CSE/EE 461: Introduction to Computer Communications Networks
Autumn 2007

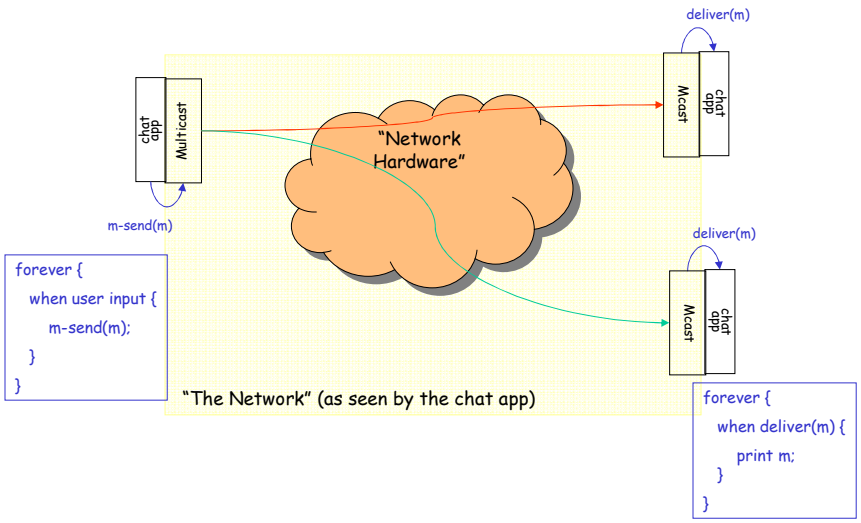
Module 1.5
Introduction – Reliable Multicast

John Zahorjan
zahorjan@cs.washington.edu
534 Allen Center

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

The Chat Room Application



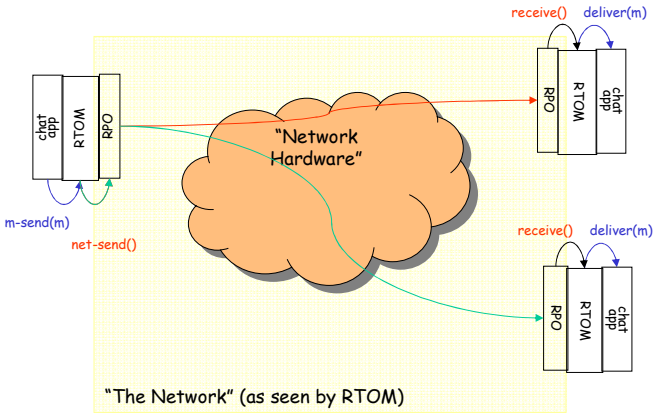
When Will That Implementation Work?

- Reliable, Totally Ordered Multicast (RTOM)
  - 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
  - (Also want liveness: all messages are eventually displayed)

Implementing RTOM

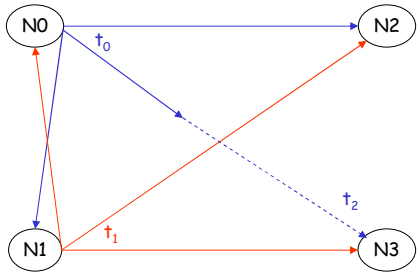
- RTOM has its own view of what the network is
  - The interface provided by lower layer networking software and/or hardware
- Assumed properties of that interface (RPO):
  - Reliability Assumption: Reliable
    - If A does a net-send(m,B), B will eventually receive m
      - Note: The delivery delay 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 deliver()'ed 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'

The Chat Room Application



Why Is This Not Trivial?

- Unpredictable delays in the network is enough



t_0 : N0 sends; N1,N2 receive

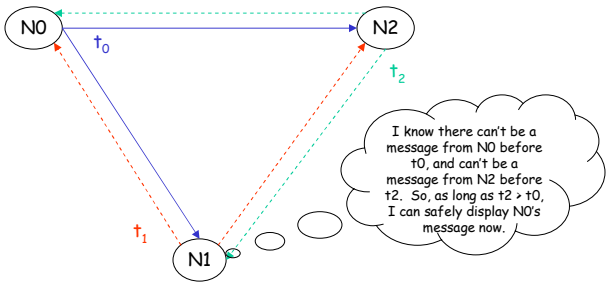
t_1 : N1 sends; all receive

t_2: N3 receives N0's message

Basic Idea of Solution

- Sort messages you've received, by time sent
  - Why time sent, rather than received?
    - Need the results of sorting to be the same on all nodes
- Display message only when you're sure there is no earlier message still on its way to you
  - RPO says that if you've seen a message from p sent at time t, there can't be a message from p sent before t still in the network
  - Make all nodes "acknowledge" each message, so that we have a constant supply of new info from each other node

Basic Idea In A Picture



- Unless t1 > t0 and t2 > t0, can't display N0's message (yet)
  - Suppose t2 < t0. N2 might still send a message at some time before t0.
- Therefore, can't use local time of send for these times, because of clock skew and drift

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_s is received at c:
  - $t_c = \max(t_c, t_s) + 1$

Finally, the Implementation

- On m-send(m) at client s:

```
t_s = t_s + 1;
foreach client c {
    net-send(c,m,t_s);
}
```

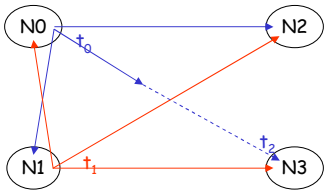
- When (m,t_s) is received at c:

```
t_c = max(t_c, t_s) + 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_c);
   }
}

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


An Example

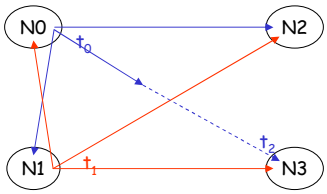


Vectors show what each node knows about the local time at all of the nodes. The algorithm does explicitly keep these vectors – the times for other nodes are in the messages in the queue.

Except for the first send from N0, we're assuming all other messages are received by all nodes, and that no two messages are ever in the network at the same time. (That last bit just for simplicity in constructing this example.)

Event	N0	N1	N2	N3
Startup	(0,*,*,*)	(*,0,*,*)	(*,*,0,*)	(*,*,*,0)
N0 sends	(1,*,*,*)	(1,2,*,*)	(1,*,2,*)	(*,*,*,0)
N1 ACKs	(3,2,*,*)	(1,2,*,*)	(1,2,3,*)	(*,2,*,3)
N2 ACKs				
N1 sends				
N3 receives				
N3 ACKs				

An Example



Vectors show what each node knows about the local time at all of the nodes. The algorithm does explicitly keep these vectors – the times for other nodes are in the messages in the queue.

Except for the first send from N0, we're assuming all other messages are received by all nodes, and that no two messages are ever in the network at the same time. (That last bit just for simplicity in constructing this example.)

Event	N0	N1	N2	N3
Startup	(0,*,*,*)	(*,0,*,*)	(*,*,0,*)	(*,*,*,0)
N0 sends	(1,*,*,*)	(1,2,*,*)	(1,*,2,*)	(*,*,*,0)
N1 ACKs	(3,2,*,*)	(1,2,*,*)	(1,2,3,*)	(*,2,*,3)
N2 ACKs	(4,2,3,*)	(1,4,3,*)	(1,2,3,*)	(*,2,3,4)
N1 sends	(6,5,3,*)	(1,5,3,*)	(1,5,6,*)	(*,5,3,6)
N3 receives	(6,5,3,*)	(1,5,3,*)	(1,5,6,*)	(1,5,3,7)
N3 ACKs	(8,5,3,7)	(1,8,3,7)	(1,5,8,7)	(1,5,3,7)

One Last Thing: Layering Is Natural