Lecture 9: Time, Clocks and Event Ordering

OK, it will be a complex hunt tonight: Let's synchronize our watches...
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
```

• When does a target get re-built?
• Unambiguous on single computer
• What if timestamps are assigned on different machines?
• Looks like myprogram should not get recompiled
Physical clocks

- Typical computer timer is a precisely-machined quartz crystal
  - Oscillates at a well-defined frequency when kept under tension
  - Frequency depends on tension, kind of crystal, cut
- 2 associated registers, “counter” and “holding”
  - Counting register decremented by one on each oscillation
  - When zero, interrupt is generated (called a tick)
  - On each clock tick, adds 1 to the time stored in memory, and counter is reloaded from “holding”
- Can’t guarantee that two crystals oscillate at exactly the same frequency
  ⇒ clock skew!
Clock synchronization

• Simple algorithm:
  • Time server maintains global notion of time
    • GPS receiver
    • Atomic clock
  • Each machine periodically contacts time server asking for current global time
  • Machine updates local time with global time
• Problems?
Cristian’s algorithm (1989)

1. Client P requests the time from server S
2. S responds with the time T from its own clock.
3. P sets its time to be $T + \text{RTT}/2$

- Assumes propagation delay is the same for send and receive
- Accuracy can be improved by making multiple requests and using the minimum RTT.
Berkeley Algorithm (1989)

1. A *master* is chosen by election
2. The *master* polls the *slaves* who reply with their time
3. The *master* observes RTT of the messages and estimates the time of each *slave*
4. The *master* averages the slave and own clock times
   • Ignores values that are far outside of the others
5. The *master* sends out the amount (positive or negative) that each *slave* must adjust its clock
Network Time Protocol

- Most commonly used protocol
- Time servers form a tree
  - Layers in tree are called *strata*
  - Stratum 0 are reliable external time source
  - Stratum 1 servers are directly connected to a time source
  - Other servers connected via network links, UDP
- Precision?
  - Millisecond in WAN
  - Microsecond in LAN

_Benjamin D. Esham* (bdesham) - Based upon *Ntp.png* by [en:User:Kim Meyrick].
Better Clock Synchronization

- Precision Time Protocol, PTP (<1 us accuracy)
  - Takes advantage of time sources in network switches and NICs
- Tree hierarchy (similar to NTP)
  - Uses *Best Master Clock (BMC)* algorithm to select *grandmaster device*

- But exact time is often less important than knowing how to order distributed events.
  - Which happened first?
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 $B$
  - It’s a “causal” definition
The “Happens Before” Relation

• Given two events $A$ and $B$, $A \implies B$ ($A$ happens before $B$) if:
  1. $A$ and $B$ are executed at the same process, and $A$ occurs before $B$
  2. $A = \text{send}(m)$ and $B = \text{receive}(m)$ for some message $m$
  3. There is an event $C$ such that $A \implies C$ and $C \implies B$

• No clear relationship $\Rightarrow$ concurrent events
Observing “Happens Before” Relation

• Associate with each event a *logical timestamp* $T$ such that:

  If $A \Rightarrow B$ then $T(A) < T(B)$.

• Logical clocks
  • Are *local* to each process/machine
  • Do not measure real time, only measure events
  • “Capture” the *happened-before* relation numerically
  • Provide a *partial ordering* (use logical clock values as timestamps)

• Algorithm to achieve it – *Lamport Clocks* [Leslie Lamport]
Lamport Clocks

• Assign *logical timestamp* $T$ such that:

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

• Algorithm

  1. The $i$-th process keeps a non-negative integer counter $T_i$, initially 0
  2. When $i$-th process performs computation event, $T_i \leftarrow T_i + 1$
  3. When $i$-th process sends msg $m$, it computes $T_i \leftarrow T_i + 1$ and appends $T(m) \leftarrow T_i$ to $m$
  4. 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 $A$

• Can use $LC(A)$ notation to refer to Lamport Clock for event $A$
Example of Lamport’s Algorithm
Lamport Clocks problem

- Lamport clock is used to create a partial causal ordering of events between processes.
- Given a logical Lamport clock:
  - If \( A \Rightarrow B \) then \( LC(A) < LC(B) \)
- The relation only goes one way
  - If an event A comes before another event B, then A’s logical clock is less than B’s logical clock.
- Given two Lamport clocks, can we order events?
  - i.e., if \( LC(A) < LC(B) \) then \( A \Rightarrow B \) ? **NO!**
- **Problem:** Lamport clocks do capture causal dependencies, but may imply more dependencies than truly exist.
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 – **Vector Clocks** [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 vector $T(m) \leftarrow T_i$ 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 $A$
    
    • $T(A) < T(B) \equiv \left[ \forall j: T(A)[j] \leq T(B)[j] \land \exists i: T(A)[i] < T(B)[i] \right]$
      
    • Sometimes use $VC(A)$ to refer to vector clocks.
Example of Vector Clocks

![Diagram showing an example of Vector Clocks with nodes labeled A to R and time steps 1 to 3, comparing VC(A) < VC(F), VC(D) < VC(N), VC(E) < VC(J), VC(J) < VC(R), VC(K) < VC(N), VC(I) < VC(P).]
Comparison

• Lamport clocks:
  • If $A \Rightarrow B$ then $LC(A) < LC(B)$
• Vector clocks:
  • $A \Rightarrow B$ if and only if $VC(A) < VC(B)$

• Lamport clocks: we have a guarantee that two causally-related events will have timestamps that reflect their order
  • However, just by looking at LC timestamps, we cannot conclude that there is a causal happens-before relationship!

• Vector clocks: both implications are true (including that if A’s vector clock is $< B$’s vector clock, they are causally related).
Distributed Algorithms

- Distributed system is composed of \( n \) processes
- A process executes a sequence of events
  - Local computation
  - Sending a message \( m \)
  - Receiving a message \( m \)
- A distributed algorithm is an algorithm that runs on more than one process.
Properties of Distributed Algorithms

• Safety
  • Means that some particular “bad” thing never happens.

• Liveness
  • Indicates that some particular “good” thing will (eventually) happen.
• Safety violation: if cars moving in opposite directions enter the lane at the same time.
• **Liveness:** does every car eventually get a chance to go through (i.e., make progress)?
• **Progress property** (opposite of starvation)
Properties of Distributed Algorithms

• Safety
  • Means that some particular “bad” thing never happens.

• Liveness
  • Indicates that some particular “good” thing will (eventually) happen.

• Timing and failure assumptions affect how we reason about these properties and what we can prove.
Timing Model

- Specifies assumptions regarding delays between:
  - execution steps of a correct process
  - send and receipt of a message sent between correct processes

- 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).

- Partial synchrony is more realistic in distributed system
Synchronous timing assumption

• Processes share a clock
• Timestamps mean something between processes
• Communication can be guaranteed to occur in some number of clock cycles
Asynchronous timing assumption

• Processes operate asynchronously from one another.
• No claims can be made about whether another process is running slowly or has failed.
• There is no time bound on how long it takes for a message to be delivered.
Partial synchrony assumption

- “Timing-based distributed algorithms”
- Processes have some information about time
  - Clocks that are synchronized within some bound
  - Approximate bounds on message-deliver time
  - Use of timeouts
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**

• Many gradations of faulty. Two of interest are:
  - **Fail-Stop failures**
    - A faulty process halts execution prematurely.
  - **Byzantine failures**
    - *No assumption* about behavior of a faulty process.
Errors as failure assumptions

• Specific types of errors are listed as failure assumptions
  • Communication link may lose messages
  • Link may duplicate messages
  • Link may reorder messages
  • Process may die and be restarted
Fail-Stop failure

• A failure results in the process, \( p \), stopping
  • Also referred to as crash failure
  • \( p \) works correctly until the point of failure
• \( p \) does not send any more messages
• \( p \) does not perform actions when messages are sent to it
• Other processes can detect that \( p \) has failed
Fault/failure detectors

• A perfect failure detector
  • No false positives (only reports actual failures).
  • Eventually reports failures to all processes.

• Heartbeat protocols
  • Assumes partially synchronous environment
  • Processes send “I’m Alive” (“heartbeat”) messages to all other processes regularly
  • If process \( i \) does not hear from process \( j \) in some time  
    \[ T = T_{\text{delivery}} + T_{\text{heartbeat}} \]
    then it determines that \( j \) has failed
  • Depends on \( T_{\text{delivery}} \) being known and accurate
Other Failure Models

• We can classify some of the likely failure modes that lie between crash and Byzantine
  • Omission failure
    • Process fails to send messages, to receive incoming messages, or to handle incoming messages
  • Timing failure
    • process’s response lies outside specified time interval
  • Response failure
    • Value of response is incorrect
Byzantine failure

- Process $p$ fails in an arbitrary manner.
- $p$ is modeled as a malevolent entity
  - Can send the messages and perform the actions that will have the worst impact on other processes
  - Can collaborate with other “failed” processes
- Common constraints on Byzantine assumption
  - Incomplete knowledge of global state
  - Limited ability to coordinate with other Byzantine processes
  - Restricted to polynomial computation (i.e., assume $P \neq NP...$)