Multicast Problem

Node with a piece of information to be communicated to everyone

Distributed Group of “Nodes” =
Processes at Internet-based host
Other Communication Forms

- **Multicast** → message sent to a group of processes
- **Broadcast** → message sent to all processes (anywhere)
- **Unicast** → message sent from one sender process to one receiver process
Who Uses Multicast?

- A widely-used abstraction by almost all cloud systems
- Storage systems like Cassandra or a database
  - Replica servers for a key: Writes/reads to the key are multicast within the replica group
  - All servers: membership information (e.g., heartbeats) is multicast across all servers in cluster
- Online scoreboards (ESPN, French Open, FIFA World Cup)
  - Multicast to group of clients interested in the scores
- Stock Exchanges
  - Group is the set of broker computers
  - Groups of computers for High frequency Trading
- Air traffic control system
  - All controllers need to receive the same updates in the same order
Multicast Ordering

- Determines the meaning of “same order” of multicast delivery at different processes in the group
- Three popular flavors implemented by several multicast protocols
  1. FIFO ordering
  2. Causal ordering
  3. Total ordering
1. FIFO ordering

• Multicasts from each sender are received in the order they are sent, at all receivers

• Don’t worry about multicasts from different senders

• More formally
  
  – *If a correct process issues (sends)*
    
    \[ \text{multicast}(g,m) \text{ to group } g \text{ and then } \text{multicast}(g,m') \text{, then every correct process that delivers } m' \text{ would already have delivered } m. \]
M1:1 and M1:2 should be received in that order at each receiver.

Order of delivery of M3:1 and M1:2 could be different at different receivers.
Multicasts whose send events are causally related, must be received in the same causality-obeying order at all receivers.

Formally:
- If \( \text{multicast}(g,m) \Rightarrow \text{multicast}(g,m') \) then any correct process that delivers \( m' \) would already have delivered \( m \).
- (\( \Rightarrow \) is Lamport’s happens-before)
M3:1 → M3:2, and so should be received in that order at each receiver.

M1:1 → M3:1, and so should be received in that order at each receiver.

M3:1 and M2:1 are concurrent and thus ok to be received in different orders at different receivers.
Causal vs. FIFO

- Causal Ordering $\implies$ FIFO Ordering

Why?
- If two multicasts $M$ and $M'$ are sent by the same process $P$, and $M$ was sent before $M'$, then $M \rightarrow M'$
- Then a multicast protocol that implements causal ordering will obey FIFO ordering since $M \rightarrow M'$

- Reverse is not true! FIFO ordering does not imply causal ordering.
**Why Causal at All?**

- Group = set of your friends on a social network
- A friend sees your message m, and she posts a response (comment) m’ to it
  - If friends receive m’ before m, it wouldn’t make sense
  - But if two friends post messages m” and n” concurrently, then they can be seen in any order at receivers
- A variety of systems implement causal ordering: Social networks, bulletin boards, comments on websites, etc.
3. Total Ordering

- Also known as “Atomic Broadcast”
- Unlike FIFO and causal, this does not pay attention to order of multicast sending
- Ensures all receivers receive all multicasts in the same order
- Formally
  - *If a correct process P delivers message m before m’ (independent of the senders), then any other correct process P’ that delivers m’ would already have delivered m.*
The order of receipt of multicasts is the same at all processes. 

M1:1, then M2:1, then M3:1, then M3:2 

May need to delay delivery of some messages
Since FIFO/Causal are orthogonal to Total, can have hybrid ordering protocols too

- FIFO-total hybrid protocol satisfies both FIFO and total orders
- Causal-total hybrid protocol satisfies both Causal and total orders
That was what ordering is
But how do we implement each of these orderings?
Each receiver maintains a per-sender sequence number (integers)

- Processes $P_1$ through $P_N$
- $P_i$ maintains a vector of sequence numbers $P_i[1\ldots N]$ (initially all zeroes)
- $P_i[j]$ is the latest sequence number $P_i$ has received from $P_j$
FIFO Multicast: Updating Rules

- Send multicast at process $P_j$:
  - Set $P_j[j] = P_j[j] + 1$
  - Include new $P_j[j]$ in multicast message as its sequence number
- Receive multicast: If $P_i$ receives a multicast from $P_j$ with sequence number $S$ in message
  - if ($S == P_i[j] + 1$) then
    - deliver message to application
    - Set $P_i[j] = P_i[j] + 1$
  - else buffer this multicast until above condition is true
FIFO Ordering: Example

P1
[0,0,0,0]

P2
[0,0,0,0]

P3
[0,0,0,0]

P4
[0,0,0,0]
FIFO Ordering: Example
**FIFO Ordering: Example**

```
P1
[0,0,0,0]  [1,0,0,0]  [2,0,0,0]
P1, seq: 1  P1, seq: 2

P2
[0,0,0,0]  [1,0,0,0]  [0,0,0,0]
Deliver!

P3
[0,0,0,0]  [0,0,0,0]  [1,0,0,0]
Buffer!

P4
[0,0,0,0]  [1,0,0,0]  [1,0,0,0]
Deliver!
Deliver this!
Deliver buffered <P1, seq:2>
Update [2,0,0,0]
```
**FIFO Ordering: Example**

The diagram illustrates a FIFO (First-In, First-Out) ordering example involving four processes: P1, P2, P3, and P4. The processes interact with messages labeled with sequences and timestamps. The messages include:

- **Deliver!**
- **Buffer!**
- **Deliver this!**
- **Update**

The processes and their sequences are:

- **P1**: Sequence 1
- **P1**: Sequence 2
- **P2**: Sequence 0
- **P3**: Sequence 0
- **P4**: Sequence 0

The messages and their interactions are shown by arrows connecting the processes over time. The timestamps and sequences are indicated at various points in the diagram.
FIFO Ordering: Example
FIFO Ordering: Example
Total Ordering

- Ensures all receivers receive all multicasts in the same order
- Formally
  - *If a correct process* $P$ *delivers message* $m$ *before* $m'$ *(independent of the senders), then any other correct process* $P'$ *that delivers* $m'$ *would already have delivered* $m$. 
**Sequencer-based Approach**

- Special process elected as leader or sequencer
- Send multicast at process $P_i$:
  - Send multicast message $M$ to group and sequencer
- Sequencer:
  - Maintains a global sequence number $S$ (initially 0)
  - When it receives a multicast message $M$, it sets $S = S + 1$, and multicasts $<M, S>$
- Receive multicast at process $P_i$:
  - $P_i$ maintains a local received global sequence number $S_i$ (initially 0)
  - If $P_i$ receives a multicast $M$ from $P_j$, it buffers it until it both
    1. $P_i$ receives $<M, S(M)>$ from sequencer, and
    2. $S_i + 1 = S(M)$
  - Then deliver it message to application and set $S_i = S_i + 1$
Multicasts whose send events are causally related, must be received in the same causality-obeying order at all receivers.

Formally

- If \( \text{multicast}(g,m) \rightarrow \text{multicast}(g,m') \) then any correct process that delivers \( m' \) would already have delivered \( m \).
- \( \rightarrow \) is Lamport’s happens-before
Causal Multicast: Datastructures

- Each receiver maintains a vector of per-sender sequence numbers (integers)
  - Similar to FIFO Multicast, but updating rules are different
  - Processes P\textsubscript{1} through P\textsubscript{N}
  - P\textsubscript{i} maintains a vector P\textsubscript{i}[1…N] (initially all zeroes)
  - P\textsubscript{i}[j] is the latest sequence number P\textsubscript{i} has received from P\textsubscript{j}
Causal Multicast: Updating Rules

• Send multicast at process \( P_j \):
  – Set \( P_j[j] = P_j[j] + 1 \)
  – Include new entire vector \( P_j[1…N] \) in multicast message as its sequence number

• Receive multicast: If \( P_i \) receives a multicast from \( P_j \) with vector \( M[1…N] (= P_j[1…N]) \) in message, buffer it until both:
  1. This message is the next one \( P_i \) is expecting from \( P_j \), i.e.,
     • \( M[j] = P_i[j] + 1 \)
  2. All multicasts, anywhere in the group, which happened-before \( M \) have been received at \( P_i \), i.e.,
     • For all \( k \neq j \): \( M[k] \leq P_i[k] \)
     • i.e., Receiver satisfies causality
  3. When above two conditions satisfied, deliver \( M \) to application and set \( P_i[j] = M[j] \)
Causal Ordering: Example
Causal Ordering: Example
Causal Ordering: Example

P1
[0,0,0,0] -> [1,0,0,0] Deliver!

P2
[0,0,0,0] -> [1,0,0,0] Deliver!

P3
[0,0,0,0] -> [1,1,0,0] Missing 1 from P1 Buffer!

P4
[0,0,0,0] -> [1,0,0,0] Deliver!

Time
Causal Ordering: Example

Time

P1
[0,0,0,0]

P2
[0,0,0,0] [1,0,0,0] Deliver!

P3
[0,0,0,0] [1,0,0,0] Deliver!

P4
[0,0,0,0] [1,0,0,0] [1,0,0,1] Deliver!

Receiver satisfies causality

Deliver!

Missing 1 from P1 Buffer!
Causal Ordering: Example

P1
[0,0,0,0]  [1,0,0,0]
[1,0,0,0] Delivered!

P2
[0,0,0,0] [1,0,0,0] [1,1,0,0] Delivered!
[1,1,0,0] Receiver satisfies causality

P3
[0,0,0,0] [1,0,0,0] [1,0,0,1]
Missing 1 from P1
Buffer!

P4
[0,0,0,0] [1,0,0,0] [1,0,0,1]
Delivered!

Time
Causal Ordering: Example

P1
[0,0,0,0] [1,0,0,0] Deliver!

P2
[0,0,0,0] [1,0,0,0] Deliver!

P3
[0,0,0,0] Missing 1 from P1 Buffer!

P4
[0,0,0,0] [1,0,0,0] Deliver!

[1,1,0,0] Deliver!
Receiver satisfies causality

[1,1,0,0] Deliver!
Receiver satisfies causality

[1,0,0,1] Deliver! Receiver satisfies causality for buffered multicasts

[1,0,0,0] Deliver P1’s multicast

[1,0,0,1] Deliver P2’s buffered multicast

[1,0,0,0] Deliver P4’s buffered multicast

Missing 1 from P1 Buffer!

Time
Causal Ordering: Example

P1
[0,0,0,0] [1,0,0,0] Deliver!

P2
[0,0,0,0] [1,0,0,0] Deliver!

P3
[0,0,0,0] [1,0,0,0] Deliver!

P4
[0,0,0,0] [1,0,0,0] [1,0,0,1] Deliver!

Receiver satisfies causality

Deliver P1’s multicast
Receiver satisfies causality for buffered multicasts
Deliver P2’s buffered multicast
Deliver P4’s buffered multicast

Missing 1 from P1
Buffer!

Missing 1 from P1
Buffer!
Summary: Multicast Ordering

• Ordering of multicasts affects correctness of distributed systems using multicasts
• Three popular ways of implementing ordering
  – FIFO, Causal, Total
• And their implementations
• What about reliability of multicasts?
• What about failures?
Reliable Multicast

- Reliable multicast loosely says that every process in the group receives all multicasts
  - Reliability is orthogonal to ordering
  - Can implement Reliable-FIFO, or Reliable-Causal, or Reliable-Total, or Reliable-Hybrid protocols
- What about process failures?
- Definition becomes vague
Reliable Multicast (Under Failures)

• Need all correct (i.e., non-faulty) processes to receive the same set of multicasts as all other correct processes
  – Faulty processes stop anyway, so we won’t worry about them
Let’s assume we have reliable unicast (e.g., TCP) available to us.

First-cut: Sender process (of each multicast M) sequentially sends a reliable unicast message to all group recipients.

First-cut protocol does not satisfy reliability:
- If sender fails, some correct processes might receive multicast M, while other correct processes might not receive M.
**REALLY Implementing Reliable Multicast**

- Trick: Have receivers help the sender
  1. Sender process (of each multicast M) sequentially sends a reliable unicast message to all group recipients
  2. When a receiver receives multicast M, it also sequentially sends M to all the group’s processes
Analysis

- Not the most efficient multicast protocol, but reliable
- Proof is by contradiction
- Assume two correct processes \( P_i \) and \( P_j \) are so that \( P_i \) received a multicast \( M \) and \( P_j \) did not receive that multicast \( M \)
  - Then \( P_i \) would have sequentially sent the multicast \( M \) to all group members, including \( P_j \), and \( P_j \) would have received \( M \)
  - A contradiction
  - Hence our initial assumption must be false
  - Hence protocol preserves reliability
Virtual Synchrony or View Synchrony

- Attempts to preserve multicast ordering and reliability in spite of failures
- Combines a membership protocol with a multicast protocol
- Systems that implemented it (like Isis Systems) have been used in NYSE, French Air Traffic Control System, Swiss Stock Exchange
Each process maintains a membership list
The membership list is called a View
An update to the membership list is called a View Change
- Process join, leave, or failure
Virtual synchrony guarantees that all view changes are delivered in the same order at all correct processes
- If a correct P1 process receives views, say \{P1\}, \{P1, P2, P3\}, \{P1, P2\}, \{P1, P2, P4\} then
- Any other correct process receives the same sequence of view changes (after it joins the group)
  - P2 receives views \{P1, P2, P3\}, \{P1, P2\}, \{P1, P2, P4\}
Views may be delivered at different physical times at processes, but they are delivered in the same order
A multicast M is said to be “delivered in a view V at process Pi” if
- Pi receives view V, and then sometime before Pi receives the next view it delivers multicast M

Virtual synchrony ensures that
1. The set of multicasts delivered in a given view is the same set at all correct processes that were in that view
   - What happens in a View, stays in that View
2. The sender of the multicast message also belongs to that view
3. If a process Pi does not deliver a multicast M in view V while other processes in the view V delivered M in V, then Pi will be forcibly removed from the next view delivered after V at the other processes
Satisfies virtual synchrony
Does not satisfy virtual synchrony
Satisfies virtual synchrony
Time

View\{P1,P2,P3,P4\}

View\{P1,P2,P3,P4\}

View\{P1,P2,P3,P4\}

View\{P1,P2,P3,P4\}

M1

M2

M3

Crash

Does not satisfy virtual synchrony
Satisfies virtual synchrony

M2 (not delivered at P2)
Does not satisfy virtual synchrony
Does not satisfy virtual synchrony
Satisfies virtual synchrony
What about Multicast Ordering?

• Again, orthogonal to virtual synchrony
• The set of multicasts delivered in a view can be ordered either
  – FIFO
  – Or Causally
  – Or Totally
  – Or using a hybrid scheme
Called “virtual synchrony” since in spite of running on an asynchronous network, it gives the appearance of a synchronous network underneath that obeys the same ordering at all processes

So can this virtually synchronous system be used to implement consensus?

No! VSync groups susceptible to partitioning

– E.g., due to inaccurate failure detections
Partitioning in View synchronous systems
SUMMARY

• Multicast an important building block for cloud computing systems
• Depending on application need, can implement
  – Ordering
  – Reliability
  – Virtual synchrony
ANNOUNCEMENTS

• Midterm next Tuesday
• Locations:
  – DCL 1320: if your last name starts with A-Q
  – 1 Noyes 217 (Map): if your last name starts with R-Z
• Material through lecture 12 (Time and Ordering)