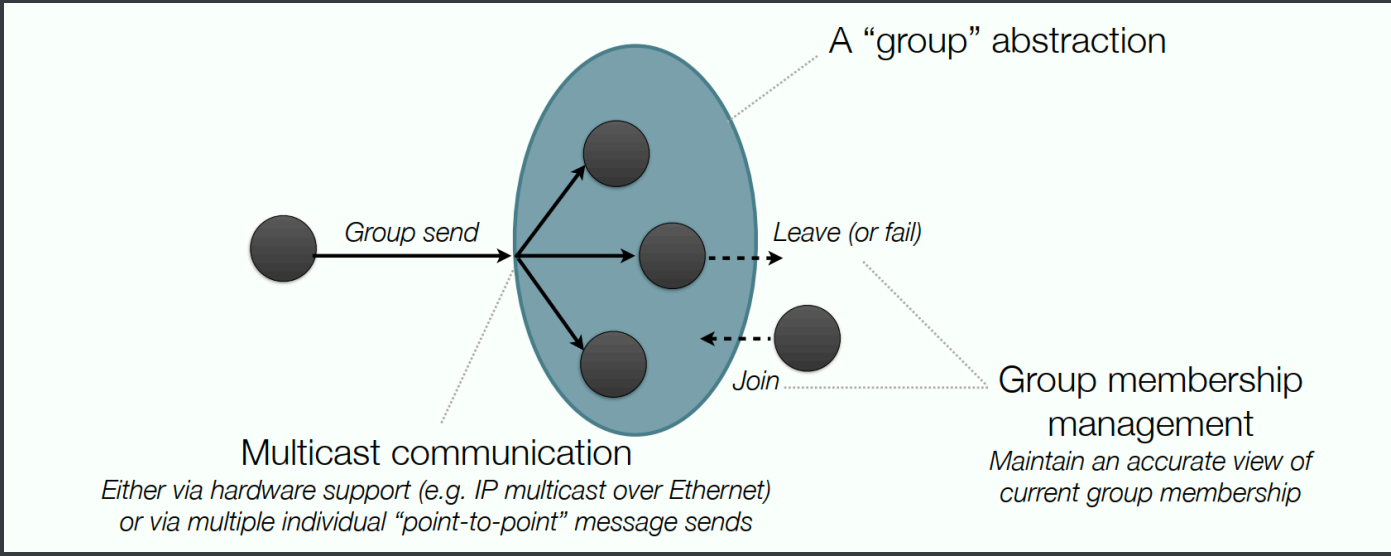
group communication & broadcast protocols

group communication: decoupling in space



- applications can define named **groups**
- processes can **join** & **leave** groups
- processes can **send messages to a group**
- definitely decoupling in space, **might** also decoupling in time if messages are queued while some group members are offline

reliable broadcast

- best effort vs. reliable

  best effort: only send once, and hope it arrives

  reliable: ackowledgements, re-transmissions

- in terms of broadcast, basic broadcast vs. reliable broadcast

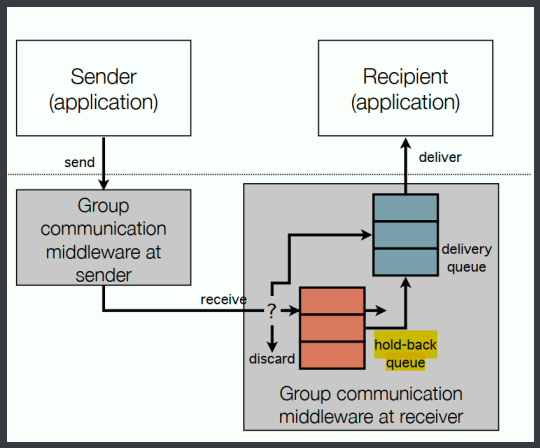
  basic ... : sender iterates over all group members

  reliable ... : if *m* is delivered to any group member, it must be delivered to all group members.

  （如果用basic broadcast的遍历方法，如果sender中途crash了，就不能保证reliable了）

- again, **receiving** vs. **delivering**

  middleware receives, then delivers to an application



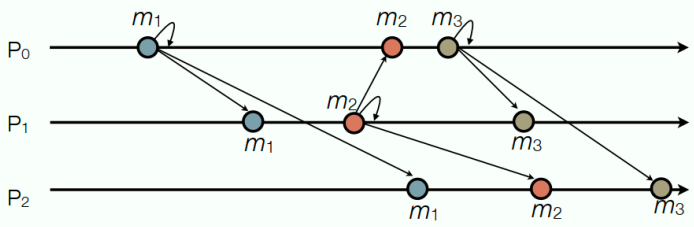
  diacard / holdback queue / delivery queue

ordered reliable multicast algorithms

每条线代表一个process，线上的一个点代表message被deliver

这些算法都不能 tolerate process crashes!

1. FIFO broadcast
   - messages sent by **the same process** must be delivered in strict order



   - Valid orders: (m_2, m_1, m_3) or (m_1, m_2, m_3) or (m_1, m_3, m_2)

   valid orders指的是deliver orders

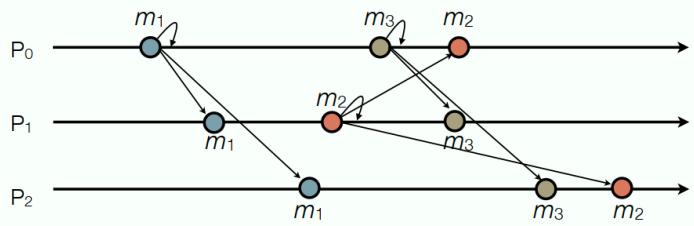
- pseudo code

```
on initialisation do
  sendSeq := 0; delivered := ⟨0, 0, . . . , 0⟩; buffer := {}
end on

on request to broadcast m at process Pi do
  send (i, sendSeq, m) via reliable broadcast;
  sendSeq := sendSeq + 1
end on

on receiving msg from reliable broadcast at process Pi do
  buffer := buffer ∪ {msg}
  while ∃ (sender, seq, m) ∈ buffer for which seq = delivered[sender] do
     deliver m to the application
     buffer := buffer \ {(sender, seq, m)}
     delivered[sender] := delivered[sender] + 1
  end while
end on
```

the `buffer` is a "hold-back" queue,

the `delivered` array keeps track of whether the previous message from a same sender, making sure of FIFO on messages from same sender

2. causal broadcast
   - causal messages must be delivered in causal order,
   - concurrent messages can be delivered in any order



- Here: $m_1 \rightarrow m_2$ and $m_1 \rightarrow m_3$ but $m_2 \parallel m_3$, so valid orders are: (m_1, m_2, m_3) or (m_1, m_3, m_2)

stronger than FIFO, always implying FIFO

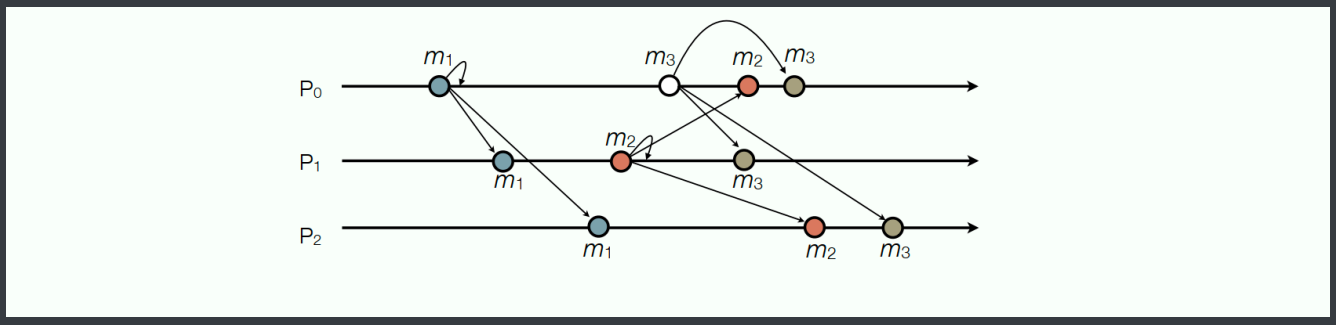
- pseudo code

```
on initialisation do
  sendSeq := 0; delivered := ⟨0, 0, . . . , 0⟩; buffer := {}
end on

on request to broadcast m at process Pi do
  deps := copy(delivered); deps[i] = sendSeq;
  send (i, deps, m) via reliable broadcast;
  sendSeq := sendSeq + 1
end on

on receiving msg from reliable broadcast at process Pi do
  buffer := buffer ∪ {msg}
  while ∃ (sender, deps, m) ∈ buffer for which deps ≤ delivered do
     deliver m to the application
     buffer := buffer \ {(sender , deps , m)}
     delivered[sender] := delivered[sender] + 1
  end while
end on
```

send entire vector `deps` instead of a single sequence nunmber;

the `<=` is the same comparison as defined for vector clocks

3. total order broadcast
   - all processes must deliver messages in the same order
   - including delivering to oneself !!!
   - order could be arbitrary, just must be the same



here, (m1, m2, m3)

- implementation:
  - **single leader** approach

    the message first gets to the single leader process, and then gets broadcast by FIFO by the leader

    but leader crashes ⛔, changing leader is difficult
  - **lamport clocks** approach

    直接改造lamport's algorithm for mutual exclusion, 只需要request和release信息，其他信息gets queued until current message is "released"
- application examples:

database replication:

to ensure consistency, all replicas should receive and apply database updates in the same order

blockchain networks:

all network nodes need to agree on the exact order of the payments

4. FIFO-total order broadcast
    - both causal & total-order
    - causal constraints real-time sequence, total-order guarantees sync

implementing ordered broadcast algorithms

2 layers:

reliable = best-effort + retransmitting

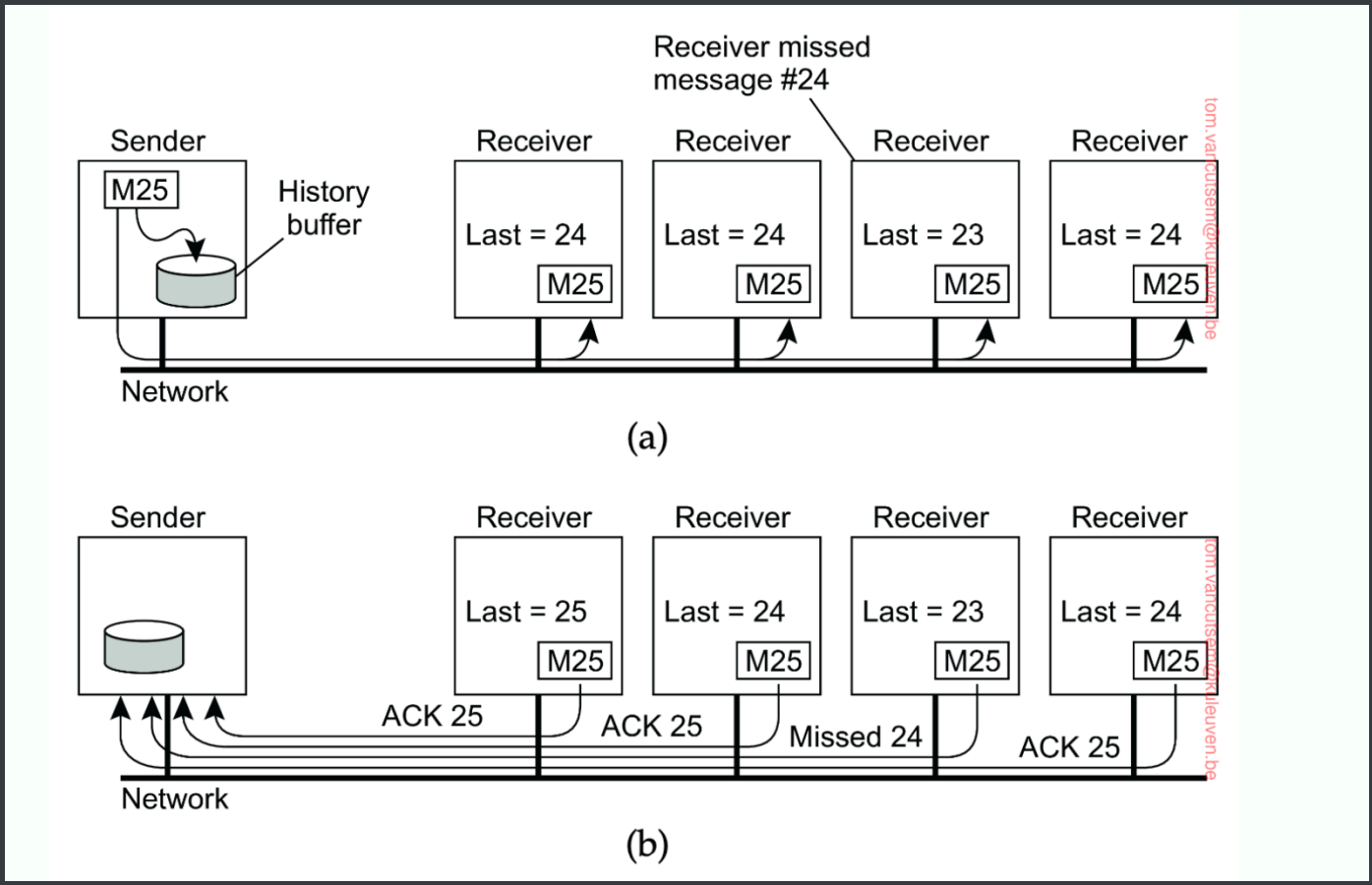
add delivery order

basic reliable broadcast with acknowledgments

sequence number to indicate the last message that a receiver has received,

so that they can tell the sender which prior messages have been missed;

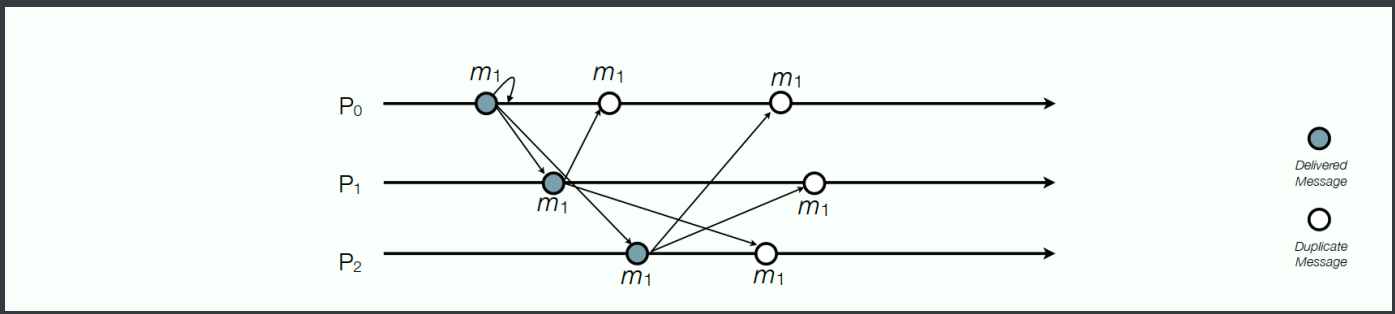
then the sender re-sends;

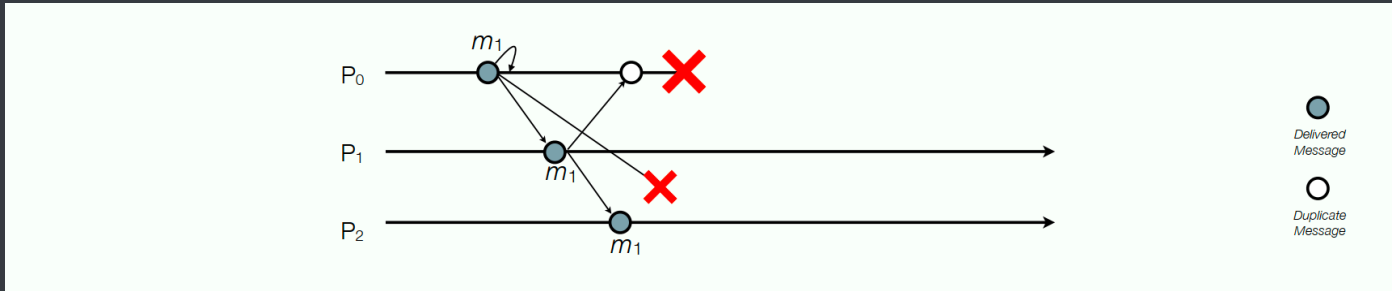


but still not ensured to be reliable **if the sender fails**;

eager reliable broadcast

the **first time** a process receives a particular message, it re-broadcasts to each other process

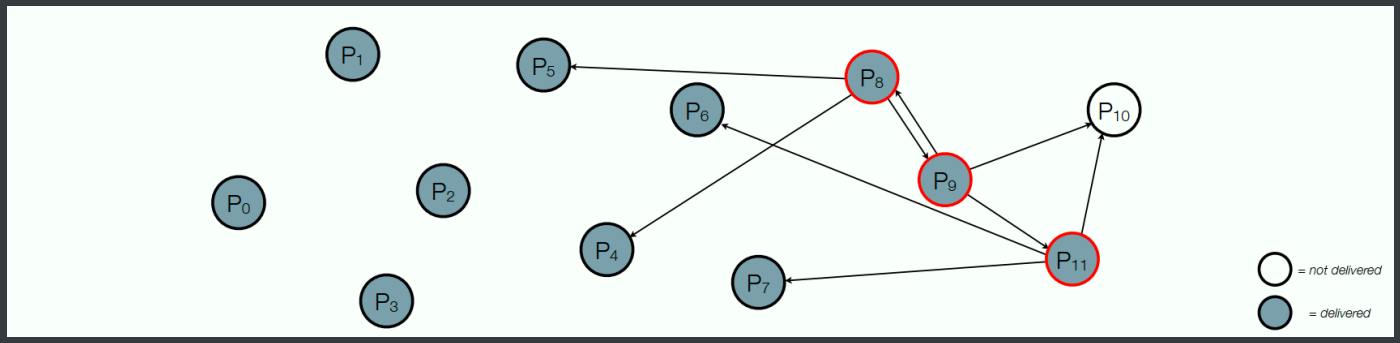




but impractical, $O(n^2)$ messages per broadcast

gossip broadcast protocols

if n is large, a process just forward a message to **randomly chosen** (=f) other processes, when it for the first time receives a message;



in this case f = 3 - P8, P9, P11 respectively rebroadcast to 3 other processes

with high probability eventually reaches all processes,

works really well for large peer-to-peer networks, e.g. blockchains; not the best for tight, small networks