Coordination 2

To do ...

- Group communication
- Basic, reliable and ordered multicast

Coordination and agreement in US Congress 1798-2015
Modes of communication

- **Unicast**: $1 \leftrightarrow 1$
  - Point to point, your everyday

- **Anycast**: $1 \rightarrow \text{<nearest of a set>}$
  - Used with BGP, CDN, DNS, ...

- **Broadcast**: $1 \rightarrow \text{all processes in a system}$
  - ARP

- **Multicast**: $1 \rightarrow \text{many}$
  - Group communication
Group communication

- An abstraction over network or overlay network
- Central concept: group (process may join/leave)
- A msg to the group is sent to all (multicast) with certain guarantees

An important building block for

- Reliable information dissemination
- Collaborative applications (e.g., games)
- A range of fault-tolerance strategies, including consistent update of replicated data
- System monitoring and management
Group communication – Design issues

- Closed vs open
  - Closed – only members can multicast to it

- Peer vs hierarchical
  - P2P – each member communicate with the group
  - Hierarchical – communicate through a coordinator

- Group creation/destruction, membership mgmnt
  - Support group membership changes
  - Detecting participant failures, notifying all of changes
  - Performing group address translation if hierarchical
  - Challenges limit scalability of group communication
Processes may be in multiple groups
The operation \textit{multicast}(m,g) sends \( m \) to all processes in a group \( g \)
Every msg carries the unique id of the sender and the unique destination group identifier
\textit{Deliver}(m) delivers a msg sent to the calling process
\quad – \textit{Deliver} \neq \textit{Receive} – The msg is not always handed to the application layer of the process receiving it
Multicast receiver algorithms decide when to deliver a msg to the process/application.

A received msg may be:
- Delivered immediately – put on a delivery queue that the process reads
- Placed on a hold-back queue – maybe to wait for an earlier msg
- Rejected/discarded – maybe duplicate or an earlier msg we don’t need anymore
A useful primitive, basic multicast

- A correct process will eventually deliver the message, as long as the multicaster does not crash
  - Beyond IP multicast, defined by B-multicast and B-deliver

- A straightforward way to implement it using a **reliable** one-to-one send operation

  
  \[
  \text{To B-multicast}(g,m): \text{for each process } p \text{ in } g, \text{send}(p,m) \\
  \text{On receive}(m) \text{ at } p: \text{B-deliver}(m) \text{ at } p
  \]

- Potential ack-implosion
  - Acks from reliable send may arrive close to each other, …
  - overloaded multicast process will start to drops them, …
  - leading to more msgs and more acks …

- A more scalable implementation using IP-multicast
Reliable multicast

- Define reliable multicast with \textit{R-multicast} and \textit{R-deliver}

- Must satisfy
  - Integrity – A correct $p$ in group $g$ delivers $m$ at most once
  - Validity – If a correct process multicast $m$, it eventually delivers $m$
  - Agreement – If a correct process delivers $m$, all other correct processes in the group eventually deliver $m$
    - Related to atomicity – All or nothing at all
    - Not true with \textit{B-multicast} which is based on reliable one-to-one sends, so some process may deliver while others do not

- Validity guarantees liveness for (only) the sender?!
  - Yes, but validity + agreement = overall liveness
  - If one process eventually delivers a message $m$, …
Reliable multicast

- **Building on B-multicast**

  On initialization
  \[\text{Received} := \{\}\]

  For process \(p\) to R-multicast message \(m\) to group \(g\),
  \[\text{B-multicast}(g,m) \quad \text{// } p \text{ is included as destination}\]

  On B-deliver\((m)\) at process \(q\)
  \[\text{If } m \text{ is not Received}
  \quad \text{Received} = \text{Received} \cup \{m\}
  \quad \text{If } (q \neq p) \text{ then B-multicast}(g,m) \quad \text{// if not the sender}
  \quad \text{R-deliver } m\]

- **Some observations**
  - Since messages may arrive more than once, detect and discard duplicates
  - Correct but inefficient, sent \(|g|\) times to each
Uniform properties and agreement

- Agreement so far refers to correct/never fail processes
  - Uniform property – holds whether or not processes are correct
- **Uniform** agreement
  - If a process, correct or failed, delivers \( m \), then all correct processes in \( g \) eventually deliver \( m \)

```
On B-deliver at process q
If m is not received
   Received = Received U {m}
If (q != p) then B-multicast(g,m)
R-deliver m
End
```

- If it crashes after R-deliver, since it first B-multicast it follows that all correct processes will eventually deliver it
Uniform properties and agreement

- Consider a “minor” change in the previous code …

```
On B-deliver at process q
    If m is not received
        Received = Received U {m}
        R-deliver m
        If (q != p) then B-multicast(g,m)
    End
Crash!
```

- Matters if a process can take an action that produces an observable inconsistency before it crashes
  - Updates to the state of replicas of a bank account

- As there is a uniform version of agreement, there are uniform versions of validity, integrity and ordering properties
Ordering multicast

- Basic algorithm delivers messages in arbitrary order
  - Not enough for many applications

- Common ordering requirements
  - FIFO – Messages from same sources delivered in FIFO order
    - Partial, not total, ordering
  - Causal – Causally related messages arrive in order
    - Causal implies FIFO, partial ordering as well
  - Total – Consistent ordering everywhere
    - Not particular order, but the same everywhere
    - FIFO-total and causal-total orderings

- Other hybrids
  - Reliable + ordering: reliable totally ordered – atomic multicast
FIFO ordering multicast

- If a correct process \( multicast(g,m) \) and then \( multicast(g,m') \), every correct process that delivers \( m' \), delivers \( m \) before \( m' \).

\[
\begin{align*}
p_1 & \quad m_1^1 & \quad m_1^2 & \quad m_1^3 \\
p_2 & \quad m_3^1 & \quad & \quad \\
p_3 & \quad & \quad & \\
\end{align*}
\]

- Idea of the algorithm:
  - Keep track of the last message received from everyone;
  - When a message arrives, holds it back unless it is the next message you are supposed to get.
FIFO ordering multicast

- To implement it (FO-multicast, and FO-deliver)

\[ S_g^p \] – count of how many msgs \( p \) has sent to \( g \)
\[ R_g^q \] – seq # of latest msg \( p \) delivered from \( q \) that was sent to group \( g \)

**For \( p \) to FO-multicast a msg to group \( g \),**
- Piggybacks the value \( S_g^p \) onto the msg
- B-multicast the msg to \( g \)
- Increment \( S_g^p \) by 1

**Upon \( p \) receipt of a msg from \( q \) with seq # \( S \)**
- If \( p \) checks whether \( S = R_g^q + 1 \) then
  - this is the next expected msg, FO-deliver it
  - set \( R_g^q = S \)
- Else
  - If \( S > R_g^q + 1 \), place it in the hold-back queue
Total ordering multicast

- If a correct process delivers $m$ before $m'$, every correct process that delivers $m'$, delivers $m$ before $m'$

- Basic approach to implement it – assign totally ordered identifiers to multicast messages

- Delivery algorithm is similar to FIFO but using group-specific sequence #s instead of process-specific ones
  - To assign a number – via a sequencer or all processes agree on a sequence number
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

In absence of proper synchronization: replica #1 ← $1111, while replica #2 ← $1110
Implementing total ordering – Sequencer

- To multicast \( m \) to group \( g \), a process
  - Attaches a unique id \( \text{id}(m) \) to it
  - Sends to other members of \( g \) and the sequencer(\( g \))

- **Sequencer(\( g \))**
  - Maintains a group-specific seq \# \( S_g \)
  - Use it to assign increasing seq \# to the msgs that it \textit{B-delivers}

- A message remains in the hold-back queue until it can be \textit{TO-delivered} according to the corresponding seq \#
Implementing total ordering - sequencer

Algorithm for group member p
On initialization $r_g := 0$
To TO-multicast $m$ to $g$

\[ B\text{-multicast}(g \cup \{\text{sequencer}(g)\}, <m, i>) \]

On B-deliver($<m, i>$) with $g = \text{group}(m)$

Place $<m,i>$ in hold-back queue

On B-deliver($m_{\text{order}} = <\text{"order"}, i, S>)$ with $g = \text{group}(m_{\text{order}})$

wait until $<m, i>$ in hold-back queue and $S = r_g$

TO-deliver $m$

$r_g := S + 1$

Algorithm for sequencer of $g$
On initialization $s_g := 0$

On B-deliver($<m, i>$) with $g = \text{group}(m)$

\[ B\text{-multicast}(g, <\text{"order"}, i, s_g>) \]

$s_g := s_g + 1$
Implementing total ordering – Sequencer

- But, sequencer can become a bottleneck and it is a critical point of failure
- Some ideas
  - Multiple sequencers to deal with failures
  - Token-based sequencer (if there’s only one process sending totally ordered multicast, give it the token!)
  - Fully distributed …
Total ordering – Distributed

• Basic idea – every process wanting to multicast a message acts as a sequencer*
  – To assign a sequence number, ask everyone for a proposed sequence number (*Proposed seq*)
    • B-multicast of the message is the poll
  – Pick the largest proposed number
  – Let everyone know what that is (*Agreed seq*)

• **Remember multiple processes may be trying to multicast a message at once**

*Based on Birman and Josephs 1987, ISIS
For process p to multicast a msg m to g
(1) B-multicast <m, i> to g (i is a uid for m)
(3) collects all proposed seq #,
   Selects largest, a, as next agreed seq #
   B-multicast<i,a> to g

Each process q in g keeps
• \( A_g^q \), the largest seq #s it has observed so far for g
• \( P_g^q \), its own largest proposed seq #s

Each process q
(2) When getting <m, i>
   Replies to p with proposed seq # of \( P_g^q = \text{Max}(A_g^q, P_g^q) + 1 \)
   Assigns proposed seq # to msg for now, place it in hold-back queue
(3) When getting <i,a>
   sets \( A_g^q = \text{Max}(A_g^q, a) \)
   Attaches a to the msg, reorder msgs in the hold-back queue

When msg in the front of the queue has been assigned
its agreed seq #, move it to the delivery queue

* Based on Birman, Joseph, Reliable communication in the presence of failures, ACM TOCS 5(1), 1987
Causally ordered multicast

- If $\text{multicast}(g,m) \rightarrow \text{multicast}(g,m')$ based only on messages exchanged by processes in $g$, every correct process that delivers $m'$, delivers $m$ before $m'$

- i.e., a message $m$ is delivered only if all causally preceding messages have already been delivered
  - Keep it in a hold-back queue until then
Causally ordered multicast

- Clock adjustment only when sending/receiving
  - Consider two processes; $p_i$ sending a message to $p_j$
  - $p_i$ increments $V_i[i]$ only when sending a message
  - $p_j$ “adjusts” $V_j$ when receiving a message

- $p_j$ postpones delivery of $m$ until:
  - $ts(m)[i] = V_j[i] + 1$
    - $m$ 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 multicast

Algorithm for group member $p_i (i = 1, 2, ..., N)$

On initialization
$$V_i^g[j] := 0 \ (j = 1, 2, ..., N)$$

To CO-multicast message $m$ to group $g$
$$V_i^g[i] := V_i^g[i] + 1;$$
$$B\text{-multicast}(g, <V_i^g, m>)$$

On B-deliver $<V_j^g, m>$ from $p_j \ (j \neq i)$, with $g = \text{group}(m)$
place $<V_i^g, m>$ in hold-back queue
wait until $V_j^g[j] = V_j^g[j] + 1$ and $V_j^g[k] \leq V_j^g[k] \ (k \neq j)$
CO-deliver $m; \ // \ after \ removing \ it \ from \ the \ hold-back \ queue$
$$V_j^g[j] = V_j^g[j] + 1$$
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 $k \neq j$
Closing ideas – Reliable and in order

- Note that
  - FIFO and causal are partial orderings
  - Causal => FIFO
  - Total ordering allows message delivery to be ordered arbitrarily, as long as it is the same for all
  - No mention of reliability
    - If you don’t deliver \( m' \), you are OK

- We can define some hybrids
  - FIFO-total, causal-total, reliable FIFO, …