Atomic multicast 1/4

➤ Consider implementing atomic multicast

Want:

- ➤ Multicast message received by all destinations or none
- ➤ Non-delivery only if sender fails during protocol

Assume:

- > Implement non-uniform dynamic atomic multicast
 - → message received by all destinations that do not fail or none
- ➤ Reliable point-to-point communication

Distributed Systems

Atomic multicast 2/4

Easy!

- ➤ Simply have sender send message to all destinations in turn
- ➤ What if sender fails while sending?
 - → some destinations receive message and others not!

Atomic multicast 3/4

So,

At sender:

➤ Send message to all sites where there is a destination process

At receiving site:

- ➤ If message has not already been received
 - \rightarrow send copy of message to all other sites where there is a destination process
 - → deliver message to local destination processes

Distributed Systems

.

Atomic multicast 4/4

- ➤ This works because if any site receives message and remains operational, all sites will receive the message
 - → remember point-to-point links are reliable
- ➤ A site will not receive a message only if *no* site that did receive the message stays operational long enough to forward it
- ➤ Protocol is expensive because of extra messages sent and need to keep some information around to detect duplicate messages
- ➤ Number of messages sent could be reduced if messages are only retransmitted in case of failure of original sender

Ordered multicast

- ➤ Will look at totally and causally ordered multicast
- ➤ Most ordered broadcast protocols make use of *hold-back*
- ➤ A message that is received by some process might be held back (i.e. not delivered) until other messages that should be delivered before it are received and delivered
- ➤ Message on the hold back queue may be *deliverable* or *undeliverable*
- ➤ A deliverable message may be delivered only when it reaches the head of the queue

Distributed Systems

Totally ordered multicast 1/9

- ➤ Want message delivered to every destination in the same order
- ➤ Basic idea is to assign totally ordered timestamps to every message and deliver messages in timestamp order
- Two possible approaches to generating timestamp
 - → use a centralised sequencer or token
 - → distributed agreement

Totally ordered multicast 2/9

Sequencer

- ➤ One process acts as the sequencer
- ➤ All messages are sent to the sequencer
- ➤ Sequencer assigns a timestamp to each message and multicasts message to all other destinations
- ➤ All destinations deliver messages in timestamp order
 - → messages held back until all previous messages delivered
- Lost messages are easily detected and can be obtained from sequencer

Distributed Systems

Totally ordered multicast 3/9

- ➤ Simple, but:
- ➤ Increased message latency
- > Sequencer is single point of failure
- > Sequencer may become a bottleneck

Totally ordered multicast 4/9

ISIS total-ordering protocol

- Assume message sent from single sender to multiple destinations
- Each received message is put in the hold back queue of the receiver
 - → marked as undeliverable
- ➤ The receiver assigns a *proposed* timestamp to the message and returns it in a message to the sender
 - → must be larger than any timestamp proposed or received by that process in the past
 - → made unique by including process identifier as a suffix to the timestamp
- ➤ Sender chooses *largest* proposed timestamp as final timestamp for message and informs destinations

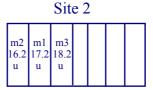
Distributed Systems

Totally ordered multicast 5/9

- Receivers assign final timestamp to message in hold-back queue and mark message as deliverable
- ➤ Hold-back queue is reordered in timestamp order
- ➤ When the message at the head of the hold-back queue is deliverable it is delivered

Totally ordered multicast 6/9





	Site 3						
m1 17.3 u	m3 18.3 u	m2 19.3 u					

Site 3

Step 1:messages delivered to 3 sites in different orders

m3 m2 15.1 17.1 u u	m1 m1 17.3 d		
---------------------------	--------------	--	--

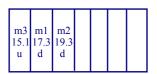


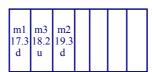
Step 2: m1 assigned final timestamp and delivered at site 3

Distributed Systems

- 1

Totally ordered multicast 7/9







Step 3:m2 assigned final timestamp and m1 delivered at node 2

m1 m 17.3 18 d d

m3 18.3 d	m2 19.3 d					
-----------------	-----------------	--	--	--	--	--

ı					
	m3 18.3	m2 19.3			
ı	d	d			
ı					

Step 4: m3 assigned final timestamp; all messages delivered

Totally ordered multicast 8/9

Why does this work?

- ➤ Guarantees that once a message is delivered no earlier message will arrive
- ➤ Notice that because sender always chooses highest proposed timestamp, reordering the hold-back queue can only ever result in a message moving backwards
 - → can never be assigned a timestamp lower than its proposed timestamp
- ➤ So, when a deliverable message reaches the head of the queue, no message with a lower timestamp can possibly arrive.

Distributed Systems 1

Totally ordered multicast 9/9

- > Again protocol is expensive:
 - → two rounds of communication similar to 2PC protocol
 - → must wait for preceding messages to be delivered
- > Can tolerate failure of any destination
- Like sequencer protocol, have to handle failure of sender during protocol execution
- > Sender not likely to be a bottleneck

Causally ordered multicast 1/3

- ➤ The basic requirement is that before a message is delivered to some process, all the messages that causally precede it must already have been delivered
- ➤ Those messages are the ones that were received by the sender of the message before sending the new message
- ➤ Could have the sender attach all those messages to each new message
 - → be definition, it has them!
- ➤ A better solution is simply to attach a description of what those messages were using a vector timestamp

Distributed Systems 1

Causally ordered multicast 2/3

- Each process p_i maintains a vector clock VT_i
- ➤ VT_i updated as usual as messages are delivered to p_i
 - \rightarrow VT_i encodes the set of messages from all processes that have been delivered to p_i
- > VT_i is used to timestamp multicast messages
 - \rightarrow pi increments $VT_i[i]$ before sending every message and adds it as timestamp TS to the message
- ➤ A multicast message can be delivered immediately to its sender
 - → by definition, this message follows all the messages that were already delivered to its sender
- At any other process p_j ($j \neq i$) incoming messages are placed on the hold back queue

Causally ordered multicast 3/3

➤ A message can be delivered only after all messages from the same sender...

i.e.
$$TS[i] = VT_i[i] + 1$$

- \blacktriangleright ... and all causally preceding messages have been delivered to p_j i.e. $VT_i[K] \ge TS[k]$ for all $k \ne i$
- ➤ Moreover, timestamp allows missing messages (and who originally sent them) to be identified
- ➤ No single point of failure but need some way of determining when a particular message has been delivered everywhere
- ➤ For large groups, size of timestamps is an issue