Group Communication

- Point-to-point *vs.* one-to-many
- Multicast communication
- Atomic multicast
- Virtual synchrony
- Group management
- ISIS

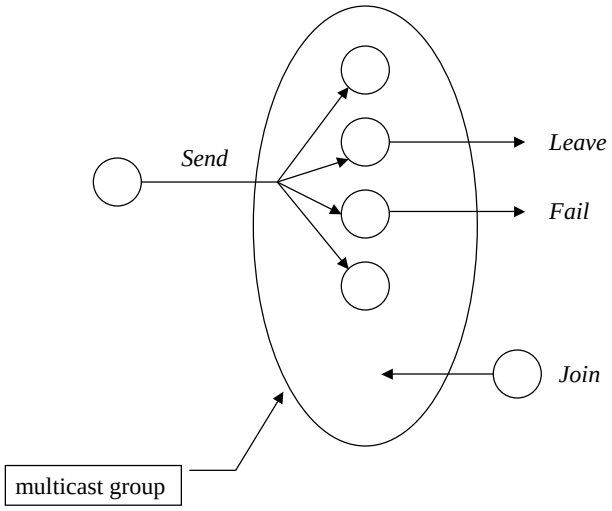
Reading:

- *Coulouris: Distributed Systems, Addison Wesley, Chapter 4.4, Chapter 11*

Group Communication: Introduction

- One-to-many communication
- Dynamic membership
- Groups can have various communication patterns
  - peer group
  - server group
  - client-server group
  - subscription (diffusion) group
  - hierarchical groups

Group Membership Management



Multicast Communication

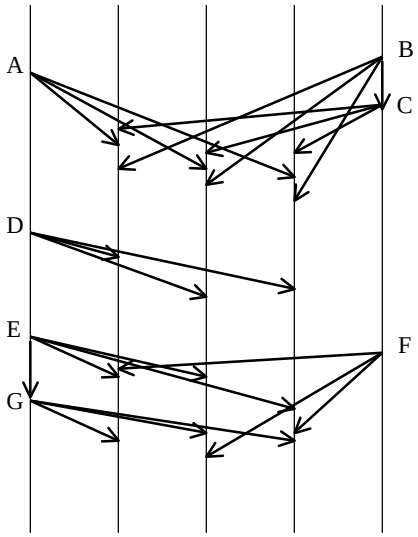
- Reliability guarantees:
  - *Unreliable* multicast: Attempt is made to transmit the message to all members without acknowledgement.
  - (*Reliable* multicast: Message may be delivered to some but not all group members.)
  - *Atomic* multicast: All members of the group receive message, or none of them do.

- Message **reception**: message has been received and buffered in the receiver machine. Not yet delivered to the application.
- Message **delivery**: The previously received message is delivered to the application.

Multicast Communication: Message Ordering

- *Globally (chronologically) ordered* multicast: All members are delivered messages in order they were sent.
- *Totally (consistently) ordered* multicast: Either m_1 is delivered before m_2 to all members, or m_2 is delivered before m_1 to all members.
- *Causally ordered* multicast: If the multicast of m_1 *happened-before* the multicast of m_2, then m_1 is delivered before m_2 to all members.
- *Sync-ordered* multicast: If m_1 is sent with sync-ordered multicast primitive, and m_2 is sent with *any ordered* multicast primitive, then either m_1 is delivered before m_2 at all members, or m_2 is delivered before m_1 at all members.
- *Unordered* multicast: no particular order is required on how messages are delivered.

Message Ordering: Examples

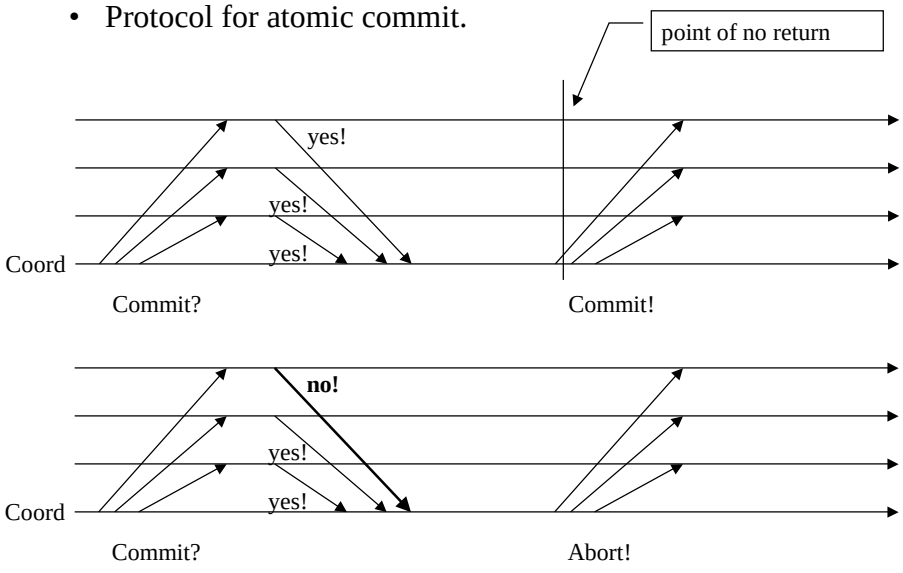


Atomic Multicast

- Simple multicast algorithm: Send a message to every process in the multicast group, using reliable message passing mechanism (e.g. TCP).
  - Is not atomic: does not handle processor failures.
- "Fix" to simple multicast algorithm: Use 2-phase-commit (2PC) technique and treat multicast as transaction.
  - Works, but correctness guarantees stronger than necessary
  - 1. If sending process *s* fails to obtain ack from process *p*, *s* must abort delivery of message.
  - 2. If *s* fails after delivering *m* to all processors, but before sending "commit" message, delivery of *m* is blocked until *s* recovers.
- 2PC protocol does more work than is really necessary.

2-Phase-Commit Protocol

- Protocol for atomic commit.



Basic 2-Phase-Commit

Coordinator

- multicast: *ok to commit*?
- collect replies
  - all *ok* => send *commit*
  - else => send *abort*

Participant:

- *ok to commit* =>
  save to temp area, reply *ok*
- *commit* =>
  make change permanent
- *abort* =>
  delete temp area

Handling Participant Failures in 2PC

Coordinator

- multicast: *ok to commit*?
- collect replies
  - all *ok* =>
    - log "commit" to "outcomes" table
    - send *commit*
  - else =>
    - send *abort*
- collect acknowledgements
- garbage-collect "outcome" information

Participant:

- *ok to commit* =>
  save to temp area, reply *ok*
- *commit* =>
  make change permanent
- *abort* =>
  delete temp area

- after failure:
  for each pending protocol, contact coordinator to learn outcome

Handling Participant Failures in 2PC

Coordinator
- multicast: *ok to commit*?
- collect replies
  - all *ok* =>
    - log "commit" to "outcomes" table
    - wait until on persistent storage
    - send *commit*
  - else
    - send *abort*
- collect acknowledgements
- garbage-collect "outcome" information

after failure

for each pending protocol in "outcomes" table

send outcome (commit or abort)

wait for acknowledgements

garbage-collect "outcome" information

Participant:

first time message received:
- *ok to commit* =>
  save to temp area, reply *ok*
- *commit* =>
  make change permanent
- *abort* =>
  delete temp area

Message is a duplicate (recovering coordinator)
- send acknowledgement

After failure:
for each pending protocol,
contact coordinator to learn outcome

Dynamic Group Membership Problem

- Dynamic Uniformity: Any action taken by a process must be consistent with subsequent actions by the operational part of system.
- D.U. not required whenever the operational part of the system is taken to "define" the system, and the states and actions of processes that subsequently fail can be discarded.
- D. U. vs. commit protocols:
  - Commit protocol: If any process commits some action, all processes will commit it. This obligation holds within a statically defined set of processes: a process that fails may later recover, so the commit problem involves an indefinite obligation with regard to a set of participants that is specified at the outset. In fact, the obligation even holds if a process reaches a decision and then crashes without telling any other process what that decision was.
  - D.U.: The obligation to perform an action begins as soon as any process in the system performs that action, and then extends to processes that remain operational, but not to processes that fail.

The Group Membership Problem

- Group Membership Service (GMS) maintains membership of distributed system on behalf of processes.
- Operations:

Operation	Function	Failure Handling
`join(proc-id, callback) returns(time, GMS-list)`	Calling process is added to membership list. Returns logical time and list of current members. Callback invoked whenever core membership changes	Idempotent: can be reissued with same outcome.
`leave(proc-id) returns void`	Can be issued by any member of the system. GMS drops specified process from membership list and issues notification to all members of the system. Process must re-join.	Idempotent.
`monitor(proc-id,callback) returns callback-id`	Can be issued by any member of the system. GMS registers a callback and will invoke callback(proc-id) later if the designated process fails.	Idempotent.

Implementing a GMS

- GMS itself needs to be highly available.
- GMS server needs to solve the GMS problem on its own behalf.
- **Group Membership Protocol** (GMP) needed for membership management in GMS (few processes), while more light-weight protocol can be used for the remainder of the system (with large numbers of processes).
- The specter of **partitions**: What to do when single GMS splits into multiple GMS sub-instances, each of which considers the other to be faulty? $\Rightarrow$ **primary partition**
- Merging partitions?

A Simple Group Membership Protocol

- Failure detection by time-out on ping operations.
- GMS coordinator: GMS member that has been operational for the longest period of time.
- Handling of members suspected of having failed (*shunning*)
  - Upon detection of apparent failure: stop accepting communication from failed process. Immediately multicast information about apparent failure. Receiving processes shun faulty process as well.
  - If shunned process actually operational, it will learned that it has been shunned when it next attempts to communicate. Now must re-join using a new process identifier.

A Simple Group Membership Protocol (2)

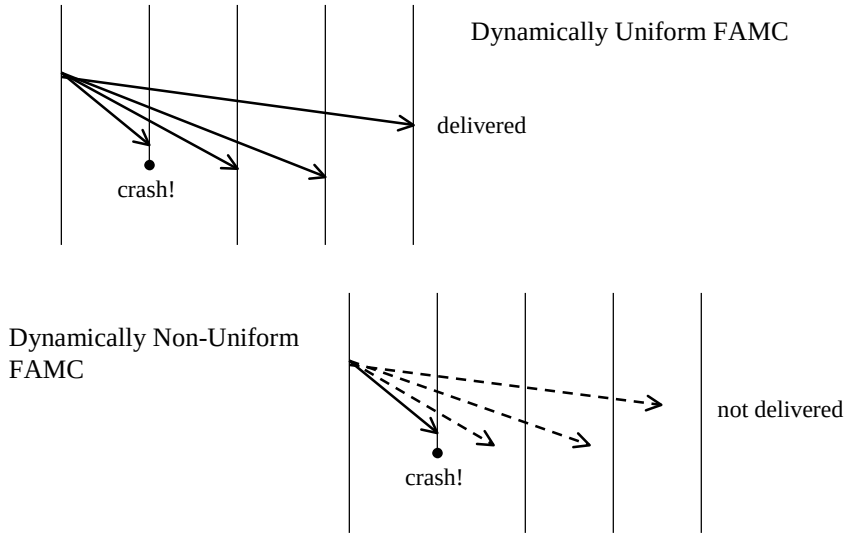
- Round-based protocol (join/leave requests)
- Two phases when old GMS coordinator not part of members to join/leave.
- First round:
  - GMS coordinator sends list of joins/leaves to all current members.
  - Waits for as many acks as possible, but requires majority from current membership.
- Second round:
  - GMS commits the update, and sends notification of failures that were detected during first round.
- Third round necessary when current coordinator is suspected of having failed, and some other coordinator must take over.
  - New coordinator starts by informing at least a majority of the GMS process listed in the current membership that coordinator has failed.
  - Then continue as before.

Atomic Multicast in Presence of Failures

Definition: **Failure-Atomic Multicast** (FAMC): For a specified class of failures, multicast will either reach all destinations, or not.

- Dynamically Uniform FAMC: If any process delivers, then all processes that remain operational will deliver, regardless of whether first process remains operational after delivering.
- Not Dynamically Uniform FAMC: If one waits long enough, one finds that either all processes that remained operational delivered, or none.
- Why do we care?

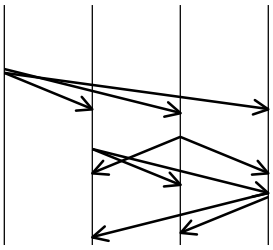
Dynamically Uniform vs. Not Dynamically Uniform



Dynamically Non-Uniform FAMC

Simple (inefficient) multicast protocol:

- Sender
  - Add header with list of members at time when message is sent.
  - Send out message to all members of the group.
- Member:
  - Upon receipt of the message, immediately deliver.
  - Resend message to all destinations.
  - Receives one message from sender and one from each non-failed receiver.
  - Discard all copies of message that arrive after message delivered.

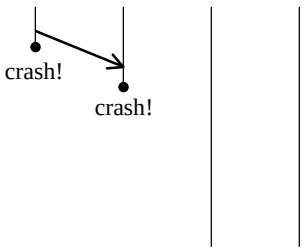


- Protocol expensive!

Dynamically Non-Uniform FAMC (2)

- The protocol is Failure-Atomic, i.e. if a process delivers message, all destinations that remain operational must also receive and deliver message.
- The protocol is **not** Dynamically Uniform Failure-Atomic:

Example:



Dynamically Uniform FAMC

- Simple modification to previous protocol:
  - Delay delivery of messages until copy has been received from every destination in current membership list provided by Group Membership Service.

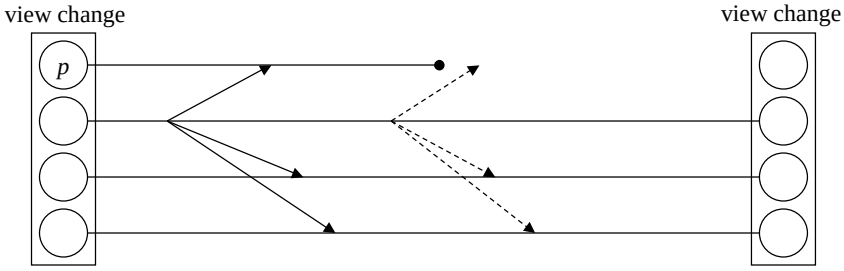
Virtual Synchrony

- "Send to all members or to none"
  - Who are the members, in particular in presence of failures?
- **Group view**: current list of members in the group.
  - Group view is consistent among all processes.
  - Members are added/deleted through **view changes**.

- **Virtually synchronous atomic multicast**:
  - 1. There is a unique group view in any consistent state on which all members of the group agree.
  - 2. If a message *m* is multicast in group view *v* before view change *c*, either no processor in *v* that executes *c* ever receives *m*, or every processor in *v* that executes *c* receives *m* before performing *c*.

Virtual Synchrony (2)

- Define *G* as set of messages multicast between any two consecutive view changes.
- All processors in a group view *v* that do not fail receive all messages in *G*.
- A processor *p* that fails may not receive all of *G*; but we know what *p* received; this simplifies recovery.



- View change managed by group membership protocol.

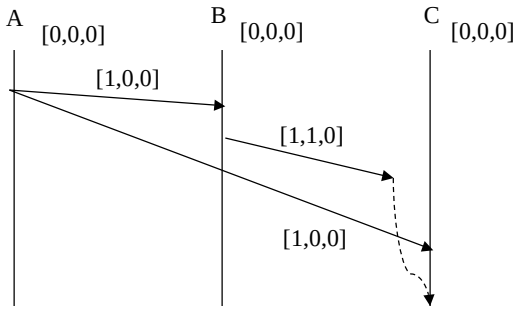
ISIS

`http://simon.cs.cornell.edu/Info/Projects/ISIS`

- Group communication toolkit
- Facilities:
  - Multicast
  - Group view maintenance
  - State transfer
- Synchrony
  - Closely synchronous
    - All common events are processes in same order (total and causal ordering)
  - Virtually synchronous
    - Failures are synch-ordered
- Multicast protocols:
  - FBCAST: unordered
  - CBCAST: causally ordered
  - ABCAST: totally ordered
  - GBCAST: sync-ordered
    - used for managing group membership

ISIS: CBCAST

- Group has *n* members
- Each member *i* maintains timestamp vector TS_i with *n* components.
- $TS_i[j]$ = timestamp of last message received by *i* from *j*.

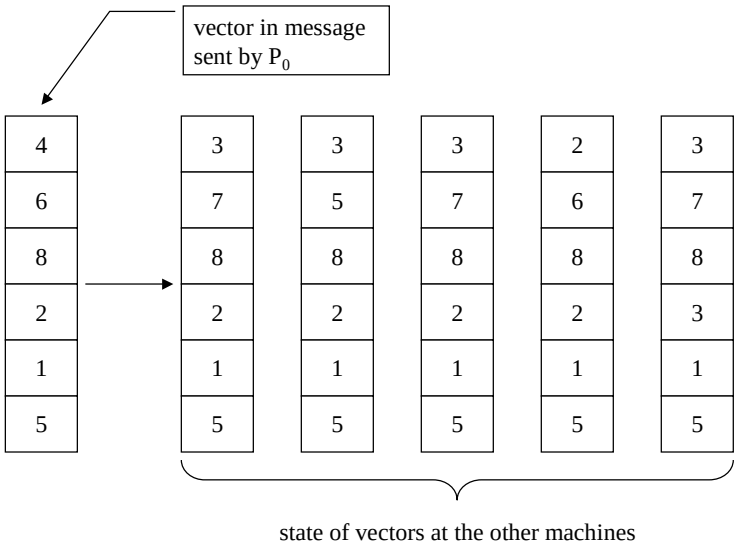


CBCAST (2)

```
mc_send(msg m, view v)
P_i: TS_i[i] := TS_i[i]+1
     send m to all members of view v
     send TS_i[] as part of message m.
```

```
mc_receive(msg m)
P_i: let P_j be sender of m
     let ts_j be timestamp vector in m
     check:
     1. ts_j[j] = TS_i[j]
        /* this is next message in sequence from P_j
           no messages have been missed. */
     2. for all k<>j: ts_j[k] <= TS_i[k]
        /* Sender has not seen a message that the
           receiver has missed. */
     If both tests passed, message is delivered, else
     it is buffered.
```

CBCAST: Example



Virtually Synchronous Group View Changes

- Virtual synchrony: all messages sent during a view v_i are guaranteed to be delivered to all operational members of v_i before ISIS delivers notification of v_{i+1}.
- Process *p* joins to produce group v_{i+1}:
  - no message of v_i is delivered to *p*
  - all messages sent by members of v_{i+1} after notification has been sent by ISIS will be delivered to *p*.
- Sender *s* fails in view v_i:
  - messages are stored at receivers until they are *group stable*.
  - if sender of non group stable message fails, holder of message is elected, and continues multicast.
- Some member *q* of v_i fails, producing v_{i+1}:
  - did *q* receive all messages in v_i?
  - did *q* send messages to other failed processes?

ABCAST: causally and totally ordered

Originally: form of 2PC protocol

1. Sender *S* assigns timestamp (sequence number) to message.
2. *S* sends message to all members.
3. Each receivers picks timestamp, larger than any other timestamp it has received or sent, and sends this to *S*.
4. When all acks arrived, *S* picks largest timestamp among them, and sends a commit message to all members, with the new timestamp.
5. Committed messages are sent in order of their timestamps.

Alternatives:

Sequencers

Interlude: Causally and Totally Ordered Communication: A Dissenting Voice

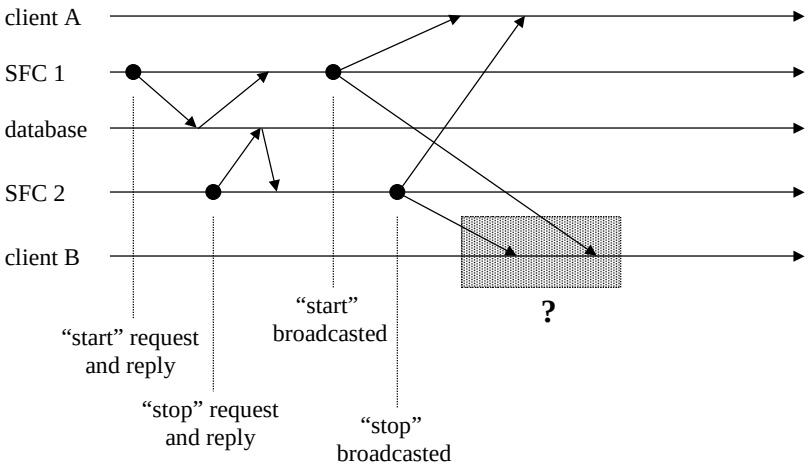
Reference: D. Cheriton and D. Skeen

"Understanding the Limitations of Causally and Totally Ordered Communication", *14th ACM Symposium on Operating Systems Principles*, 1993

- Unrecognized causality (can't say "for sure")
  - causal relationships between messages at semantic level may not be recognizable by the *happens-before* relationship on messages.
- Lack of serialization ability (can't say "together")
  - cannot ensure serializable ordering between operations that correspond to groups of messages.
- Unexpressed semantic ordering constraints (can't say "whole story")
  - many semantic ordering constraints are not expressible in *happens-before* relationship
- No efficiency gain over state-level techniques (can't say efficiently)
  - not efficient, not scalable

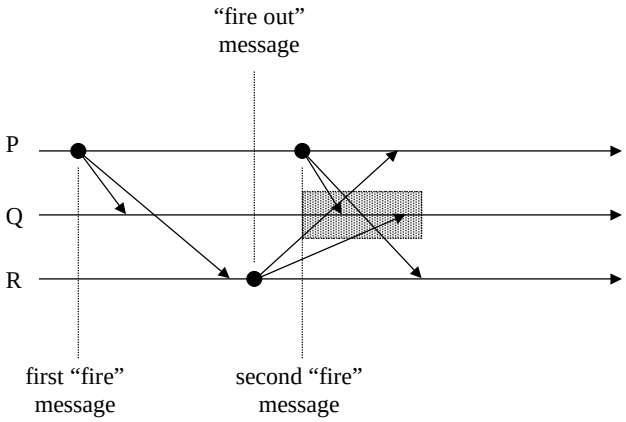
Interlude (2): Unrecognized Causality

Example 1: Shop Floor Control



Interlude (3): Unrecognized Causality

Example 2: Fire Control



Reliable Multicast Protocol

(B.Whetten,T.Montgomery,S.Kaplan.
"A High-Performance, Totally Ordered Multicast Protocol",
`ftp://research.ivv.nasa.gov/pub/doc/RMP/RMP_dagstuhl.ps`...)

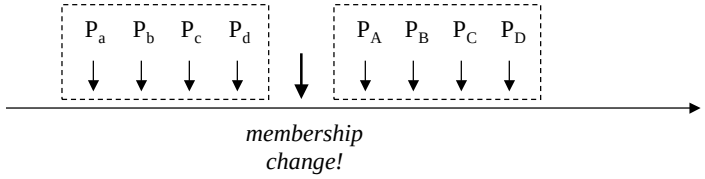
- Entities:
  - process:
    - sender/receiver of packets
  - group:
    - basic unit of group communication.
    - set of processes that receive messages sent to given IP Multicast address and port.
  - membership of a group can change over time
- Taxonomy:
  - Quality of Service
  - Synchrony
  - Fault-Tolerance

RMP: Quality of Service (QoS)

- Quality of Service related to semantics.
- unreliable
  - packet is received zero-or-more times at destination
  - no ordering
- reliable
  - packet is received at least once at each destination
- source-ordered
  - packet arrives exactly once at each destination
  - same order as sent from source
  - no ordering guarantee when more than one source
- totally ordered
  - serializes all packets to a group

RMP: Virtual Synchrony

- e.g. in ISIS (Birman *et al.*)
  - All sites see the same set of messages before and after a group membership change.



- Allows distributed applications to execute as if communication was synchronous when it actually is asynchronous.

RMP: Fault-Tolerance

- node failures, network partitions
- atomic delivery within partition:
  - If one member of the group in a partition delivers packet (to application), all members in that partition will deliver packet if they were in the group when the packet was sent.
  - No guarantee about delivery or ordering between partitions.
- *K*-resilient atomicity:
  - Totally ordered
  - Delivery is atomic at all sites that do not fail or partition, provided that no more than *K* sites fail or partition at once.
  - with *K=floor(N/2)+1* atomicity guaranteed for any number of failures.

RMP: Fault-Tolerance (cont)

- majority resilience:
  - If two members deliver any two messages, they agree on ordering of messages.
  - Guarantees total ordering across partitions, but not atomicity.
- total resilience (safe delivery):
  - Sender knows that all members received it before it can be delivered.
  - One or more sites can fail *before delivering the packet.*

Algorithms in RMP

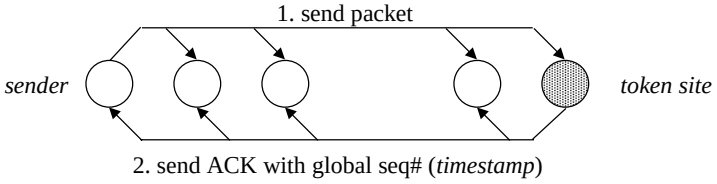
- Basic delivery algorithm
  - handles delivery of packets to members
- Membership change algorithm
  - handles membership change requests, updates view at members.
- Reformation algorithm
  - reconfigures group after failure, synchronizes members
- Multi-RPC algorithm
  - allows non-members to sent to group
- Flow control and congestion control
  - similar to Van Jacobson TCP congestion control algorithm

ACKs in Reliable Multicast

- **Def**: Packet becomes stable: Sender knows that all destinations have received packet.
- positive ACKs:
  - quick stability
  - scalability?
- cumulative ACKs:
  - parameter: number of packets per ACK
  - load *vs.* length of time for packet to go stable
- negative ACKs:
  - burden of error detection shifts to destination
  - sequence numbers
  - time to go stable unbounded
  - lost packet only detected after another packet is received.

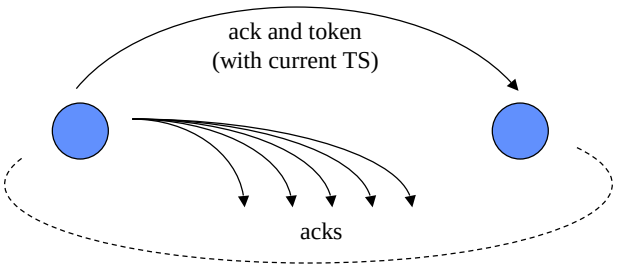
Basic Delivery Algorithm

- NACKs for reliable delivery, ACKs for total ordering and stability.
- packet ID: *{RMP proc ID, seq # for proc, QoS level}* uniquely identifies packet.



- Functions of ACK:
  - positive acknowledgment to sender ("token site has received packet")
  - allows for total and causal ordering of packets
  - timestamp as global basis for detection of dropped packets.
  - ACK contains info for more than one packet (ordered)
- Q: When does packet become stable?

Reaching Stability



- While sending ACK, token site forwards token to next process in group:
  - Before accepting token, member is required to have all packets with timestamps less than in ACK.
  - If site in group with *N* members receives token, it knows that all packets with *TS* <= *currTS-N* have been received by all members.

Basic Delivery Algorithm at Receiving Node

- Ordering of packets, detection of missing packets, buffering of packets for re-transmission.
- Each site has
  - *DataList*: contains Data packets that are not yet ordered
  - *OrderingQ*: contains slots:
    - pointer to packet
    - delivery status (missing, requested, received, delivered)
    - timestamp
- Data packet arrives: placed in *DataList*
- ACK arrives: placed in *OrderingQ*, creating one or more slots at end of queue if necessary (with info for more than one packet)
- Data packet or ACK arrives:
  - scan *OrderingQ*: match Data packets in *DataList* with slots that have been created by an ACK.
  - when match is found, Data packet is transferred to slot.
  - when hole occurs in *OrderingQ*, send out NACK, requesting for retransmission of packet.
- *OrderingQ* is “flushed” whenever token arrives.

A Cool Homepage on Multicast Protocols:

`http://hill.lut.ac.uk/DS-Archive/MTP.html`