Chapter 6
Synchronization (1)
Plan

- Clock synchronization in distributed systems
  - Physical clocks
  - Logical clocks
- Ordered multicasting
  - JGroups
- Mutual exclusion
- Election
Time

- We have been somewhat careful to avoid talking about time until now
- But, in distributed system we need practical ways to deal with time
  - E.g., we may need to agree that update A occurred ‘before’ update B
  - Or offer a “lease” on a resource that expires ‘at’ time 10:10:01.50
  - Or guarantee that a time critical event will reach all interested parties ‘within’ 100ms
But what does Time “mean”?

- Time on a machine’s local clock
  - But was it set accurately?
  - And could it drift, e.g. run fast or slow?
  - What about faults, like stuck bits?
- Time on a global clock?
  - E.g., with GPS receiver
  - Still not accurate enough to determine which events happen before other events
- Or could try to agree on time…
Basic approaches

- Physical time
  - Synchronize local clocks (internal synchronization)
  - Synchronize with external source (external synchronization)

- Logical time
  - Avoid absolute statements about time
Computer Clocks

- Typical computer has \textit{timer} circuit for keeping tracks of time
  - Quartz crystal that \textit{oscillate} at a well-defined frequency
  - \textit{Counter} decremented each oscillation
  - \textit{Holding register} incremented when counter reaches zero
- Time always goes forwards on a single computer…
  - But not when compared across distributed nodes
- Crystals oscillate at slightly different frequency \textit{-> clock skew}
Computer Clocks

Difference between time on M2 and M1 goes from > 0 to < 0
Physical Time

• How to get absolute physical time on computer?
  – Atomic clocks are expensive…

• Universal Coordinated Time (UTC) sources often broadcast UTC second starts
  – E.g., WWV, Fort Collins, Colorado
  – Accuracy of ~ 10 msec (10 x 10^{-3} sec)

• Global Positioning System
  – Claimed error < 60 nsec for (60 x 10^{-9} sec)
Clock Synchronization Algorithms

- Generally have time server
  - Needs external time source
    - WWV, GPS, ninja, administrator, …

- System model
  - Timers work within a maximum drift rate, $\rho$
    \[ 1 - \rho \leq \frac{dC}{dt} \leq 1 + \rho \]
  - To guarantee that clocks do not drift more than $\delta$ seconds apart, synchronize at least every $\Delta t$ seconds:
    \[ \Delta t \cdot 2\rho \leq \delta \Rightarrow \Delta t \leq \frac{\delta}{2\rho} \]
Cristian’s Algorithm

- A’s offset relative to B, \( \theta \), is given by

\[
dT_{\text{req}} \approx dT_{\text{res}} \Rightarrow \theta \approx \left( T_3 + \frac{(T_2 - T_1) + (T_4 - T_3)}{2} \right) - T_4 = \frac{(T_2 - T_1) + (T_3 - T_4)}{2}
\]

- \( \theta \geq 0 \) : speed up A’s (virtual) clock towards B’s
- \( \theta < 0 \) : slow down A’s (virtual) clock towards B’s
Network Time Protocol

- The delay estimation, $\delta$, is given by $\delta = \frac{(T_2 - T_1) + (T_4 - T_3)}{2}$
  - Calculate $(\theta_A, \delta_A)$ and $(\theta_B, \delta_B)$ multiple (8) times
  - Choose $\theta$ for which $\delta$ is minimal as offset estimate

- Divide time servers into *strata*
  - External clock is stratum 0
  - Stratum A server adjust their time according to stratum B server if $B < A$, then becomes stratum B+1 server
The Berkeley Algorithm

- Time daemon/server is actively requesting current time difference from machines
  - Calculates average and requests machines to adjust clock
- Works if no interface with external machine with physical time synchronization
Clock Synchronization in Wireless Networks

• Challenges
  – Most machines cannot contact each other directly
  – Large overhead in multihop routing
  – Cannot deploy fixed time servers
  – Need to optimize for energy consumption
Reference Broadcast Synchronization (RBS)

Variable:
- Context switches, system call overhead
- Media Access Control protocol

Constant:
- Message preparation
- Critical path

Delivery time to app.

$\approx 0$

Sufficiently constant:
- Can timestamp early
Reference Broadcast Synchronization (RBS)

- Assume we know clock skew and have corrected for it
  - I.e., clock offset can be regarded as constant
- Algorithm
  - Transmitter broadcasts $m$ reference packets
  - Each of $n$ receivers note when reference was observed according to local clock
  - Receivers exchange observations
  - Calculate pairwise offset as $\text{Offset}[i,j] = \frac{1}{m} \sum_{k=1}^{m} (T_{i,k} - T_{j,k})$
  - Where
    - $T_{i,k}$ is time of receipt of message $k$ at node $i$
Logical Time

• The Berkeley and RBS algorithms do not necessarily give any correspondence to UTC
  – But they do give correspondence to computer clock
• A lot of times this is not necessary
  – If we just want to know if event $a$ happened ”before” or ”after” event $b$
Lamport’s Approach

• Leslie Lamport suggested that we should reduce time to its basics
  – Cannot order events according to a global clock
    • None available…
  – Can use logical clock
    • Time basically becomes a way of labeling events so that we may ask if event A happened before event B
    • Answer should be consistent with what could have happened with respect to a global clock
      – Often this is what matters
Drawing time-line pictures:
Drawing time-line pictures:

- A, B, C and D are “events”.
  - Could be anything meaningful to the application
    - microcode, program code, file write, message handling, …
  - So are snd(m) and rcv(m) and deliv(m)
- What ordering claims are meaningful?
Drawing time-line pictures:

- A happens before B, and C before D
  - “Local ordering” at a single process
  - Write $A \rightarrow^p B$ and $C \rightarrow^q D$
Drawing time-line pictures:

- \( \text{snd}_p(m) \) also happens before \( \text{rcv}_q(m) \)
  - “Distributed ordering” introduced by a message
  - Write \( \text{snd}_p(m) \xrightarrow{M} \text{rcv}_q(m) \)
Drawing time-line pictures:

- A happens before D
  - Transitivity: A happens before \(\text{snd}_p(m)\), which happens before \(\text{rcv}_q(m)\), which happens before D
Drawing time-line pictures:

- B and D are concurrent
  - Looks like B happens first, but D has no way to know. No information flowed…
The Happens-Before Relation

- We’ll say that “A happens-before B”, written $A \rightarrow B$, if
  - 1) $A \rightarrow^P B$ according to the local ordering, or
  - 2) $A$ is a snd and $B$ is a rcv and $A \rightarrow^M B$, or
  - $A$ and $B$ are related under the transitive closure of rules 1. and 2.

- Thus, $A \rightarrow D$

- So far, this is just a mathematical notation, not a “systems tool”
  - A new event seen by a process happens logically after other events seen by that process
  - A message receive happens logically after a message has been sent
Logical clocks

• A simple tool that can capture parts of the happens before relation

• First version: uses just a single integer
  – Designed for big (64-bit or more) counters
  – Each process $p$ maintains $C_p$, a local counter
  – A message $m$ will carry $C_m$
Rules for managing logical clocks

- When an event happens at a process $p$ it increments $C_p$
  - Any event that matters to $p$
  - Normally, also $snd$ and $rcv$ events (since we want receive to occur “after” the matching send)

- When $p$ sends $m$, set
  - $C_m = C_p$

- When $q$ receives $m$, set
  - $C_q = \max(C_q, C_m) + 1$
Time-line with LT annotations

- $C(A) = 1$, $C(snd_p(m)) = 2$, $C(m) = 2$
- $C(rcv_q(m)) = \max(1,2)+1 = 3$, etc…
Logical clocks

• If A happens before B, A→B, then C(A)<C(B)
  – A→B : A = E0 →… →En = B, where each pair is ordered either by →ₚ or →ₘ
  • LT associated with these only increase

• But converse might not be true:
  – If C(A)<C(B) can’t be sure that A→B
  – This is because processes that don’t communicate still assign timestamps and hence events will “seem” to have an order
Can we do better?

• One option is to use *vector* clocks
• Here we treat timestamps as a vector
  – One counter for each process
• Rules for managing vector times differ from what we did with logical clocks
Vector clocks

- Clock is a vector: e.g. VC(A)=[1, 0]
  - We’ll just assign p index 0 and q index 1
  - Vector clocks require either agreement on the numbering/static membership, or that the actual process id’s be included with the vector

- Rules for managing vector clock
  - When event happens at p, increment VC_p[index_p]
    - Normally, also increment for snd and rcv events
  - When sending a message, set VC(m)=VC_p
  - When receiving, set VC_q=max(VC_q, VC(m))
    - Where “max” is max on components of vector
    - VC(m)[index_p]-1 events causally preceded m at p
Time-line with VT annotations

Could also be [1,0] if we decide not to increment the clock on a snd event. Decision depends on how the timestamps will be used.
Rules for comparison of VCs

- We’ll say that $VC_A \leq VC_B$ if
  - $\forall i, VC_A[i] \leq VC_B[i]$
- And we’ll say that $VT_A < VT_B$ if
  - $VC_A \leq VC_B$ but $VC_A \neq VC_B$
  - That is, for some $i$, $VC_A[i] < VC_B[i]$
- Examples?
  - $[2,4] \leq [2,4]$
  - $[1,3] < [7,3]$
  - $[1,3]$ is “incomparable” to $[3,1]$
Time-line with VC annotations

- VC(A)=[1,0]. VC(D)=[2,4]. So VC(A)<VC(D)
- VC(B)=[3,0]. So VC(B) and VC(D) are incomparable
Vector time and happens before

• If $A \rightarrow B$, then $\text{VC}(A) < \text{VC}(B)$
  – Write a chain of events from $A$ to $B$
  – Step by step the vector clocks get larger

• But also $\text{VC}(A) < \text{VC}(B)$ then $A \rightarrow B$
  – Two cases (formally by induction)
    • If $A$ and $B$ both happen at same process $p$ – all events seen by $p$ increments vector clocks
    • If $A$ happens at $p$ and $B$ at $q$, can trace the path back by which $q$ “learned” $\text{VT}(A)[p]$ since $q$ only updates $\text{VT}(A)[p]$ based on message receipt from, say, $q'$
      – If $q' \neq p$ trace further back

• (Otherwise $A$ and $B$ happened concurrently)
What can we use this for?

- May want different kinds of guarantees of when multicast messages are delivered at receivers
  - None
    - Delivery in arbitrary order (in practice FIFO)
    - E.g., many reads information
  - FIFO
    - Delivery in order of sending (with respect to one process)
    - E.g., only one process updates data but many reads
  - Causal
    - Delivery with respect to happens-before
    - E.g., efficient distributed locking
  - Total
    - Delivery with respect to a total/global order
    - E.g., for replicated updates

- I.e., ordered multi-casting
What can we use this for?

- May want different kinds of guarantees of when multicast messages are delivered at receivers:
  - None
  - Delivery in arbitrary order (in practice FIFO)
    - E.g., many reads information
    - FIFO
  - Delivery in order of sending (with respect to one process)
    - E.g., only one process updates data but many reads
    - Causal
  - Delivery with respect to happens-before
    - E.g., efficient distributed locking
    - Total
    - Delivery with respect to a total/global order
      - E.g., for replicated updates

- I.e., ordered multi-casting

We will, for now, assume that
- There are no failures
- We have stable process groups
Example: Totally Ordered Multicasting

- **Update 1**
  - Add 100$ to account
- **Update 2**
  - Add 1% interest to account
- **Update 1; Update 2**
  - $(1000$ + $100$)$ \times 1.01 = 1111$
- **Update 2; Update 1**
  - $1000$\times 1.01 + $100$ = 1110$
Totally Ordered Multicasting

- **Implementation**
  - Use a sequencer
    - Get a number and use that as sequence number for multicast
    - Receiving processes deliver in that order
  - Distributed
    - Each process attaches a Lamport clock to message
      - When received, acknowledge to everybody
    - Order messages in queue according to sequence number (and process number)
      - Deliver message at front when it has been acknowledged by all processes
      - Queues become identical
Enforcing Causal Communication

• Want to ensure that is $snd(m) \rightarrow snd(m^*)$ then $del(m) \rightarrow del(m^*)$
  – Use vector timestamps
  – Only increment when sending and receiving
  – Adjust when delivering
• Deliver a message $m$ from $i$ when
  – $m$ is the next message expected from $i$
    • i.e., $ts(m)[i] = VC_j[i] + 1$
  – All messages that $i$ has seen from other processes have been seen by us
    • i.e., $ts(m)[k] <= VC_j[k]$, $k \neq j$
Cost

• Orderings give increased guarantees and implementation cost
  – None < FIFO < causal < total

• None/FIFO ordering is easy to implement
  – Except for failures, transport layers take care of this
  – May be used efficiently

• Causal ordering harder to implement efficiently
  – But can be implemented very efficiently
  – May be used efficiently
    • Do not need to wait for own massages
    • Can deliver concurrent messages immediately

• Total ordering is hard to implement efficiently
  – Sometimes needed
  – Also hard to use efficiently
Summary

• We cannot achieve global, synchronized time in a distributed system

• Physical time
  – Can synchronize (with high probability) within a constant $\rho$

• Logical time
  – Advance time only as consequence of logical middleware events
  – Can be used to implement, e.g., ordered multicast