrow T_i \) to \( m \)
  • When \( i \)-th process receives msg \( m \), \( T_i \leftarrow \max\{T_i, T(m)\} + 1 \)
  • For event \( A \) at \( i \)-th process, define \( T(A) = T_i \) computed during \( A \)
Example of Lamport's Algorithm

More Accurate Logical Clocks

• Suppose we want a logical timestamp \( T \) such that:
\[
A \Rightarrow B \text{ if and only if } T(A) < T(B).
\]

• Algorithm to achieve it [Mattern; Fidge]:
  • \( i \)-th process keeps a vector \( T_i \) with \( n \) elements
    • Each element \( T_i[j] \) is a non-negative integer counter, initially 0
  • When \( i \)-th process performs any event, \( T_i[i] \leftarrow T_i[i] + 1 \)
  • When \( i \)-th process sends \( m \), it also appends \( T(m) \leftarrow T_i \) to \( m \)
  • When \( i \)-th process receives \( m \), it also computes
    \[
    T_i[j] \leftarrow \max\{T_i[j], T(m)[j]\} \text{ for each } j \neq i
    \]
  • For event \( A \) at \( i \)-th process, define \( T(A) = T_i \) computed during \( A \)
  • \( T(A) < T(B) \equiv [\forall j: T(A)[j] \leq T(B)[j] \land \exists j: T(A)[j] < T(B)[j]]\)
Example of Vector Clocks

```
<table>
<thead>
<tr>
<th></th>
<th>D</th>
<th>G</th>
<th>H</th>
<th>K</th>
<th>O</th>
<th>P</th>
<th>R</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6</td>
<td>7</td>
</tr>
<tr>
<td>2</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>4</td>
<td>4</td>
<td>5</td>
<td>5</td>
</tr>
<tr>
<td>3</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>3</td>
<td>3</td>
<td>3</td>
<td>3</td>
</tr>
</tbody>
</table>
```

```
<table>
<thead>
<tr>
<th></th>
<th>A</th>
<th>B</th>
<th>C</th>
<th>J</th>
<th>N</th>
<th>Q</th>
</tr>
</thead>
<tbody>
<tr>
<td>3</td>
<td>3</td>
<td>3</td>
<td>3</td>
<td>3</td>
<td>3</td>
<td>3</td>
</tr>
<tr>
<td>2</td>
<td>3</td>
<td>3</td>
<td>3</td>
<td>3</td>
<td>3</td>
<td>3</td>
</tr>
<tr>
<td>1</td>
<td>3</td>
<td>3</td>
<td>3</td>
<td>3</td>
<td>3</td>
<td>3</td>
</tr>
</tbody>
</table>
```
Agreement Problems

- High-level goal: Processes in a distributed system reach agreement on a value
- Numerous problems can be cast this way
  - Transactional commit, atomic broadcast, ...

- The system model is critical to how to solve the agreement problem—or whether it can be solved at all
  - Failure assumptions
  - Timing assumptions
Failure Model

- A process that behaves according to its I/O specification throughout its execution is called **correct**.
- A process that deviates from its specification is **faulty**.
- There are many gradations of faulty. Two of interest are:
  - **Crash failures**: A faulty process halts execution prematurely.
  - **Byzantine failures**: No assumption about behavior of a faulty process.
Timing Model

- Specifies assumptions regarding delays between
  - execution steps of a correct process
  - send and receipt of a message sent between correct processes
- Again, many gradations. Two of interest are:

  - **Synchronous**
    - Known bounds on message and execution delays.
  - **Asynchronous**
    - *No assumptions* about message and execution delays (except that they are finite).
Consensus

• Each process begins with a value
• Each process can irrevocably decide on a value
• Up to $t < n$ processes may be faulty

• Problem specification
  • **Termination**: Each correct process decides some value.
  • **Agreement**: Correct processes do not decide different values.
  • **Validity**: If all processes begin with the same input, then any value decided by a correct process must be that input.