Today
- Group communication
- Basic, reliable and
- ordered multicast

*How can processes agree on an action or a value?*
Modes of communication

- **Unicast**
  - \(1 \leftrightarrow 1\)
  - Point to point

- **Anycast**
  - \(1 \rightarrow \text{<nearest of a set>}\)
  - Introduced with IPv6; used with BGP

- **Broadcast**
  - \(1 \rightarrow \text{all processes in a system}\)

- **Multicast**
  - \(1 \rightarrow \text{many}\)
  - Group communication
Group communication

- An abstraction over network or an equivalent overlay network
- Central concept: group
  - Process may join/leave
  - A message to the group is sent to all members (multicast) with certain guarantees
- An important building block for
  - Reliable information dissemination
  - Collaborative applications (e.g., games included)
  - 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 – to communicate go through a coordinator

- Group creation/destruction, membership management
  - Providing an interface for group membership changes
  - Detecting participant failures, notifying all of changes
  - Performing group address translation if hierarchical
  - Challenges limit scalability of group communication
Group communication – System model

- Processes may be in multiple groups
- The operation $\text{multicast}(m,g)$ sends msg $m$ to all processes in a group $g$
- Every msg carries the unique identifier of the sender and the unique destination group identifier
- $\text{Deliver}(m)$ delivers a msg sent to the calling process
  - $\text{Deliver} \neq \text{Receive}$ – The msg is not always handed to the application layer of the process receiving it
Multicast receiver algorithms decide when to deliver a message to the process/application.

A received message 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 message
- Rejected/discarded – maybe duplicate or an earlier message we don’t need anymore
A useful, basic multicast

- A correct process will eventually deliver the message, as long as the multicaster does not crash
  - This is beyond IP multicast
  - We call the primitives \textit{B-multicast} and \textit{B-deliver}
- A straightforward way to implement it using a reliable one-to-one \textit{send} operation
  - To \textit{B-multicast}(g,m): for each process \( p \) in \( g \), \textit{send}(\( p \),m)
  - On \textit{receive}(m) at \( p \): \textit{B-deliver}(m) 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 …
Reliable multicast

- Define reliable multicast with $R$-multicast and $R$-deliver
- Must satisfy
  - Integrity – A correct $p$ in group $g$ delivers $m$ at most once
  - Validity – If a correct process multicast $m$, then it will eventually deliver $m$
  - Agreement – If a correct process delivers $m$, then all other correct processes in group will eventually deliver $m$
    - Related to atomicity – all or nothing
    - This does not hold for $B$-multicast which is based on reliable one-to-one sends, so some process may deliver while others do not

- Validity guarantees liveness for 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 \( g \),
  
  \[ \text{B-multicast}(g,m) \quad \text{// p is included as destination} \]

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

- Some observations
  
  - Since msgs may arrive more than once, detect and discard duplicates
  
  - Correct but clearly inefficient (use IP multicast to help)
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 ($q \neq p$) then B-multicast(g,m)
  
  R-deliver $m$

  End

  Crash!

- 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 ...

  \[
  \text{On B-deliver at process } q
  \]
  \[
  \text{If } m \text{ is not received}
  \]
  \[
  \text{Received } = \text{Received } U \{m\}
  \]
  \[
  \text{R-deliver } m
  \]
  \[
  \text{If } (q \neq p) \text{ then B-multicast(g,m)}
  \]
  \[
  \text{End}
  \]

- Matters if a process can take an action that produces an observable inconsistency before it crashes
- As there is a uniform version of agreement, there are uniform versions of validity, integrity and ordering properties

\[\text{Crash!} \quad \text{Not the same!}\]
Ordering multicast

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

- Common ordering requirements
  - FIFO – Msgs from same sources delivered in the order sent
    - Partial 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 $\text{multicast}(g, m)$ and $\text{multicast}(g, m')$, every correct process that delivers $m'$, delivers $m$ before $m'$

- 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)

  - For \( p \) to FO-multicast a msg to group \( g \),
    - Piggybacks the value \( S_g^p \) on to the msg
    - B-multicast the msg to \( g \) and increment \( S_g^p \) by 1

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

\( S_g^p \) – 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 \( g \)
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

- Two main methods to assign identifiers
  - A sequencer process assigns them
  - All processes collectively agree on the assignment of sequence #s
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 $id(m)$ to it
  - Sends to $\text{sequencer}(g)$ and other members of $g$

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

- A msg remains in the hold-back queue until it can be $TO$-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 \( g \)

- \( B\)-multicast(\( g \cup \{\text{sequencer}(g)\}, \langle m, i \rangle \))

**On B-deliver(\( \langle m, i \rangle \)) with** \( g = \text{group}(m) \)

- Place \( \langle m, i \rangle \) in hold-back queue

**On B-deliver(\( m_{\text{order}} = \langle \text{“order”}, i, S \rangle \)) with** \( g = \text{group}(m_{\text{order}}) \)

- wait until \( \langle m, i \rangle \) 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(\( \langle m, i \rangle \)) with** \( g = \text{group}(m) \)

- \( B\)-multicast(\( g \), \( \langle \text{“order”}, i, s_g \rangle \))

- \( s_g := s_g + 1 \)
Implementing total ordering – Sequencer

- Obviously, the sequencer can become a bottleneck and it is a critical point of failure
  - Multiple sequencers to deal with failures
    - e.g., Kaashoek et al. 1989* – put a message in the hold-back queue of \( f + 1 \) nodes before it is delivered to ensure resilience to \( f \) 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 msg acts as a *sequencer*
- To assign a sequence number it polls everyone by a proposed sequence number
  - B-multicast of the message is the poll
- It picks the largest proposed number
- It lets everyone know what that is

*Remember multiple processes may be trying to multicast a message at once*
Total ordering – Distributed

• 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
  – (2) Replies to p with a proposal for msg’ agreed seq # of
    \( P_g^q = \text{Max}(A_g^q, P_g^q) + 1 \)
  – (2) Provisionally assigns proposed seq # to msg,
    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
  – (3) 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 multicasting

- 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 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 $k \neq j$
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, reliable causal, a reliable totally ordered multicast (sometimes called ‘atomic multicast’)