Today
- Coordination and agreement in group communication
- Consensus
Events and process states

- A distributed system – a collection P of N single-threaded processes w/o shared memory
  - Each process \( p_i \) has a state \( s_i \)
  - Each executes a series of actions – send, receive, transform state

- Events and clocks
  - Event – the execution of a single action
  - All events in a process can be placed in a total ordering \( \rightarrow_i \)
    - \( e \rightarrow_i e' \) iff \( e \) is an event that occurs after \( e' \) in \( p_i \)
  - History of a process \( p_i \): \( \text{history}(p_i) = h_i = < e_i^0, e_i^1, e_i^2, ... > \)
  - Computers have their own hardware-based clock, \( C_i \), which can be used to assign timestamps to events to \( h_i \)
    - \( C_i(t) = \alpha H_i(t) + \beta \) here \( H_i \) is the hardware clock of value at physical time \( t \)
Physical clocks

- Agreeing on time in distributed systems is not trivial
- Until the invention of the atomic clock, time was measured astronomically
- Universal Coordinated Time (UTC):
  - Based on # of transitions/sec of cesium 133 (Cs$^{133}$) atom
  - Currently, real time is as avg of ~50 cesium-clocks
  - A periodic leap second compensates for days getting longer
- UTC is broadcast through short wave radio & satellite; satellites can give an accuracy of about ±0.5 ms
- We want to distribute this to a bunch of machines
  - Each runs its own timer, keeping a clock $C_p(t)$ (t being UTC)
  - Ideally we want $C_p(t) = t$ for all processes, i.e. $dC/dt = 1$
Physical clocks

- However, $1 - r \leq \frac{dC}{dt} \leq 1 + r$
  - Frequency of clock at time $t - C'_p(t)$
  - Skew of clock, extent to which its frequency differs from that of a perfect clock $- C'_p(t) - 1$
  - Offset relative to a specific time $t$ $C_p(t) - t$
  - $r$ is the maximum drift rate of a clock (given by its manufacturer)

- Goal: Never let two clocks in any system differ by more than $d$ time units $\Rightarrow$ synchronize at least every $\frac{d}{2r}$ seconds
Clock synchronization

- Two modes – internal and external synchronization
  - Internal – The clocks in the set must agree within a bound $d$
  - External – The clocks must be accurate respect to a source of UTC time within a bound $d$
  - Setting the time back is never allowed $\Rightarrow$ smooth adjustments

- Internal synchronization
  - Pushing server – Berkeley algorithm
  - Let the time server scan all machines periodically, calculate an avg (accounting for rtt), and inform each machine how to adjust its time

```
<p>| | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>2:50</td>
<td>3:00</td>
<td>3:25</td>
</tr>
<tr>
<td>3:00</td>
<td>3:25</td>
<td>+15</td>
</tr>
<tr>
<td>+5</td>
<td>+15</td>
<td>-20</td>
</tr>
</tbody>
</table>
```
Clock synchronization

- **External synchronization**
  - Cristian’s method - pulling server
  - Every machine asks a time server for the accurate time at least once every $d/(2r)$ seconds
  - You need an accurate measure of round trip delay, including interrupt handling and processing incoming messages

- **Network Time Protocol**
  - External synchronization for the Internet
  - A number of servers, in a hierarchy with
    - Primary servers directly connected to time sources
    - Secondary servers synchronized with others servers
    - All organized in strata – lower levels, higher confidence
Logical clocks

- We typically assume clock synchronization is related to real time, not necessary
- We have seen (Berkeley algorithm) clocks can agree on a current time without this having to be the real time
- Actually,
  - Many times all that matters is that two nodes agree on the order of events
  - If two nodes do not shared events, i.e. they don’t interact, they don’t have to be in synch ➔ Logical clocks
Happened-before relationship

- The happened-before relation on the set of events in a distributed system:
  - HB1: If $a$ and $b$ are two events in the same process, and $a$ comes before $b$, then $a \rightarrow b$
  - HB2: If $a$ is the sending of a message, and $b$ is the event of receiving that message, then $a \rightarrow b$
  - HB3: If $a \rightarrow b$ and $b \rightarrow c$, then $a \rightarrow c$
Happened-before relationship – notes

- This introduces a partial ordering of events in a system with concurrently operating processes
  - If $x$ and $y$ happen in two processes that do not exchange messages, then neither $x \rightarrow y$ nor $y \rightarrow x$
  - $x$ and $y$ are concurrent

- What happen with communication through other channels? e.g., phone

- If $x \rightarrow y$, does it mean $x$ cause $y$?
Happened-before – an example
Lamport clock

- How to maintain a global view on system’s behavior that is consistent with the happened before relation?

- Attach a timestamp $C(e)$ to each event $e$, satisfying the following properties:
  - P1: If $a$ and $b$ are two events in the same process, and $a \rightarrow b$, then $C(a) < C(b)$
  - P2: If $a$ corresponds to sending a message $m$, and $b$ to the receipt of that message, then also $C(a) < C(b)$

- How to attach a timestamp to an event when there’s no global clock $\Rightarrow$ maintain a consistent set of logical clocks, one per process
Lamport clock

- Each process $p_i$ maintains a local counter $C_i$ and adjusts this counter according to the following rules:
  1. For any two successive events that take place within $p_i$, $C_i$ is incremented by 1
  2. When $p_i$ sends a message $m_i$, it includes a timestamp $\text{ts}(m) = C_i$
  3. Whenever $p_j$ receives $m$, $p_j$ adjusts its local counter $C_j$ to $\text{max}(C_j, \text{ts}(m))$; then executes step 1 before passing $m$ to the application

- Property 1 is satisfied by (1)
- Property 2 by (2) and (3)
- Note: To impose total ordering (instead of partial), attach process ID
Lamport timestamps – an example

![Diagram showing Lamport timestamps with physical time and process identifiers]

- **Physical time:**
  - 1
  - 2
  - 3
  - 4
  - 5

- **Processes:**
  - **$p_1$:**
    - 1
    - 2
    - 3
  - **$p_2$:**
    - 4
  - **$p_3$:**
    - 5

- **Messages:**
  - $m_1$: from process 2 to process 3
  - $m_2$: from process 4 to process 5
Example use – totally ordered multicast

- To guarantee that concurrent updates on a replicated database are seen in the same order everywhere:
  - \( P1 \) adds $100 to an account (initial value: $1000)
  - \( P2 \) increments account by 1%
  - There are two replicas

Result: in absence of proper synchronization: replica #1 \( \leftarrow \$1111 \), while replica #2 \( \leftarrow \$1110 \).
Totally ordered multicast

- **Solution:**
  - $p_i$ sends timestamped $msg_i$ to all others
  - Message is put in a local queue $queue_i$
  - Any incoming message at $p_j$ is queued in $queue_j$, by its timestamp, and ack to every other process
  - $p_j$ passes $msg_i$ to its application if:
    1. $msg_i$ is at the head of $queue_j$
    2. for each process $p_k$, there is a message $msg_k$ in $queue_j$ with a larger timestamp

- **Assume communication is reliable and FIFO ordered**

- **Also known as state machine replication**
  - Replicas follow the same transitions in the same finite state machine
Vector clocks

- With Lamport’s clocks – if $x \rightarrow y$, $C(x) < C(y)$, but if $C(x) < C(y)$, we can’t infer $x$ causally preceded $y$

- Vector clock for a system with $N$ processes – an array of $N$ integers
  - Processes piggyback vector timestamps on each message

- Rules for updating clocks
  - Just before $p_i$ sends a message $m$, it adds 1 to $V_{i[i]}$, and sends $V_i$ along with $m$ as vector timestamp $vt(m)$
  - When a $p_j$ receives a message $m$ that it received from $p_i$ with vector timestamp $ts(m)$, it
    1. updates each $V_{j[k]}$ to $\max\{V_{i[k]}, ts(m)[k]\}$ for $k = 1 \ldots N$
    2. increments $V_{j[j]}$ by 1
Vector clocks

- For process $p_i$ with vector $V_i[1..n]$,
  - $V_i[i]$ number of events that have taken place at process $p_i$
  - $V_i[j]$ number of events that $p_i$ knows have taken place at process $p_j$

- Comparing vector timestamps
  - $V = V'$ iff $V[j] = V'[j]$ for $j = 1 .. N$
  - $V \leq V'$ iff $V[j] \leq V'[j]$ for $j = 1 .. N$
  - $V < V'$ iff $V[j] < V'[j]$ for $j = 1 .. N$
Vector clocks – an example

Physical time

\( \mathbf{p}_1 \)

- \( (1,0,0) \) - \( a \)
- \( (2,0,0) \) - \( b \)
- \( m_1 \)

\( \mathbf{p}_2 \)

- \( (2,1,0) \) - \( c \)
- \( (2,2,0) \) - \( d \)
- \( m_2 \)

\( \mathbf{p}_3 \)

- \( (0,0,1) \) - \( e \)
- \( (2,2,2) \) - \( f \)
Causally ordered multicasting

- Ensure that a msg is delivered only if all causally preceding msgs have already been delivered
- Clock adjustment only when sending/receiving messages:
  - $p_i$ increments $V_{ij}$ only when sending a message
  - $p_j$ “adjusts” $V_j$ when receiving a message
- $p_j$ postpones delivery of $m$ until:
  - $ts(m)[i] = V_{ij} + 1$
    - $m$ is next msg $p_j$ was expecting from $p_i$
  - $ts(m)[k] \leq V_{jk}$ for $k \neq j$
    - $p_j$ has seen all msgs seen by $p_i$ when it sent the message
Causally ordered multicasting

- Suppose $P_j$ receives $m$ from $P_i$ with timestamp $ts(m)$
- $P_j$ postpones delivery of $m$ until:
  - $ts(m)[i] = VC_j[i] + 1$
  - $ts(m)[k] \leq VC_j[k]$ for $k \neq j$
Global state

- How can we determine a given property holds in a distributed system in execution?
  - With perfectly synchronized clock, easy
    - Take snapshot at 6:25:00PM CST
    - But there’s nothing like it
Global state

- **Why**
  - Garbage collection – an object is garbage if there are no longer any references anywhere in the system
  - Deadlock detection
  - Debugging

- **Some notes**
  - Each process can record the states that take place there \(- s_i^k\) – state of \(p_i\) right before \(kth\) event (so, \(s_i^0\) is the initial event)
  - To capture the state of the channel – process record the sending and received of all messages as part of their state – if sent but not yet received, then it’s in the channel
Global states, cuts and frontiers

- Global state – can be made of any set of state of all processes \( S = (s_1, s_2, \ldots, s_N) \)
  - *But which global states are meaningful?*
- Cut of a system’s execution – a subset of its global history (which is a union of prefixes of processes histories)
  \[ C = h_1^{C_1} \cup h_2^{C_2} \cup \ldots \cup h_N^{C_N} \]
Global states, cuts and frontiers

- The set of events $\{e_i^C : i = 1, 2, \ldots N\}$ is called the *frontier* of the cut.
Consistent cuts and global states

- A cut $C$ is consistent if, for each event it contains all the events that happened-before that event.
- A consistent global state is one that corresponds to a consistent cut.

Inconsistent cut

Consistent cut

Frontier: $<e_1^0, e_2^0>$

Frontier: $<e_1^2, e_2^2>$
Runs and linearization

- A run – a total ordering of all events in a global history that is consistent with each local history’s ordering
- A linearization or consistent run – an ordering of the events in a global history that is consistent with the happened-before relation on $H$
- All linearization (but not all runs) pass only through consistent global states
- A state $S'$ is reachable from a state $S$ if there is a linearization that passes through $S$ and then $S'$
Chandy and Lamport’s snapshot algorithm

- **Goal** – to record a consistent global state
- **Assumptions**
  - Algorithm records state locally at processes, it says nothing about collecting it
  - Neither channel nor processes fail; reliable, exactly-once comm.
  - Channels are unidirectional and provide FIFO ordering
  - Graph of processes and channels is strongly connected
  - Any process may initiate the global snapshot at any time (just send a marker on a non-existing channel)
  - Processes continue execution and communication while snapshot is taking place
Algorithm

Algorithm uses special *marker* messages and two rules

*Marker receiving rule for process* $p_i$
On receipt of a marker message at $p_i$ over channel $c$
If ($p_i$ has not yet recorded its state) it
  - records its process state now
  - records the state of $c$ as the empty set
  - turns on recording of messages arriving
    over other incoming channels
else
  $p_i$ records the state of $c$ as the set of messages it has received over $c$ since it saved its state
endif

*Marker sending rule for process* $p_i$
After $p_i$ has recorded its state, for each outgoing channel $c$
  - $p_i$ sends one marker message over $c$
    (before it sends any other message over $c$)
Example execution

Two processes trading widgets; \( p_1 \) sends order over \( c_2 \) for widgets at \$10\) per widget; later \( p_2 \) sends widgets over \( c_1 \).

Initial state; \( p_2 \) has already received an order for 5 widgets that’s about the send.

\( p_1 \) saves state in global state \( S_0 \) and sends a marker before sending a new order.

\( p_2 \) emits order for 5 widgets from previous request; system enters \( S_2 \).
Example execution

$p_1$ receives order and $p_2$ receives marker; $p_2$ saves its state <$50, 1995> and that of channel $c_2$ as empty; sends marker over $c_1$

$p_1$ receives marker over $c_1$ it records the state of that channel as the single message (five widgets) that has received after first recorded its state

Final recorded state: $p_1$: <$1000,0>, $p_2$: <$50,1995>, $c_1$:<(five widgets)>, $c_2$: <>

Note that this state differ from all the global states through which the system actually passed!

The snapshot algorithm selects a cut, and therefore a state, that is consistent; that is, for all $e_i \rightarrow e_j$, if $e_j$ is in the cut, then $e_i$ is too
Example use – totally ordered multicast

- To guarantee that concurrent updates on a replicated database are seen in the same order everywhere:
  - $P1$ adds $100$ to an account (initial value: $1000$)
  - $P2$ increments account by 1%
  - There are two replicas

Result: in absence of proper synchronization: replica #1 ← $1111$, while replica #2 ← $1110$. 
Totally ordered multicast

Solution:
- \( p_i \) sends timestamped \( msg_i \) to all others
- Message is put in a local queue \( queue_i \)
- Any incoming message at \( p_j \) is queued in \( queue_j \), by its timestamp, and ack to every other process
- \( p_j \) passes \( msg_i \) to its application if:
  1. \( msg_i \) is at the head of \( queue_j \)
  2. for each process \( p_k \), there is a message \( msg_k \) in \( queue_j \) with a larger timestamp

Assume communication is reliable and FIFO ordered
Also known as state machine replication
- Replicas follow the same transitions in the same finite state machine
Causally ordered multicasting

- Ensure that a msg is delivered only if all causally preceding msgs have already been delivered
- Clock adjustment only when sending/receiving messages:
  - $p_i$ increments $V_{ij}$ only when sending a message
  - $p_j$ “adjusts” $V_j$ when receiving a message
- $p_j$ postpones delivery of $m$ until:
  - $ts(m)[i] = V_{ij} + 1$
    - $m$ is next msg $p_j$ was expecting from $p_i$
  - $ts(m)[k] \leq V_j[k]$ for $k \neq j$
    - $p_j$ has seen all msgs seen by $p_i$ when it sent the message
Causally ordered multicasting

- Suppose $P_j$ receives $m$ from $P_i$ with timestamp $ts(m)$
- $P_j$ postpones delivery of $m$ until:
  - $ts(m)[i] = VC_j[i] + 1$
  - $ts(m)[k] \leq VC_j[k]$ for $k \neq j$
Summary

- Synchronization is about doing the right thing at the right time …
- What’s the right time?
  - An issue when you don’t share clocks
- What’s the right thing to do?
  - Who can access what when?
  - Who is in charge?