Distributed Systems

Multicast and Agreement

Rik Sarkar
James Cheney

University of Edinburgh
Spring 2014
Recap: Leader election

• Strategy 1: use aggregation trees
• Strategy 2: Use a ring
  – Send messages in only one direction
  – Ids propagate in only one direction
  – Larger ids suppress smaller ids
• Strategy 3: Use a ring
  – Send messages in both direction
  – Exponentially growing neighborhoods
  – Larger ids suppress smaller ids
Strategy 4: Bully Algorithm

• Assume:
  – Each node knows the id of all nodes in the system (some may have failed)
  – Synchronous operation

• Node p decides to initiate election
• p sends election message to all nodes with id > p.id
• If p does not hear “I am alive message” from any node, p broadcasts a message declaring itself as leader
• Any working node q that receives election message from p, replies with own id and “I am alive” message
  – And starts an election (unless it is already in the process of an election)
• Any node q that hears a lower id node being declared leader, starts a new election
Strategy 4: Bully Algorithm

• Assume:
  – Each node knows the id of all nodes in the system (some may have failed)
  – Synchronous operation

• Works even when processes fail
• Works when (some) message deliveries fail.

• What are the storage and message complexities?
Multicast

• Send message to multiple nodes
• A node can join a multicast *group*, and receives all messages sent to that group
• The sender sends only once: to the group address
• The network takes care of delivering to all nodes in the group
• Note: groups are restricted to specific networks such as LANs & WANs
  – Multicast in the university network will not reach nodes outside the network
Multicast

• A special version of broadcast (restricted to a subset of nodes)
• In a LAN
  – Sender sends a broadcast
  – Interested nodes accept the message others reject
• In larger networks we can use a tree
  – Remember trees can be used for broadcast
  – Interested nodes join the tree, and thus get messages
  – All nodes can use the same tree to multicast to the same group
IP Multicast

- IP has a specific multicast protocol
- Addresses from 224.0.0.0 to 239.255.255.255 are reserved for multicast
  - They act as groups
  - Some of these are reserved for specific multicast based protocols
- Any message sent to one of the addresses goes to all processes subscribed to the group
  - Must be in the same “network”
  - Basically depends on how routers are configured
- In a LAN, communication is broadcast
- In more complex networks, tree-based protocols can be used
IP Multicast

• Any process interested in joining a group informs its OS
• The OS informs the “network”
  – The network interface (LAN card) receives and delivers group messages to the OS & process
  – The router may need to be informed
  – IGMP – Internet group management protocol
IP Multicast

• Sender sends only once
• Any router also forwards only once
• No acknowledgement mechanism
  – Uses UDP
• No guarantee that intended recipient gets the message
• Often used for streaming media type content
• Not good for critical information
Multicast

• Can we design a reliable protocol?

• If there are multiple messages, can we ensure they are delivered in correct order?
Reliable Multicast

• The sending process is in the multicast group
• Nodes may fail (by crashing)
• We will use one to one communication between processes
  – The communication is reliable (may be using suitable ack-based protocol)
  – If both processes are alive, the message gets delivered. i.e. the network does not fail
• Note that these assumptions are necessary.
  – If network and message delivery can fail, then there may be 2 sets of processes who never communicate with each other
  – Thus message from one set will never reach the other
Reliable Multicast

- \textit{multicast}(g,m) : multicast message m to group g
- \textit{receive}(m): The OS or network card receives the message and gives to the multicasting process
- \textit{deliver}(m): The multicast process delivers m to the application
Reliable Multicast - definition

• Must have the following properties:
  – Integrity: A working process p in group g delivers m at most once, and m was multicast by some working process
  – Agreement: If a working process delivers m then all other working processes in group g will deliver m
Basic Multicast

- Suppose send(p,m) is reliable
- Define Basic multicast p.Bmulticast(g,m):
  - For each q in g:
    - P.send(q,m)
  - On p.receive(m): # by multicasting algorithm
    - P.Bdeliver(m) # to the application
- Assumes the sender does not crash in operation
- Therefore, does not implement Agreement in presence of crashes
Reliable Multicast

• Use Bmulticast as function/procedure

• Implement $Rmulticast(g,m)$ and $Rdeliver(m)$
Reliable Multicast

- Initialization: $\text{Received} = \{\}$
- $p.R_{multicast}(g,m)$:
  - $p.B_{multicast}(g, m)$
- $Q.B_{deliver}(m)$:
  - If $m$ is not in $\text{Received}$:
    - $\text{Received} = \text{Received} \cup \{m\}$
    - If $p \neq q$ : $q.B_{multicast}(g,m)$
    - $q.R_{deliver}(m)$

- The key point is that $q$ sends the message to other working nodes before it accepts the message and delivers to the interested application
Reliable Multicast

• Integrity: A message is delivered at most once and was multicast by some correct process
  – Obvious, since send(p,m) is reliable

• Agreement: Since a process forwards the message to others before it delivers to the local application
  – If it was in the reverse order, then the following could have occurred:
    • Application gets the message and takes action according to it (such as send a message to update a database)
    • The machine fails, so that no other working processes receive the multicast
    • Result: inconsistent state
  – In the present case, a process failing in between the 2 actions is like it having failed before the multicast starts.
Multicast ordering

• We want messages delivered in “correct” (intended, consistent etc) order

• FIFO: If a process p performs 2 multicasts, then every working process that delivers these 2 messages deliver in the correct order

• Causal: if \( p.m\text{ulticast}(g,m) \rightarrow q.m\text{ulticast}(g,m') \) then every process which delivers both, deliver m before m’

• Total: All working processes deliver messages in the same order
Multicast ordering

• Causal implies FIFO

• Total ordering
  – Requires messages are delivered *same* order by each process
  – But this order may have no relation to causality or message sending order
  – Can be modified to be FIFO-total or Causal-total orders
FIFO ordered multicast

• Our reliable multicast implements FIFO
  – Assuming the Bmulticast sends to group members in same order
  – Sequence numbers can be used to implement FIFO otherwise
FIFO ordered multicast

• Our reliable multicast implements FIFO
  – Assuming the Bmulticast sends to group members in same order
  – Sequence numbers can be used to implement FIFO otherwise
Causally ordered Multicast

• Each process has a Vector clock
• Suppose p sends a multicast m
• q receives m and holds it until:
  – It has delivered any earlier message by p
  – delivered any multicast message that has been delivered by p (to its application) before p multicast m
• These are easy to check using vector timestamps
Total ordered multicast

• Using sequencer process
  – p wants to multicast
  – It asks sequencer process for a sequence number
  – Sends multicast tagged with the sequence number
  – All processes deliver messages by sequence number

• Simple

• Single point of failure and bottleneck
Total ordered multicast

- Using collective agreement
- \( p \) first sends \( B \) multicast to the group
- Each process in group picks a sequence number
- Processes run a distributed protocol to agree on a sequence number for the message
- Messages delivered according to sequence number
Consensus

• Agreeing on things (leader, sequence numbers, time for action, action to be taken etc)
Basic Consensus

• Set of processes
• Each starts with \textit{state} = \textit{undecided}
• Each has a single value
• Have to set their decision variable to the same value and enter decided state
Basic Consensus

• Termination: each process sets its decision variable and enters decided state

• Agreement: If 2 processes have entered decided state, then their decision variables are equal

• Integrity: If all working processes proposed the same value $v$, then all of them in decided state has decision=$v$
Basic Consensus

• A simple solution:
  – Use reliable multicast to communicate all values
  – Use a simple rule (min, max etc) to decide

• Inefficient, but works!
Byzantine generals consensus

• 3 or more generals deciding whether to attack or not
• A commander issues the attack
• One or more processes may be faulty (controlled by the enemy)
• Properties:
  – Termination : everyone decides
  – Agreement : non-faulty processes agree
  – Integrity : If the commander is non-faulty, then all non-faulty processes agree with commander
Byzantine generals consensus

• Suppose 3 processes: A, B, C.
  – C is commander
  – B is faulty
• C says attack to both

• A tells B: “C told me: attack”
• B tells A: “C told me: do-not-attack”

• A knows someone is lying. But does not know who

• No solution with 3 processes

• In general, no solution with $n \leq 3f$ processes, where $f$ is number of faulty processes
Interactive consensus

- Processes have to agree on a vector of values
- Each process contributed only to part of the vector (but all processes must have same vector in the end)

- Termination : everyone decides
- Agreement: they decide the same vector $V$
- If $p_i$ proposes $x$, then in $V_i=x$ for all processes
Consensus in Asynchronous systems

• Cannot be guaranteed

• Process A is not responding:
  – Is it failed or just slow?
  – It might just send a message at the wrong time
Termination detection

• How do we know when a distributed computation has ended?
Termination detection

• We suppose that the computation is started by a process s.
  – This means, other processes start working after receiving message from s or some other process
  – They have no other way to know that a computation is in progress
• s wants to know when all other processes have concluded working
• S starts with weight = 1.0
• Other processes start with weight = 0
• When a process sends a message, it puts part (say, half) of its weight in the message.
• When a process receives a message, it adds the message weight to its own weight.

• When a process has finished computing, it sends its current weight to s

• When s has weight=1, it knows no other process is active