A Second Example

- Suppose you want to build chat room software
- You want all messages typed by all participants to show up on everyone’s screen in the same order
- Division of responsibilities:
  - Your software: most everything, except for...
  - Multicast
    - a single send(m) call causes message m to be delivered to multiple destinations
Reliable, Totally Ordered Multicast

- **multicast**: a single send(m) call causes message m to be delivered to multiple destinations
- **totally ordered**: roughly, there is a unique sorted order to the messages (less roughly, the ordering is determined by an antisymmetric, transitive, and total relation)
- **reliable**: if a correctly operating client displays message m before displaying message m’, then any other correctly operating client that displays m’ will first display m
RTOM

• We actually want more than this, in a practical setting
  – Liveness: all messages are eventually displayed
  – "Reasonableness": in normal operation, each message should be displayed promptly at all clients

• Some unreasonable (and possibly not-live) solutions:
  – Never show any messages
  – Choose a single client and show only its messages
  – We cycle in a fixed order among the clients
    • Show msg from A, then B, then ...Z, then A,...
  – Wait until all clients quit the chat, then sort the messages lexicographically and print them.

We're Going to Solve This Twice

• Method A:
  – Implement
  – run to find bugs
  – change to fix bugs
  – repeat

• Method B:
  – Let's consider the problem carefully
  – Then let's implement
First Try: The Straightforward Implementation

- When m-send(m) is invoked, immediately send it to each client (including yourself):

  ```
  foreach client c {
    net-send(c,m);
  }
  ```

- When a message m is received from the network, hand it up to the app (to display):

  ```
  deliver(m);
  ```

- What can (will) go wrong?

- Observation: receiver side timestamps are useless in solving this problem

Second Try: Sender timestamps

- Assume net-send() is reliable, and that no client crashes or has bugs

- On m-send(m):

  ```
  t = localClockTime();
  foreach client c {
    net-send(c,m,t);
  }
  ```

- When a message (m,t) is received from client s:

  ```
  put (m,t) in a sorted queue;
  while (there is a message in the queue) {
    deliver the message with the lowest timestamp;
    remove the delivered message from the queue;
  }
  ```

- Does it work?
Third Try

- Assume net-send() is reliable, and that no client crashes or has bugs

On m-send(m):

```plaintext
t = localClockTime();
for each client c {
    net-send(c, m, t);
}
```

- When a message (m, t) is received from client s:

```plaintext
    put (m, t) in a sorted queue;
    while (there is a message in the queue from each client) {
        deliver the message with the lowest timestamp;
        remove the delivered message from the queue;
    }
```

- Does it work?
  - Are you sure?
  - What assumption about what net-send() guarantees are required?
  - What other assumption is it making?
  - Why isn’t it an acceptable solution in practice?
Implementing RTOM

- RTOM has its own view of what the network is
  - The interface provided by lower layer networking software and/or hardware

- Assumed properties of that interface (RPO):
  - Reliability Assumption: Reliable
    - If A does a net-send(m,B), B will eventually receive m
      - Note: The delivery delay is finite but unpredictable
  - Ordering Assumption: Pair-wise ordered
    - If A does net-send(m,B) and later net-send(m’,B), m will be delivered to B before m’
      - Note: this property holds only “pairwise.” If A does net-send(m,B) then net-send(m’,C), there is no guarantee about the order of delivery of m and m’
Why Is This Not Trivial?

- Unpredictable delays in the network is enough

\[ t_0: \text{N0 sends; N0,N2 receive} \]
\[ t_1: \text{N1 sends; all receive} \]
\[ t_2: \text{N3 receives N0's message} \]

Essence of the Solution

- The problem is distributed
- Each node is going to make a decision, based entirely on information it has itself
  - It knows what it sent and what is has received
  - It doesn’t know (with complete accuracy) what any other node has sent or received
- The key property we need is that all nodes make consistent decisions
- To do that, we want them to:
  - Apply a deterministic function to...
  - Data that is enough alike that they get the same answer
The Function: Min( \{timestamps\} )

- If all nodes had the same set \( S \) of timestamps, and all
  made a decision, they'd make the same decision
  - That's good

- There's no way to know what set other nodes have
  - That's bad

- If, for any two sets, the two were either identical or one
  was a proper subset of the other, we'd be done
  - But, that isn't necessarily the case

One Important Aspect of the Solution

- Exploit (assumed) pairwise-ordered property of
  underlying network
What if someone doesn't send for while?

- If any of the incoming queues is empty, we can't deliver anything
- If there are messages in some queues, we'd like to be sure there will "soon" be messages in all
- One way:
  - If a client hasn't sent a message in the least T milliseconds, it must send a "I have no message" message
  - Problem with that?
- Another way:
  - Make sure that for each actual message sent by any client c, every other client sends a message shortly thereafter
  - "Acknowledgments"

Acknowledgments

Blue: data mcast
Red: ACK mcast
One Remaining Problem...

What happens?

Lamport clocks

- Each client has its own Lamport clock, with monotonically increasing timestamp $t_c$
- Every event is tagged with its timestamp
  - For us, events are m-send() invocations and message receptions
- When a local event occurs on node $c$ (m-send($m$) is invoked):
  - $t_c = t_c + 1$
- When a message with timestamp $t_m$ is received at $c$:
  - $t_c = \max(t_c, t_m) + 1$
Finally, the Implementation

- On m-send(m) at client s:
  
  \[ t_c = t_c + 1; \]
  
  foreach client c {
    net-send(c,m,t_c);
  }

- When \((m,t_m)\) is received at c:
  
  \[ t_c = \max(t_c, t_m) + 1; \]

  // broadcast an acknowledgement of m to everyone else
  if (the message received is not itself an ACK) {
    foreach client q {
      net-send(q,ACK(m,t_q);
    }
  }

  put \((m,t_m)\) in a sorted queue;
  while (the first non-ACK message in the queue has been ACK'ed by all clients) {
    deliver(though non-ACK message);
    remove that message and its ACKs from the queue;
  }

An Example

<table>
<thead>
<tr>
<th>Event</th>
<th>N0</th>
<th>N1</th>
<th>N2</th>
<th>N3</th>
</tr>
</thead>
<tbody>
<tr>
<td>Startup</td>
<td>((0,<em>,</em>))</td>
<td>((<em>,0,</em>))</td>
<td>((<em>,</em>,0))</td>
<td>((<em>,</em>,*))</td>
</tr>
<tr>
<td>N0 sends</td>
<td>((1,<em>,</em>))</td>
<td>((1,2,*))</td>
<td>((1,<em>,</em>))</td>
<td>((<em>,</em>,*))</td>
</tr>
<tr>
<td>N1 ACKs</td>
<td>((3,2,*))</td>
<td>((1,2,*))</td>
<td>((1,2,3,*))</td>
<td>((*,2,3))</td>
</tr>
<tr>
<td>N2 ACKs</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>N1 sends</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>N3 receives</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>N3 ACKs</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

All nodes have the same clock granularity, \(\Delta\), and N1's clock is the same as N0's.
An Example

<table>
<thead>
<tr>
<th>Event</th>
<th>N0</th>
<th>N1</th>
<th>N2</th>
<th>N3</th>
</tr>
</thead>
<tbody>
<tr>
<td>Startup</td>
<td>(0,<em>,</em>,*)</td>
<td>(<em>,</em>,<em>,</em>)</td>
<td>(<em>,</em>,0,*)</td>
<td>(<em>,</em>,*,0)</td>
</tr>
<tr>
<td>N0 sends</td>
<td>(1,<em>,</em>,*)</td>
<td>(1,2,<em>,</em>)</td>
<td>(1,<em>,2,</em>)</td>
<td>(1,<em>,</em>,0)</td>
</tr>
<tr>
<td>N1 ACKs</td>
<td>(3,2,<em>,</em>)</td>
<td>(1,2,<em>,</em>)</td>
<td>(1,2,3,*)</td>
<td>(1,<em>,</em>,0)</td>
</tr>
<tr>
<td>N2 ACKs</td>
<td>(4,2,3,*)</td>
<td>(1,4,3,*)</td>
<td>(1,2,3,*)</td>
<td>(1,2,3,4)</td>
</tr>
<tr>
<td>N1 sends</td>
<td>(6,5,3,*)</td>
<td>(1,5,3,*)</td>
<td>(1,5,6,*)</td>
<td>(1,<em>,</em>,0)</td>
</tr>
<tr>
<td>N3 receives</td>
<td>(6,5,3,*)</td>
<td>(1,5,3,*)</td>
<td>(1,5,6,*)</td>
<td>(1,5,3,7)</td>
</tr>
<tr>
<td>N3 ACKs</td>
<td>(8,5,3,7)</td>
<td>(1,8,3,7)</td>
<td>(1,5,8,7)</td>
<td>(1,5,3,7)</td>
</tr>
</tbody>
</table>

Two Last Things

- Is RPO realistic?
  - Does the Internet provide RPO guarantees?
  - Does a local Ethernet? A local 802.11 wireless?

- RTOM: What about this solution