03_03_Message Ordering

1. Introduction

For a message m, the set of destinations is denoted by $Dest(m)$ •

- **Point-to-point**: |Dest(m)| = 1
- **Multicast**: |Dest(m)| > 1
- **Broadcast:** |Dest(m)| = #processes

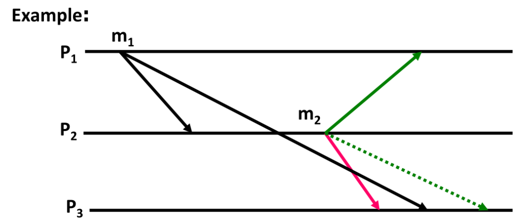
The event of multicasting a message m is denoted by **m(m)**

The event of **delivering** a message m to process Pi is denoted by $d_i(m)$

1.2 Causual Order

Message order is causal when for every two messages m_1 and m_2:

- if m(m1) $\rightarrow$ m(m2),
- then di(m1) $\rightarrow$ di(m2) for **all i** in **both Dest(m_1) and Dest(m_2)**



The contents of **m_2** can depend on the contents of **m_1**, so the delivery of **m_2** in **P_3** has to be postponed until after **m_1** has been delivered

1.2. Total Order

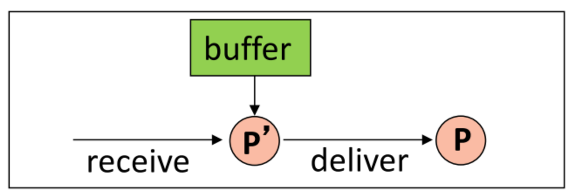
Message order is total when all processes receive all messages in the same order (if broadcast)

1.3. Relations

- Total order **does not imply** causal order
- Causal order **does not imply** total order

1.4. Basic Idea of Message Ordering Algorithm

For message ordering, assume an **additional process** P' for every process P in the system:



- when a message arrives, P' checks whether the message can be delivered to P according to the required order
- if not, P stores the message in a buffer, and re-checks when messages that arrive later have been delivered
- So **reception and delivery** of a message are **separate**

2. Algorithms for Causal Order

Assumption

FIFO Channels

2.1. Causal ordering for broadcast messages (Birman-Schiper-Stephenson algorithm)

Main Idea

- every process **numbers** its own broadcast messages

- a receiving process **checks** whether it is the broadcast message it expects from the sending process
- processes **transfer knowledge** about broadcast messages they have received in the timestamps of the messages they send
- a receiving process **checks** whether it has not missed messages

Process

- A message m is accompanied by the value V_m of the local vector clock when it is sent
- The condition for **delivery** of a message m in P_i from P_j is

Understanding

The delivery condition means:

- the message is the next one expected from P_j (equality in $D_j(m)$)
- with respect to all other processes, P_i is **at least as up to date as** P_j was when it sent the message

Implementation

I. Broadcasting a message

```
V := V + e_i              /* first increment local clock */
broadcast(m,V)
```

II. Receiving a message from P_j

```
upon receipt of (m,V_m) do
    if  D_j(m)  then
        deliver(m)
        while ( |{(m,k,V_m) in B | D_k(m)}| > 0 ) do
            deliver such a message m
    else add (m,j,V_m) to B
```

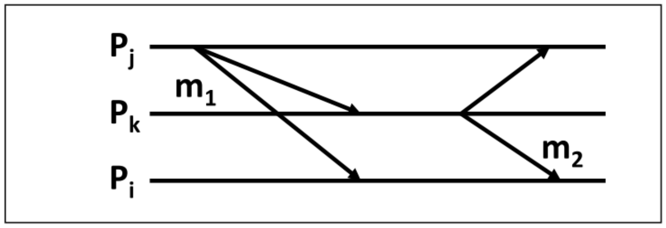
check for other messages that can be delivered

III. Delivering a message from P_j function deliver(m)

```
deliver(m) to P_i
V := V+e_j               /* only modification in component j */
remove(m,V_m) from B     /* in the other components the local vector clock */
                         /* is at least equal to the message clocck anyway */
```

Correctness

- Correctness **of the Berman-Schiper-Stephenson algorithm**:
  - for broadcasts sent by the same process, trivial
  - now suppose
    - $\mathbf{m_1}$ sent by $\mathbf{P_j}$
    - $\mathbf{m_2}$ sent by $\mathbf{P_k}$
    - both received by $\mathbf{P_i}$
    - $m(\mathbf{m_1}) \longrightarrow m(\mathbf{m_2})$



  - so $\mathbf{V(m_2)[j] \geq V(m_1)[j]}$ **(**)**
  - condition for delivery of $\mathbf{m_2}$ in $\mathbf{P_i}$: $\mathbf{V_i[j]} \overset{(*)}{\geq} \mathbf{V(m_2)[j]}$ $\mathbf{(} \overset{(**)}{\geq} \mathbf{V(m_1)[j]\ )}$
  - but $\mathbf{V_i[j]}$ is only ever modified (incremented by 1) by receiving broadcasts from $\mathbf{P_j}$, so $\mathbf{P_i}$ must first receive $\mathbf{m_1}$

2.2. Causal ordering for point-to-point messages (Schiper-Eggli-Sandoz algorithm)

Prepare (Structure)

- Use **vector clocks** (in the ordinary way), all initialized to all **zeroes**
- Every process maintains a **local buffer S** of ordered pairs each consisting of a **process id** and a **timestamp**
- A **message** sent by P_i is accompanied by the **complete current contents of the local buffer** S_i

Main Process

- Condition $D_i(m)$ for delivery of message m with accompanying buffer S_m in P_i:
  - there does not exist (i, V) in S_m
  - or there does exist (i, V) in S_m and $V \leq V_i$
- When a message is delivered in P_i, the **knowledge carried by the message** and the **knowledge available in** P_i about all other processes **are merged**
- Time to update local buffer:
  - when sending
  - when dellivering

Understanding

- Meaning of this pair (P_j, V_j) in P_i:
  - the most recent knowledge in P_i about what P_j should know

- so can be used to tell P_j what it should know
- V_j is at least as large as the timestamp of the last message from P_i to P_j

- The reason message accompanied by buffer: indicating to the receiving process what P_i thinks others should know
- $D_i(m)$
  - either m carries no knowledge at all about what P_i should have received
  - or m does carry such information, but P_i is sufficiently up to date

Implementation

I. Sending a message to $\mathbf{P_j}$:

send(m,S,V) to $\mathbf{P_j}$

insert(j,V) into **S** /* delete any old element for $\mathbf{P_j}$ */

II. Receiving a message

same as for previous algorithm

III. Delivering a message in $\mathbf{P_i}$:

```
deliver(m) to Pi
for all ( (j,V') in Sm ) do
   if ( there exists (j,V") in S ) then
      remove (j,V") from S
      V" := max(V',V")
      insert(j,V") into S
   else insert(j,V') into S
```

merge local buffer with buffer in message

3. Algorithms for Total Ordering

3.1. Simple Solution

have a **special process** P_0 **(a sequencer)**

- when a process wants to do a broadcast, it sends the message to P_0
- P_0 numbers all messages sequentially, and then broadcasts them to all processes
- P_0 keeps a history of messages so that if a process misses a message, it can ask for a resend

3.2. Total ordering for broadcast messages

Main Idea

- use **scalar clocks** (with process ids as **tie breakers**)
- all messages carry a **timestamp**

(ts1,pid1) < (ts2,pid2)
if ts1<ts2 **or**
if ts1=ts2 **and** pid1<pid2

- every process maintains an ordered **message queue**
- **all** processes **acknowledge all** messages to **all** processes (include sender and itselft)

Delivery Condition

A message can be delivered in a process when:

1. it is at **the head of** the local message queue (it is **the oldest message** the process knows about)
2. the process has **received an ack** for that message **from every process** (so no older message will arrive)

Correctness

because of acks and FIFO, all previous messages are forced out to the process