

Advanced Distributed Systems

Karl M. Göschka
Karl.Goeschka@tuwien.ac.at

[http://www.infosys.tuwien.ac.at/teaching/courses/
AdvancedDistributedSystems/](http://www.infosys.tuwien.ac.at/teaching/courses/AdvancedDistributedSystems/)

ACID properties

- „All or nothing“:
 - Failure **A**tomicity: Effects are atomic even if server crashes.
 - **D**urability: If completed successfully, effects can only be changed by subsequent (new) transactions.
- **I**solation: Intermediate effects not be visible for others
- **C**onsistency (data integrity): not in the *responsibility* of transaction handling, but
 - may have to be relaxed during transactions
 - can be compromised by weak isolation

Recoverable
objects!

Serially
equivalent!

Atomicity

- Distributed commit: 2PC, 3PC
- Distributed recovery
- Reliable multicast and failure atomicity
- Multicast ordering
- Static and dynamic group membership
- Atomic multicast and virtual synchrony

Two-Phase Commit (2PC)

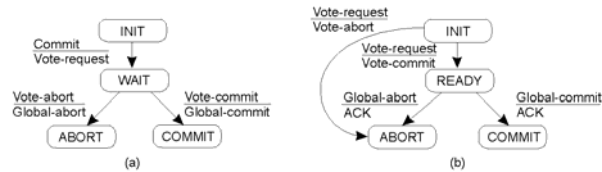
Voting phase

- Coordinator sends **vote-request**
- Participants return **vote-commit** or **vote-abort**

Commit phase

3. Coordinator sends **global-commit** or (if only one participant aborted) **global-abort**
4. Participants **that voted for commit** wait for **final decision** to commit or abort

2PC: Finite state machines



- a) The finite state machine for the coordinator.
 b) The finite state machine for a participant.

2PC: Problems with failures

- ❑ Blocking states may wait for crashed processes
 → timeout mechanisms:
 - Participant in INIT: local abort, send „vote-abort“ to coordinator
 - Coordinator in WAIT: send „global-abort“ to all participants
 - Participant in READY: → can not simply decide!
 - ❑ block until coordinator recovers
 - ❑ contact another participant Q → next slide

2PC: Participant in READY

State of Q	Action by P
COMMIT	Make transition to COMMIT
ABORT	Make transition to ABORT
INIT	Make transition to ABORT
READY	Contact another participant

- ❑ Actions taken by a participant *P* when residing in state *READY* and having contacted another participant *Q*.
- ❑ If **all other participants** „READY“ → no decision, coordinator's vote needed, blocking (rare case) → „blocking“ commit protocol

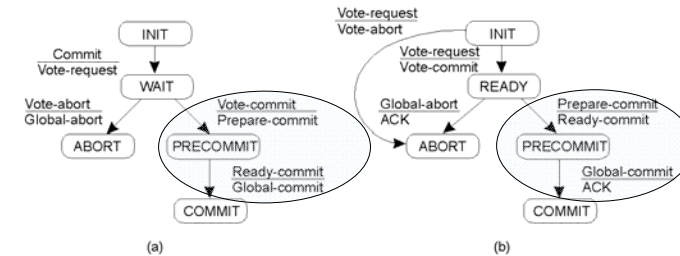
2PC: Crash and recovery

- ❑ Participant in INIT: recover to local ABORT and send „vote-abort“ to coordinator
- ❑ Participant in COMMIT/ABORT: recover to same state and retransmit decision to coord.
- ❑ Participant in READY: When recovering it **can not decide** → contact other participants
- ❑ Coordinator in WAIT: retransmit „vote-request“ after recovery
- ❑ Coordinator in COMMIT/ABORT: recover decision and retransmit „global-commit/abort“

2PC: Avoid blocking

- Each participant multicasts received messages to all others
- Three-phase commit protocol
 - not applied often in practice
 - two conditions:
 1. No single state from which it is possible to make a transition to either COMMIT or ABORT
 2. No state, in which it is **not possible** to make a **final decision**, and from which a **transition to COMMIT** can be made
 - these two conditions are sufficient and necessary for a **non-blocking commit protocol**

Three-Phase Commit (3PC)



- a) Finite state machine for the coordinator in 3PC
- b) Finite state machine for a participant

3PC: Differences

- Phase 1: Participants are in INIT, ABORT, and READY
- Phase 2: All participants are
 - either in READY + ABORT (2A)
 - or in READY + PRECOMMIT (2B)
- Phase 3: All participants are in PRECOMMIT and COMMIT
- **Coordinator blocked** in PRECOMMIT: Participant crashed, but voted commit → **global-commit** oK (Participant will later recover accordingly and commit)

3PC: Differences

- If a participant is **READY**, then others are
 - INIT, ABORT, READY (phase 1) or
 - ABORT, READY (phase 2A) or
 - PRECOMMIT, READY (phase 2B)
- If a participant is **PRECOMMIT**, then others are
 - PRECOMMIT and READY (phase 2B) or
 - PRECOMMIT and COMMIT (phase 3)
- → no participant can be in state INIT, when another participant is in PRECOMMIT

3PC: Crash and recovery

- Participant blocked in **PRECOMMIT**: Coordinator crashed, contact other participants:
 - if COMMIT → commit
 - if majority of participants → precommit → commit
 - precommitted/committed participant may become **new coordinator** to finish the protocol (the other participants and the coordinator will recover accordingly)
 - Majority: against different decisions („split-brain“) in partitions (majority voting)

3PC: Crash and recovery

- Participant blocked in **READY**: Coordinator crashed, contact other participants:
 - if INIT, ABORT → abort
 - if **each** is READY (and **majority**) → abort!
→ Crashed participants may recover to INIT or ABORT, so only abort is safe (READY and PRECOMMIT can still abort!)
 - if **one** is PRECOMMIT (and **majority**) → first precommit. Next round (if **all** are PRECOMMIT) → commit!
→ Crashed participants may recover to COMMIT, so only commit is safe (READY and PRECOMMIT can commit as well!)

Atomicity

- Distributed commit: 2PC, 3PC
- Distributed recovery
- Reliable multicast and failure atomicity
- Multicast ordering
- Static and dynamic group membership
- Atomic multicast and virtual synchrony

Recovery

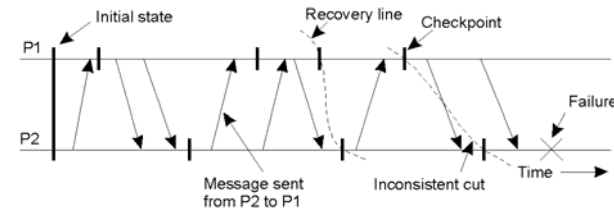
- Backward recovery:
 - previously correct state
 - checkpoints: record and restore states
 - generally applicable
 - costly
 - no failure avoidance
 - some operations are irreversible (e.g. ATM)
- Forward recovery:
 - bring the system in correct new state
 - error has to be known in advance

Checkpointing

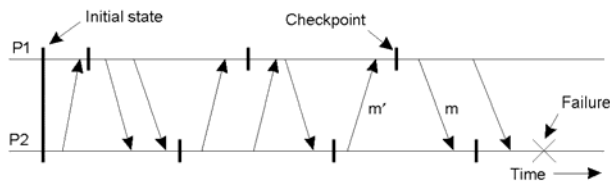
- Record a consistent global state (distributed snapshot)
- Recovery line: most recent consistent cut
- Construct consistent global state from local states:
 - Independent checkpointing: Find a recovery line → domino effect
 - Coordinated checkpointing: synchronized

Checkpointing

- A recovery line.



Independent Checkpointing



- The **domino effect**: all cuts are inconsistent → falls back to the initial state!

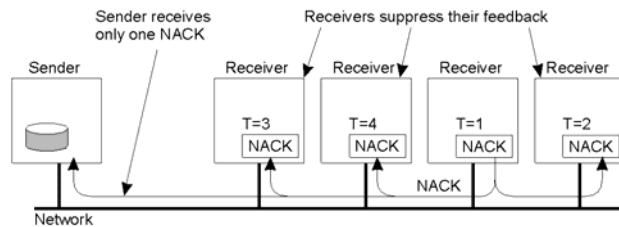
Atomicity

- Distributed commit: 2PC, 3PC
- Distributed recovery
- Reliable multicast and failure atomicity
- Multicast ordering
- Static and dynamic group membership
- Atomic multicast and virtual synchrony

Reliable multicast

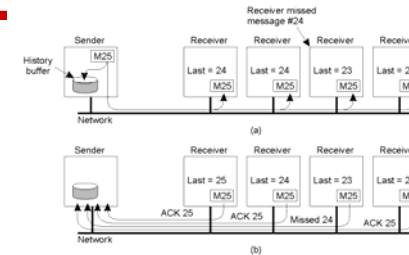
- Multicast is an essential element of many distributed algorithms
- Example: process groups, active replication
- A reliable multicast (group communication) is necessary for providing fault-tolerant distributed algorithms
- Group membership:
 - static: processes do not fail, join, leave
 - dynamic: reliable = delivery to all non-faulty group members, but agreement is needed, what the group currently looks like when a message is to be delivered

Nonhierarchical Feedback Control



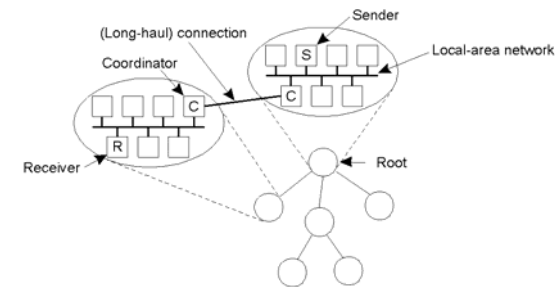
- Feedback suppression:
 - NACK only
 - first (multicast) retransmission request (after random delay) leads to the suppression of others
 - retransmission (not necessarily original sender) is also multicast
- Scales well, but accurate scheduling is difficult and feedback interrupts successful processes, too

Basic Reliable-Multicasting Schemes



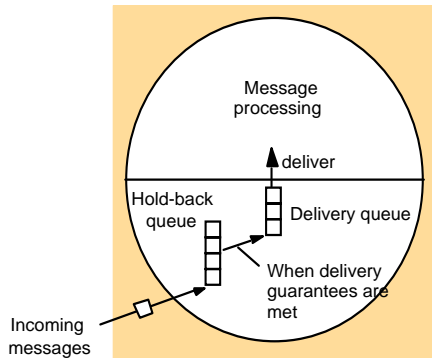
- A simple solution to reliable multicasting when **all receivers are known** and are assumed **not to fail**
 - Message transmission (multicast) and re-transmission (point-to-point)
 - Reporting feedback (piggybacked **ACK** and point-to-point **NACK**)
- Relies on message numbers; history buffer, unreliable multicast, non-faulty processes
- Scalability: **feedback implosion** vs. indefinite communication; processes retain copies of delivered messages indefinitely

Hierarchical Feedback Control



- The essence of hierarchical reliable multicasting.
 - a) Each local coordinator forwards the message to its children.
 - b) A local coordinator handles retransmission requests.
 - c) Scales well, but dynamic tree construction is a remaining problem

The hold-back queue



Atomicity

- Distributed commit: 2PC, 3PC
- Distributed recovery
- Reliable multicast and failure atomicity
- Multicast ordering
- Static and dynamic group membership
- Atomic multicast and virtual synchrony

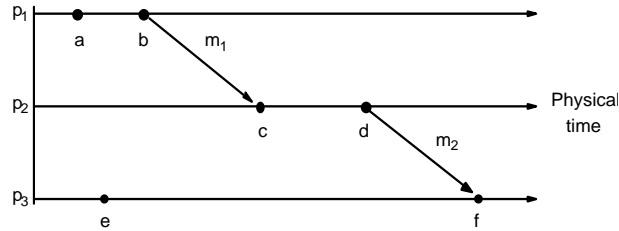
Implementation model for ordered mc

- Multicast **queue** at each server node
- Multicast messages are stored in queue on arrival
- Messages are numbered (or timestamped) in some way
- Depending on desired order of delivery, messages are **delivered from queue to the process after some coordination** with queues of other servers
- For example, if the same message is at head of queue in all queues, then it can be delivered (totally-ordered multicast)
- Ordering can be expensive, application-specific message semantics can be more efficient ("**end-to-end**"-argument)

Total, FIFO and causal ordering (1)

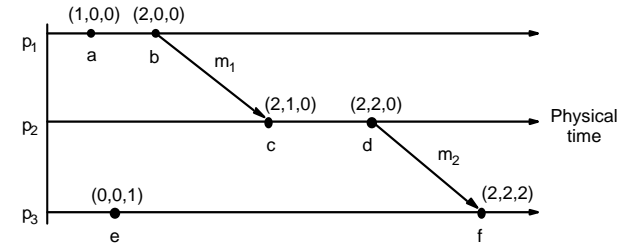
- **FIFO** ordering: If a process issues F_1 and then F_2 , then every process will deliver F_1 before F_2 (partial ordering)
- **Causal** ordering: If C_1 happened-before C_2 , then every process will deliver C_1 before C_2 (partial ordering)
- **Total** ordering: If a process delivers T_1 before T_2 , then all processes deliver T_1 before T_2
- Causal ordering implies FIFO ordering
- We do not assume or imply reliability (can be combined)

Happened-before



- Feynman (space-time) diagrams document causality
- Relationship is transitive: a happened-before f
- Imposes a partial order (not total):
 - $a \rightarrow b \rightarrow c \rightarrow d \rightarrow f$
 - $e \parallel (a, b, c, d)$, but $e \rightarrow f$

Vector Clocks - Example

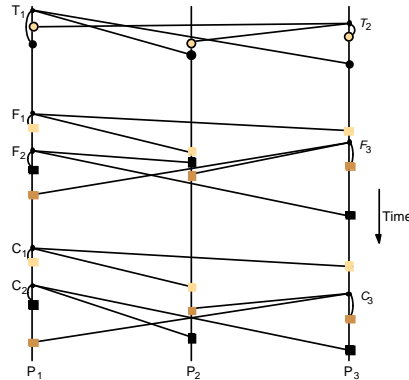


Total, FIFO and causal ordering (2)

Notice the consistent ordering of **totally** ordered messages T_1 and T_2 , the **FIFO-related** messages F_1 and F_2 and the **causally** related messages C_1 and C_3 – and the **otherwise arbitrary** delivery ordering of messages.

Hybrids:

- FIFO-total
- Causal-total



Message ordering

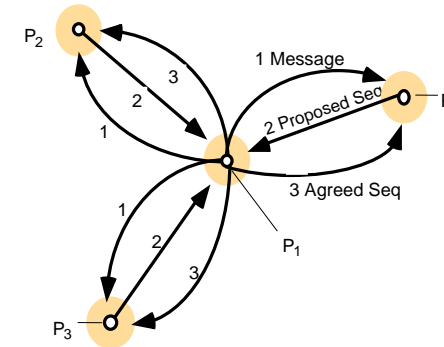
Multicast	Basic Message Ordering	Total-ordered Delivery?
Reliable multicast	None	No
FIFO multicast	FIFO-ordered delivery	No
Causal multicast	Causal-ordered delivery	No
Atomic multicast	None	Yes
FIFO atomic multicast	FIFO-ordered delivery	Yes
Causal atomic multicast	Causal-ordered delivery	Yes

- Epochs are separated by group membership changes
- Six different versions of virtually synchronous reliable multicasting regarding ordering within epochs

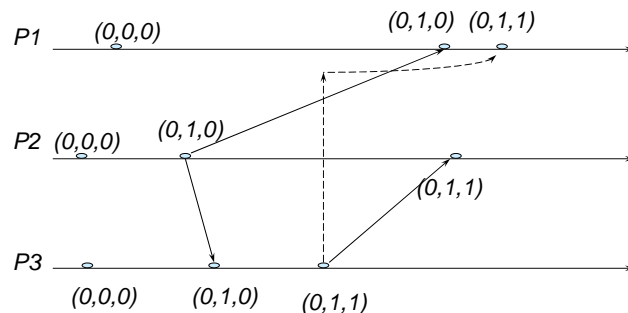
Totally-Ordered Multicasting

- ❑ clients multicast their updates with (Lamport) timestamp (FIFO, reliable)
- ❑ upon receipt, the message is put into local queue ordered by timestamp
- ❑ server acknowledges receipt of requests by multicast (for total ordering)
- ❑ eventually all processes will have the same copy of the local queue
- ❑ a message that is at the head of the queue and has been acknowledged by **all** processes is delivered to server process (respective ACKs are deleted)
- ❑ updates may not be done in “correct order” but they are done in the same order at all nodes

The ISIS algorithm for total ordering



Causal ordering using vector timestamps



Causal ordering using vector timestamps

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

On initialization

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

The number of group-g messages from process j that happened before the current message to be multicast

To CO-multicast message m to group g

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

$$B\text{-multicast}(g, \langle V_i^g, m \rangle);$$

Timestamps count the multicast messages only

On B-deliver($\langle V_j^g, m \rangle$) from p_j , with $g = \text{group}(m)$

place $\langle V_j^g, m \rangle$ 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

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

Optimization of the merge operation

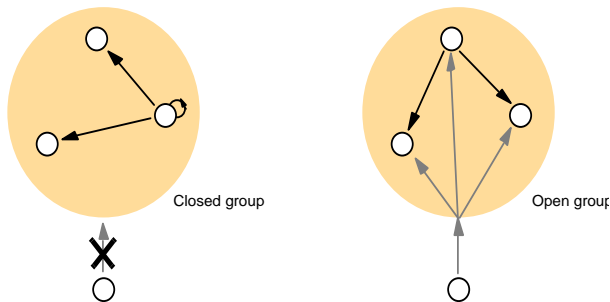
Wait conditions

- ❑ **Wait conditions:** P_i waits until it has delivered
 - 1. any earlier message from P_j (FIFO)
 - 2. any message that P_j has delivered (accepted locally) before it multicast the message m

Atomicity

- Distributed commit: 2PC, 3PC
- Distributed recovery
- Reliable multicast and failure atomicity
- Multicast ordering
- Static and dynamic group membership
- Atomic multicast and virtual synchrony

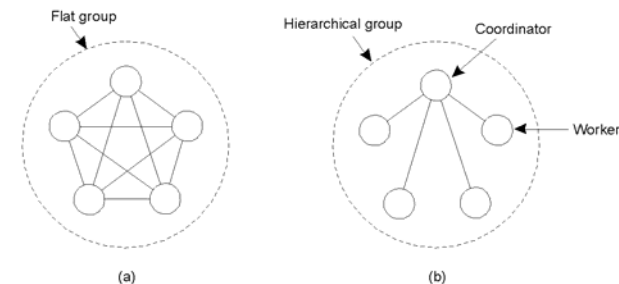
Open and closed groups



Design Issues for Process Groups

- Organize identical processes in a group
- Purpose: Collection of processes as a single abstraction
- Multicast is a key issue: all requests arrive at all servers in the same order (**atomic multicast**)
- Groups may be **dynamic**: mechanisms are needed to manage groups and group memberships
- Open groups vs. closed groups
- Flat groups vs. hierarchical groups
- A process can be member in several groups

Flat and hierarchical groups

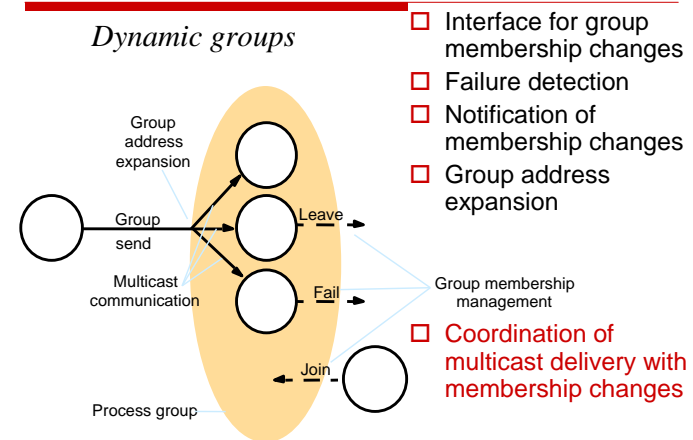


- How can a message be delivered to all members of a group?
- **Flat** group: no single point of failure
- (Simple) **hierarchical** group: Co-ordinator; decision making is easier

Group membership

- Creating and deleting groups; processes joining and leaving (or crashing)
 - **group server**: easy, efficient, but single point of failure
 - **distributed group membership service** (e.g. by reliable multicasting)
- Joining and leaving operations must be **synchronous** with data messages (e.g. by converting this operation into a sequence of messages sent to the whole group)
- **Crashing** may be more difficult to **detect** (fail-stop is too strong, usually fail-silent can be assumed)
- How to rebuild a group consistently.

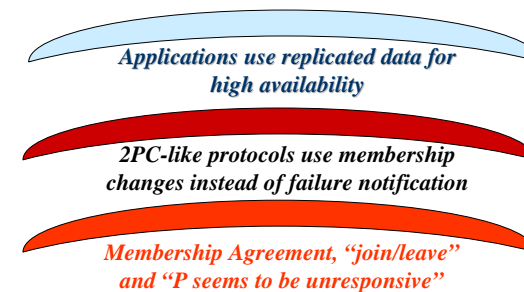
Group membership services



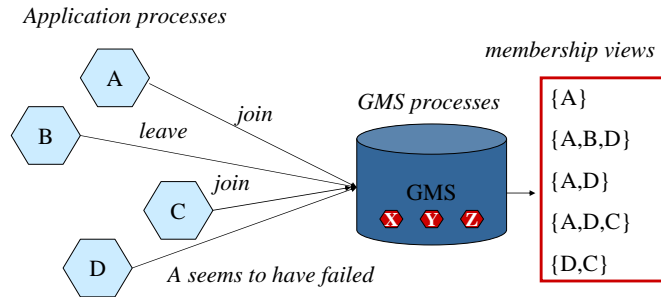
Agreement on Membership

- Recall:
 - Detecting failure is a lost cause.
 - Too many things can mimic failure
 - To be accurate would end up waiting for a process to recover (transactions)
 - Substitute agreement on membership
 - Now we can drop a process because it isn't fast enough
 - This can seem "arbitrary", e.g. A kills B...
- GMS implements this service for everyone else
- See Birman "Reliable Distributed Systems"

Architecture



Architecture



GMS majority requirement

- To move from system "view" i to view $i+1$, GMS requires explicit acknowledgement by a majority of the processes in view i
- Can't get a majority: causes GMS to lose its primaryness information
- Dahlia Malkhi has extended GMP to support partitioning and remerging; similar idea used by Yair Amir and others in Totem system

GMP protocol itself

- Used only to track membership of the "core" GMS
- Designates one GMS member as the coordinator
- Switches between 2PC and 3PC
 - 2PC if the coordinator didn't fail and other members failed or are joining
 - 3PC if the coordinator failed and some other member is taking over as new coordinator
- Question: how to avoid "logical partitioning"?

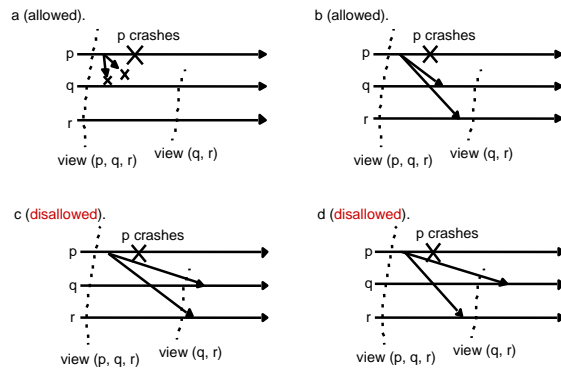
What if system has thousands of processes?

- Idea is to build a GMS subsystem that runs on just a few nodes
- GMS members track themselves
- Other processes ask to be admitted to system or for faulty processes to be excluded
- GMS treats overall system membership as a form of replicated data that it manages, reports to its "listeners"

Atomicity

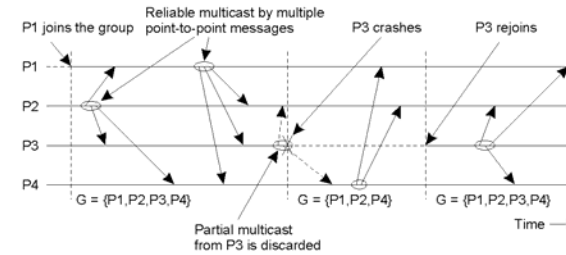
- Distributed commit: 2PC, 3PC
- Distributed recovery
- Reliable multicast and failure atomicity
- Multicast ordering
- Static and dynamic group membership
- Atomic multicast and virtual synchrony

View-synchronous group communication



Virtual Synchrony

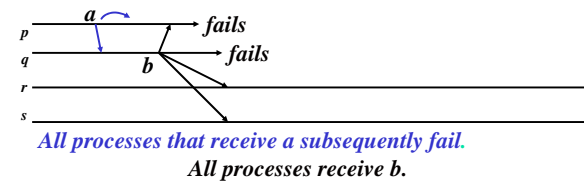
Concepts: **Group view** and **view delivery**



- Either all (non-faulty) processes in the group receive the multicast in the **same view**, or none receives it (**agreement**, atomicity)
- The **view delivery** itself is **totally ordered**

Atomic delivery

- Atomic or failure atomic delivery
 - If any process receives the message and remains operational, all operational destinations receive it



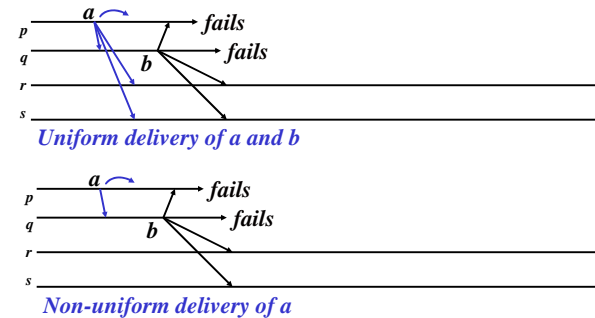
Additional properties

- A multicast is dynamically uniform if:
 - If any process delivers the multicast, all group members that don't fail will deliver it (even if the initial recipient fails immediately after delivery).
- Otherwise we say that the multicast is "not uniform"

Stronger properties cost more

- Weaker ordering guarantees are cheaper than stronger ones
- Non-uniform delivery is cheap
- Dynamic uniformity is costly
- Dynamic membership is cheap
- Static membership is more costly

Uniform and non-uniform delivery

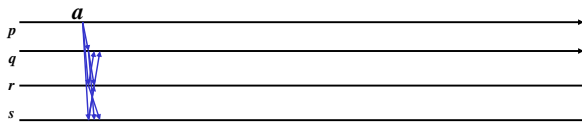


Implementing multicast primitives

- Initially assume a static process group
- Crash failures: permanent failures, a process fails by crashing undetectably. No GMS (at first).
- Unreliable communication: messages can be lost in the channels
- ... looks like the asynchronous model of FLP

Multicast by "flooding"

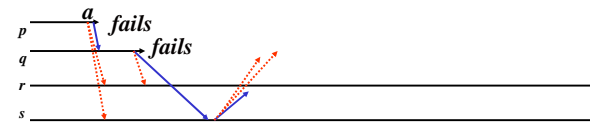
- All recipients echo message to all other recipients, $O(n^2)$ messages exchanged
- Reject duplicates on basis of message id



- When can we garbage collect the id?

Multicast by "flooding"

- All recipients echo message to all other recipients, $O(n^2)$ messages exchanged
- Reject duplicates on basis of message id



- When can we garbage collect the id?

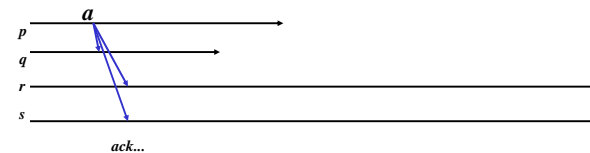
Garbage collection issue

- Must remember id as long as might still see a duplicate copy
- If no process fails: garbage collect after echoed by all destinations
- Very similar to 3PC protocol

... correctness of this protocol depends upon **having an accurate way to detect failure!**
Return to this point in a few minutes.

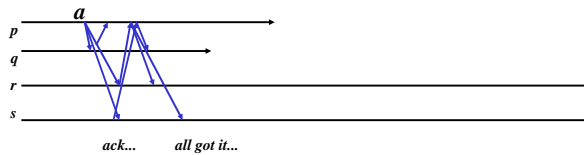
"Lazy" flooding and garbage collection

- Idea is to delay "non urgent" messages
- Recipients delay the echo in hope that sender will confirm successful delivery: $O(n)$ messages



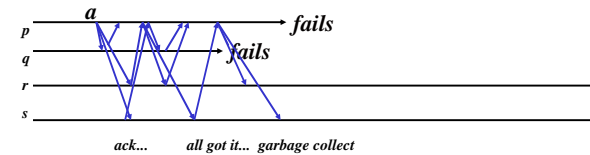
"Lazy" flooding

- Recipients delay the echo in hope that sender will confirm successful delivery: $O(n)$ messages



"Lazy" flooding

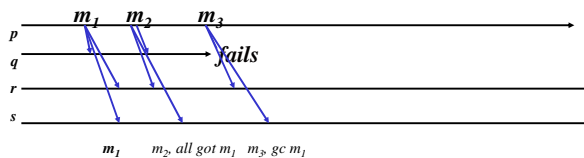
- Recipients delay the echo in hope that sender will confirm successful delivery: $O(n)$ messages



- Notice that garbage collection occurs in 3rd phase

"Lazy" flooding, delayed phases

- "Background" acknowledgements (not shown)
- Piggyback 2nd, 3rd phase on other multicasts



Lazy scheme continued

- If sender fails, recipients switch to flood-style algorithm
 - ... but now we have the same garbage collection problem: if sender fails we may never be able to garbage collect the id!
- Problem is caused by lack of failure detector

Garbage collection with inaccurate failure detections

- ... we lack an accurate way to detect failure
 - If any does seem to fail, but is really still operational and merely partitioned away, the connection might later be fixed.
 - That process might “wake up” and send a duplicate
- Hence, if we are not sure a process has failed, can't garbage collect our duplicate-suppression data yet!

Now our lazy scheme works!

- Garbage collect when **all non-faulty** processes are known to have received the message
- Use process ranking to pick a new “coordinator” if the initial one fails
- Cost only reaches n^2 if many fail during protocol
- Can delay 2nd, 3rd round if desired

Exploiting a failure detector

- Suppose that we had a failstop environment
- Process group membership managed by oracle, perhaps the GMS we saw earlier
- Failures reported as “**new group views**”
- All see the same sequence of views:
 - $G = \{p,q,r,s\} \{p,r,s\} \{r,s\}$
- Now can assume failures are accurately detected

Dynamic uniformity

- This property requires an extra phase of communication
- Phase 1: distribute message
- Phase 2: can deliver if all non-faulty processes received it in phase 1
- Insight: no process delivers a message until all have received it

Summary

- Transactions focus (primarily) on recoverability and serializability
 - Distributed Commit: 2PC, 3PC
 - Recovery: Checkpointing, message logging
- Group communication and group membership focus on order-based consistency guarantees
 - reliable multicasting (failure atomicity)
 - ordering guarantees
 - dynamic group membership (faster, but weaker agreement guarantees than static membership)
 - virtual synchrony