Cloud computing and distributed systems

Coordination and agreement

Zeynep Yücel

Ca' Foscari University of Venice
zeynep.yucel@unive.it
yucelzeynep.github.io

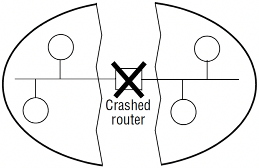
Introduction

Failure assumptions and failure detectors

- For simplicity, assume:
  - Reliable communication channels.
  - No process failure.
- Reliable protocols:
  - Messages eventually delivered.
  - Synchronous systems also have time bounds for delivery.

Introduction

Failure assumptions and failure detectors



- Communication may succeed or delay.
- Router failure.
  - Causes network partition.
- In complex networks, it is oissible to have
  - Asymmetric communication.
  - Intransitive communication.
- Reliability assumption implies repairs of failed links, but processes possibly not communicate at the same time.

Introduction

Failure assumptions and failure detectors

- Correct process shows no failures throughput its execution.
  - Before failure, it is non-failed, but not correct.
- Failure detector checks process status.
  - Local detectors.
- Detectors may be unreliable:
  - Unsuspected: seemed fine based on recent evidence, but may have failed since then.
  - Failed: crash indication exists (e.g. too long silence).

Introduction

Failure assumptions and failure detectors

- Unreliable failure detector:
  - "p is here" messages at every Ts.
  - Maximum message transmission time estimate Ds.
  - If no message received in $T + D$s, local detector reports p as suspected.
  - If message received later, it reports p as OK.
- Small timeouts cause false alarms.
- Large timeouts miss failures.
- Timeouts may be adjusted:
  - Reset timeout based on conditions.
- Synchronous systems can ensure reliability.

Distributed mutual exclusion

- Safe resource sharing requires mutual exclusion.
- Distributed mutual exclusion based on message passing.
  - Communication
  - Coordination
  - Access control
  - Release

Distributed mutual exclusion

- Users updating a text file
  - One user access at a time.
- Car park vacancy tracking.
  - Processes at each entry/exit track vehicle numbers.
  - Processes maintain total count.
  - Mutual exclusion for consistent updates.
  - Preferably without separate server.

Distributed mutual exclusion

Algorithms for mutual exclusion

- N processes $p_1, p_2, \ldots, p_N$.
- Common resource within a single critical section.
- Assume a reliable asynchronous system with no failures.
- Protocol includes entering, accessing, exiting.
- Requirements include safety, liveness, ordering.
  - ME1 (safety): At most one process may execute in CS at a time.
  - ME2 (liveness): Requests to enter and exit the CS eventually succeed.
  - ME3 (ordering): If one request to enter the CS happened-before another, then entry is granted in that order.

Distributed mutual exclusion

Algorithms for mutual exclusion

- ME2 prevents deadlock and starvation.
  - Deadlock: processes stuck due to mutual dependence.
  - Starvation: indefinite postponement while trying to enter.
- Fairness prevents starvation.
  - Entry ordering independent of request times.
    - Happened-before ordering

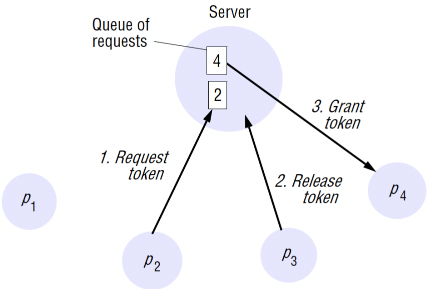
Distributed mutual exclusion

Algorithms for mutual exclusion

- Performance evaluated by:
  - Bandwidth of messages.
  - Client delay time.
  - Throughput of access rates.

Distributed mutual exclusion

The central server algorithm



- Central server algorithm.
  - Process requests access to CS.
  - Server grants a token or queues requests.
  - Process returns token after exit.
  - Entering requires two messages.
  - Exiting requires one message.
- Server may become bottleneck.

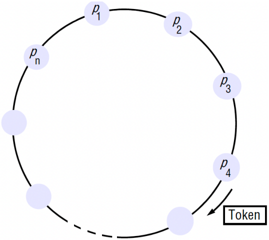
Distributed mutual exclusion

The central server algorithm

- Assumes no failures: ME! and ME2 met.
- Performance discussion:
  - Entering: two messages.
  - Delay due to round-trip time.
  - Exiting: one message.
  - Server is a potential bottleneck.

Distributed mutual exclusion

A ring-based algorithm

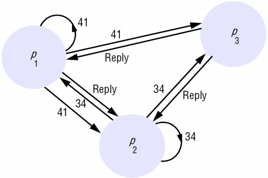


- Processes arranged in a ring.
- Token passed around in one direction.
- If a process does not need the token, it forwards it.

- If a process needs the token, it waits for it and keeps it until exit.
- Token sent to neighbor after exit.
- Meets ME1, ME2.
- Consumes bandwidth.
- Waiting time varies.
- Exiting requires one message.

Distributed mutual exclusion

An algorithm using multicast and logical clocks



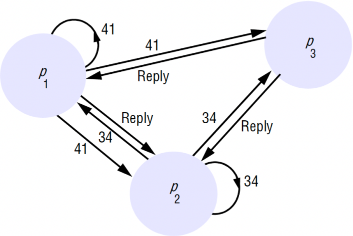
- Multicast request to enter.
- Distinct process IDs, Lamport clocks communication channels.

- Request message format $\langle T, pi \rangle$ where T is the timestamp and p_i is the process identifier.
- Each process tracks its state: RELEASED (outside), WANTED (wanting entry), or HELD (inside CS).
  - If all processes are RELEASED, they reply immediately, allowing entry.
  - If any process is in HELD state, it does not reply until it exits the critical section.
- Requests prioritized by timestamp.
- Processes defer handling requests, until their own is sent.

Distributed mutual exclusion

An algorithm using multicast and logical clocks

- ME1 met: mutual exclusion guaranteed.
  - Total timestamp ordering.



- Concurrent requests, different timestamps.

- Message complexity: 2(N - 1).

Distributed mutual exclusion

Maekawa's voting algorithm

- Voting from peers.
- Consent from a subset of peers is enough for access, is any two subsets overlap.
- Intersection ensures ME1.
  - Voting set overlap.
  - Fixed voting set size K.
  - Process in multiple sets.
- Optimal configuration: $K \sim \sqrt{N}$.

Distributed mutual exclusion

Fault Tolerance Considerations

- Message loss effect?
  - No tolerance for loss.
- Process crash effects?
  - Central server tolerates client crashes.
  - Ring-based fails on any process crash.
  - Multicast adapts for certain crashes.
  - Maekawa tolerates some crashes.

Elections

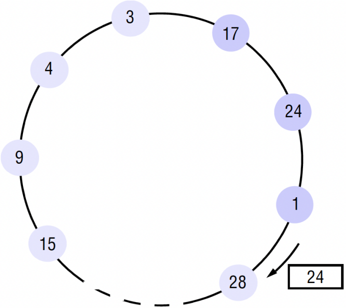
- Election: selecting a unique process for a role.
- Processes must agree on the chosen one.
- Calling for election: initiating election process.
  - Individual process makes a single call.
  - Multiple concurrent election calls possible.
- Participant or non-participant.
- Elected process is unique (even if calls can be multiple).
  - Let largest "identifier" win.
  - Example identifier: $\langle 1/load, i \rangle$ where i is process index.
- Elected process variable at process i: $elected_i$ (initialized to $\perp$).

Elections

- Requirements:
  - Safety (E1): p_i has $elected_i = \perp$ or $elected_i = P$.
  - Liveness (E2): All processes must participate and eventually will have $elected_i \neq \perp$ or crash.
- Performance is measured by network bandwidth (total messages sent) and turnaround time (duration from start to finish of an election).

Elections

A ring-based election algorithm



- Clockwise communication.
- Elect a process highest identifier.
- Start as non-participants.
- Any process can initiate.

- Compare identifiers received.
  - Forwards, if received identifier larger than its own.
  - Puts own identifier into message and sends, if received identifier smaller and process is non-participant.
  - Marks as participant when forwarding.
  - Do not forward, if already participant.

Elections

A ring-based election algorithm

- Own identifier received: becomes coordinator.
- Coordinator sends "elected" message and announces its election.
- Upon receiving, mark itself as non-participant
- Safety condition satisfied.
- Liveness ensured through traversal of the ring.
- Worst case message count.
  - Requires 3N - 1 messages.
    - N-1 to highest identifier.
    - N for loop confirmation.
    - N for coordinator announcement.

Elections

The bully algorithm

- Assumptions:
  - Reliable message delivery.
  - Synchronous system.
  - Processes know which other processes have higher identifiers and can directly communicate with them.
- Message Types:
  - Election Message.
  - Answer Message.
  - Coordinator Message.
- Timeout (T) calculation.
  - Transmission delay T_{trans}; maximum delay for message processing $T_{process}$.
  - Calculate T as $2T_{trans} + T_{process}$.

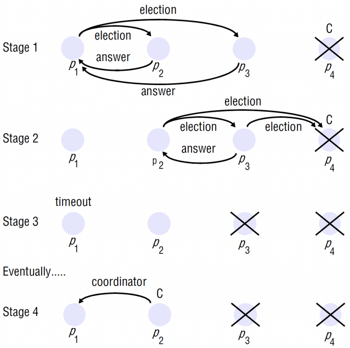
Elections

The bully algorithm

- Algorithm:
  - The one with highest identifier simply declares itself as coordinator.
  - The one with lower identifier can start an election and wait.
  - If no response received within T, self-declare.
  - If there is some response, wait for message for additional T'.
  - If no coordinator message received within T', restart election.
- Update elected coordinator.
- New coordinator can be elected despite an existing functioning one.

Elections

The bully algorithm



- Processes p_1, p_2, p_3, p_4.
- p_1 starts election.
- p_2 and p_3 respond.
- p_3 decides it's coordinator.

- p_3 fails before sending coordinator message.
- p_1's timeout expires, so it starts another election.
- p_2 elected coordinator.

Elections

The bully algorithm

- Liveness condition met.
- Safety condition violation possible.
- Performance varies:
  - Best-case: $O(N)$ messages ($N - 2$).
  - Worst-case: $O(N^2)$ messages.

Coordination and agreement in group communication

System model

- Focus on coordination issues.
- Ensure reliability and ordering.
- Consider a system:
  - Reliable one-to-one channels.
  - Possible crashes (failure).
  - Single group membership.
- $multicast(g, m)$ sends message m to all members of g.
- $deliver(m)$ delivers the multicast message to the calling process.
- Deliver vs. receive.
- Messages include identifiers: sender identifier $sender(m)$ and destination group identifier $group(m)$.

Coordination and agreement in group communication

Basic multicast (B-multicast)

- B-multicast ensures delivery.
- B-deliver for message delivery.
- Implemented with reliable sends.
  - To B-multicast(g, m) : for each process $p \in g$, call $send(p, m)$
  - On $receive(m)$ at p: call B-deliver(m) at p.
- Implementations use threads.
  - Threads can cause issues.
- More practical implementation using IP multicast.

Coordination and agreement in group communication

Reliable multicast

- Operations: R-multicast, R-deliver.
- Reliable multicast properties:
  - Integrity: One-time delivery, traceability of sender.
  - Validity: All messages eventually delivered.
  - Agreement: Consistent delivery (all or nothing).

Coordination and agreement in group communication

Implementing reliable multicast over B-multicast

Reliable multicast algorithm

On initialization
  Received := {};

For process p to R-multicast message m to group g
  B-multicast(g, m); // $p \in g$ is included as a destination

On B-deliver(m) *at process q with* g = *group*(m)
  if ($m \notin$ *Received*)
  then
    Received := *Received* $\cup$ { m } ;
    if ($q \neq p$) *then B-multicast*(g, m); *end if*
    R-deliver m;
  end if

Coordination and agreement in group communication

Implementing reliable multicast over B-multicast

- Initialization:
  - Empty list of received messages.
- Sending a Message (R-multicast):
  - Use B-multicast method.
- Receiving a Message (R-deliver):
  - Checks if the message is means for the group.
  - If so, check received messages list. If the message is not already in the list, update the list.
  - Delivers message.

Coordination and agreement in group communication

Implementing reliable multicast over B-multicast

- Validity property met.
- Integrity property satisfied.
- Agreement ensured.
- No R-deliver indicates an issue.
- Correct in asynchronous systems.
- Inefficient message sending.

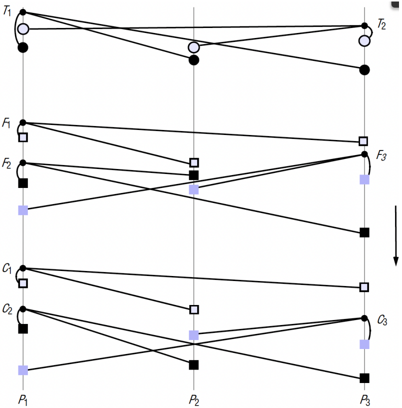
Coordination and agreement in group communication

Ordered multicast

- Arbitrary message order.
- Common ordering types:
  - FIFO Ordering: If a correct process issues $multicast(g, m)$ and then $multicast(g, m')$, then every correct process that delivers m' will deliver m before m'.
    - First message delivered first.
  - Causal Ordering: If $multicast(g, m) \rightarrow multicast(g, m')$ ($\rightarrow$ denoting happened-before relation) then any correct process that delivers m' will deliver m before m'.
    - Happened-before in delivery order.
  - Total Ordering: If a correct process delivers message m before it delivers m', then any other correct process that delivers m' will deliver m before m'.
    - Sequential message order.

Coordination and agreement in group communication

Ordered multicast



- T1, T2 total; F1, F2 FIFO; C1, C3 causal; others arbitrary.
  - Arbitrary order allowed.
    - Consistent across processes.
  - Total does not imply FIFO or causal.
  - Causal implies FIFO.
  - Define:
    - FIFO-total: Combines FIFO and total.
    - Causal-total: Combines causal and total.

Coordination and agreement in group communication

Ordered multicast

- No reliability assumption in ordered multicast.
  - Messages can be delivered independently.
- Atomic multicast: Reliable total order.
- Ordering increases latency and and consumes bandwidth.
- Application-specific semantics possible.

Coordination and agreement in group communication

The example of the bulletin board

Bulletin board: *os.interesting*		
Item	From	Subject
23	A.Hanlon	Mach
24	G.Joseph	Microkernels
25	A.Hanlon	Re: Microkernels
26	T.L'Heureux	RPC performance
27	M.Walker	Re: Mach
end		

- Bulletin board example.
- Topic-specific process groups.
- Multicast messages to users.

- Reliable multicast for all messages.
- FIFO for maintaining order of messages of a user.
- Causal for related messages.
- Total for consistent numbering.

Coordination and agreement in group communication

Implementing FIFO ordering

- Sequence numbers used in FIFO ordering.
- Here, assume a non-overlapping group.
- Variables:
  - S_g^p: Count of messages sent by process p to group g.
  - R_g^q: Latest message sequence number delivered to p from process q for group g.

Coordination and agreement in group communication

Implementing total ordering

- Ordered identifiers assigned.
  - All processes base their decisions on these.
- Delivery resembles FIFO.
- Multicast operations TO-multicast and TO-deliver.
- Two methods for assigning identifiers.

Coordination and agreement in group communication

Implementing total ordering - first method

- Process which TO-multicasts attaches an ID $id(m)$ to message m.
- Message sent to all members of g and its sequencer.
  - Sequencer maintains sequence number s_g.
- m is B-delivered and s_g is B-multicast.
- Messages stay in hold-back queue until they can be delivered according to s_g.
- Sequence numbers ensure ordering.
- Sequencer may become a bottleneck.

Coordination and agreement in group communication

Implementing total ordering

Total ordering using a sequencer

1. Algorithm for group member *p*

On initialization: $r_g := 0$;

To TO-multicast message m to group g
  B-multicast$(g \cup \{sequencer(g)\}, <m, i>)$;

On B-deliver$(<m, i>)$ *with* $g = group(m)$
  Place $<m, i>$ in hold-back queue;

On B-deliver$(m_{order} = <\text{"order"}, i, S>)$ *with* $g = group(m_{order})$
  wait until $<m, i>$ in hold-back queue and $S = r_g$;
  TO-deliver m;   // (after deleting it from the hold-back queue)
  $r_g := S + 1$;

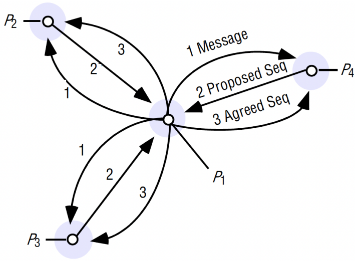
2. Algorithm for sequencer of *g*

On initialization: $s_g := 0$;

On B-deliver$(<m, i>)$ *with* $g = group(m)$
  B-multicast$(g, <\text{"order"}, i, s_g>)$;
  $s_g := s_g + 1$;

Coordination and agreement in group communication

Implementing total ordering - second method



- Total order with collective agreement.
- p B-multicastsits message to g.
- Receiving qs propose sequence numbers for messages on arrival.
- p decides and B-multicasts the agreed number.

Coordination and agreement in group communication

Implementing total ordering - second method

- Two values maintained:
  - Largest agreed sequence number.
  - Own largest proposed number.
- Steps for multicast:
  - ps B-multicasts $\langle m, i \rangle$ with ID i.
  - qs respond with proposed numbers.
  - p collects all and selects largest a.
  - p B-multicasts agreed number $\langle a, i \rangle$.
  - qs update and reorder messages in hold-back queues.
  - Transfer message from hold-back queue to delivery queue, when its agreed sequence number is assigned.

Coordination and agreement in group communication

Implementing total ordering - second method

- Total ordering achieved collectively.
  - Let m_1 be assigned an agreed sequence number and be at the front of hold-back.
  - Let m_2 be not yet assigned an agreed sequence number.
  - Sequence numbers should be monotonically increasing.

$$agreedSequence(m_2) \geq proposedSequence(m_2)$$

  - If m_1 has an agreed sequence number and is at the front of the queue, any message m_2 received later will have a larger sequence number.

$$proposedSequence(m_2) > agreedSequence(m_1)$$

  - Agreed sequence number for m_2 will always be greater than that of m_1.

$$agreedSequence(m_2) > agreedSequence(m_1)$$

Coordination and agreement in group communication

Implementing causal ordering

- Causal multicast operations: CO-*multicast* and CO-*deliver*.
- Vector timestamps maintained by each process p.
- p increments its timestamp, B-multicasts m and its timestamp to g.
- p_i places m from p_j in hold-back queue before delivery.
  - Check vector timestamps.
  - Wait for earlier messages from p_j.
  - Wait for messages delivered by p_j at the time it sent m.

Causal ordering using vector timestamps

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

On initialization

$V_i^g[j] := 0$ ($j = 1, 2 \ldots, N$);

To CO-multicast message m to group g

$V_i^g[i] := V_i^g[i] + 1$;

B-multicast(g, $<V_i^g, m>$);

On B-deliver($<V_j^g, m>$) *from* p_j ($j \neq i$), *with* $g = group(m)$

place $<V_j^g, m>$ in hold-back queue;

wait until $V_j^g[j] = V_i^g[j] + 1$ and $V_j^g[k] \leq V_i^g[k]$ ($k \neq j$);

CO-deliver m; // after removing it from the hold-back queue

$V_i^g[j] := V_i^g[j] + 1$;