

Messaging Order

Paulo Sérgio Almeida



Universidade do Minho

System Model

- ▶ We may want to assume as little as possible from the system
- ▶ Assumptions for an asynchronous distributed system
 - ▶ No global time, nor knowledge of relative speeds
 - ▶ Messages may take unbounded time to arrive
 - ▶ Messages may be lost, duplicated, reordered
- ▶ We want to write algorithms relying on stronger assumptions

Solution: delegate to an underlying messaging algorithm which ensures some guarantees

Receiving vs delivering messages

- ▶ Suppose high-level algorithm A , resorting to ...
- ▶ ... messaging algorithm M
- ▶ Algorithm M encapsulates messaging from A
- ▶ When a message m arrives at a node n we say n received m
- ▶ Algorithm M may decide it is too soon to handle m to A
- ▶ It may “quarantine” m , i.e., buffer m for some time
- ▶ When appropriate, M hands m to A
- ▶ We say then that m was delivered (to A).
- ▶ Possible events visible to A are then $\text{send}(m)$ and $\text{deliver}(m)$

Some guarantees

- ▶ Reliability; many variations; examples
 - ▶ at-most-once
 - ▶ at-least-once
 - ▶ exactly-once
 - ▶ ...
- ▶ Order
 - ▶ FIFO
 - ▶ causal
 - ▶ total
- ▶ Here we are interested in order guarantees (regardless of reliability)

Some approaches to ensuring delivery order

- ▶ Send set containing also previous messages
 - ▶ wastes bandwidth; a bit naive
 - ▶ needs protocol to GC messages to avoid growth
- ▶ Delay sending
 - ▶ waits for ack before sending next message
 - ▶ causes delays; prevents pipelining
 - ▶ not spatially scalable
- ▶ Delay delivery, buffering at receiver
 - ▶ best general purpose approach
 - ▶ allows pipelining and ack grouping
 - ▶ allows spatial scalability (if possible)
- ▶ Exploit network topology
 - ▶ very interesting if applicable
 - ▶ allows scalable designs
 - ▶ example: FIFO + spanning tree \Rightarrow Causal

FIFO order

- ▶ For any messages a and b , from sender i , to receiver j :

$$\text{send}_i(a) \rightarrow \text{send}_i(b) \Rightarrow \text{deliver}_j(a) \rightarrow \text{deliver}_j(b)$$

- ▶ Simple, cheap, spatially scalable
- ▶ Useful basic guarantee, usually the minimum to aim for
- ▶ Trivially implemented
 - ▶ tag each message with a sequence number
 - ▶ receiver buffers messages out of order
 - ▶ delivers when next suitable one is received
 - ▶ receivers track each sender separately
 - ▶ sender keeps a sequence number per receiver ...
 - ▶ ... unless messages are broadcast, where one is enough

Causal order

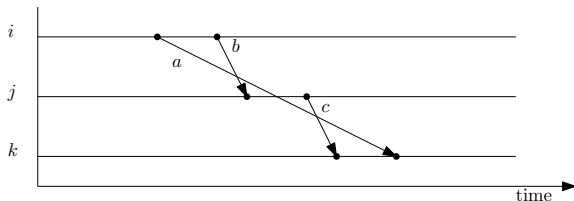
- ▶ For any messages a and b , from senders i and j , to receiver k :

$$\text{send}_i(a) \rightarrow \text{send}_j(b) \Rightarrow \text{deliver}_k(a) \rightarrow \text{deliver}_k(b)$$

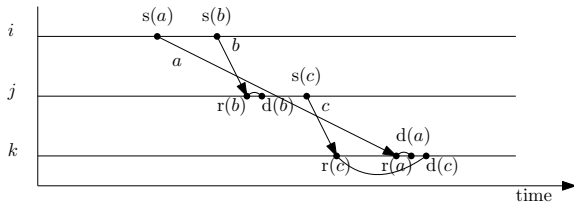
- ▶ Includes the case $i = j$; FIFO $<$ causal
- ▶ Useful guarantee to obtain causal consistency
- ▶ More complex to implement than FIFO
- ▶ More difficult to scale with number of nodes than FIFO
- ▶ Strongest still spatially scalable delivery order
 - ▶ far away nodes do not slow down nearby interactions

Delaying delivery to achieve causal order

- Violation of causal order, if message c is delivered to k before a



- Ensuring causal order by delaying delivery of c after a
(s – send, r – receive, d – deliver)



Detecting causality violation

- ▶ First ingredient to ensure causal delivery; the easy part
- ▶ Lamport clocks are not enough; provide only “if”, not “iff”

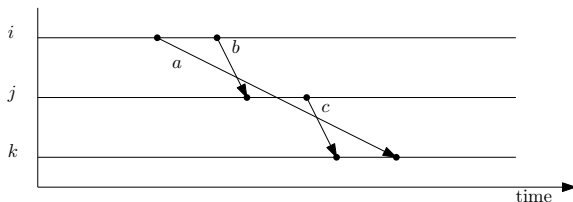
if causal order has been violated, a message m with clock L_m must have arrived at node i when $L_i > L_m$

- ▶ Vector clocks provide “iff”, allowing checking for violations

causal order has been violated if and only if a message m with clock V_m arrives at node i when $V_i > V_m$

Knowing if messages can still arrive

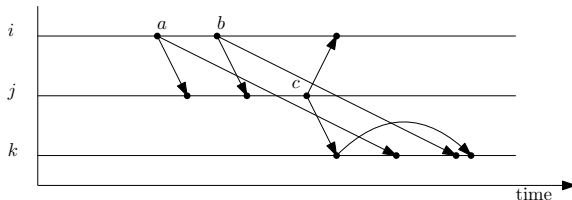
- ▶ Second ingredient: knowing if some causally in the past message can still arrive
- ▶ Complex and expensive in the general case



- ▶ How does node k know when receiving c that node i has previously sent a but it has not yet arrived?
- ▶ Less expensive for a common useful case: causal broadcast

Causal broadcast

- ▶ The common scenario: each “send” (cbcast) is to all nodes
- ▶ Useful for symmetric data replication



- ▶ c carries info that it depends on first two messages from i
- ▶ Because each message is for all nodes, when k receives c :
 - ▶ it knows it has not yet delivered two messages from i
 - ▶ that they must be delivered before c
 - ▶ so, it buffers c to be delivered after those messages

Causal broadcast algorithm

- ▶ Each node keeps a vector of integers (or map from node ids)
 - ▶ like a vector clock, but updated only on cbcast and deliver
- ▶ On cbcast(m), increment self entry:

$$V[i] := V[i] + 1$$

and send vector with message:

$$\text{for } j \in I \setminus \{i\} : \text{send}(j, m, V)$$

- ▶ On receive(j, m, V_m), i.e., m tagged with V_m , buffer m until:

$$V[j] + 1 = V_m[j] \wedge \forall k \neq j \cdot V_m[k] \leq V[k]$$

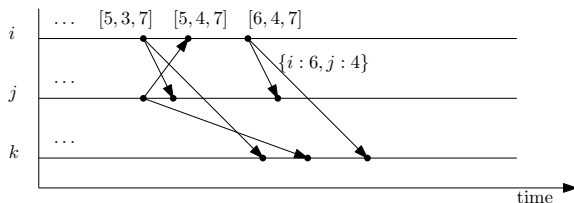
then increment j 's entry:

$$V[j] := V[j] + 1$$

and deliver m (and retest other messages in the buffer)

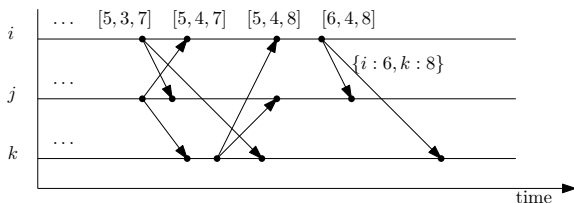
Causal broadcast – improvements (1)

- ▶ Sending the full vector is not really needed
- ▶ For each sender, $n + 1$ message delivered after message n
- ▶ We need only send entries that changed since previous cbcast
- ▶ For many nodes it can save bandwidth



Causal broadcast – improvements (2)

- ▶ A vector describes transitive dependencies
- ▶ My dependencies already wait for their dependencies
- ▶ We need only send direct dependencies (direct predecessors)



Total order

- ▶ For any two messages a and b delivered by two nodes i and j :

$$\text{deliver}_i(a) \rightarrow \text{deliver}_i(b) \Rightarrow \text{deliver}_j(a) \rightarrow \text{deliver}_j(b)$$

- ▶ Does not necessarily respect FIFO or causal;
- ▶ But would be silly not to
 - ▶ extra cost of getting FIFO pales in comparison
 - ▶ would violate basic guarantees, such as monotonic writes
- ▶ So, normally we want to respect FIFO and in such case

$$\text{FIFO} < \text{causal} < \text{total}$$

Total order broadcast

- ▶ Total order broadcast useful where a set of replicas needs to apply requests in the same order; a replicated state machine
- ▶ Can be implemented by:
 - ▶ Lamport clocks based algorithm; generalisation of lock
 - ▶ Consensus based, for fault tolerance
- ▶ In most contexts this term also implies reliability
- ▶ Undesirable spatial scalability properties:
 - ▶ minimum delivery time proportional to physical span
 - ▶ does not tolerate network partitions
- ▶ Undesirable for large scale systems aiming for responsiveness