

Distributed System Logical Time

Prof. Roberto Baldoni
Ing. Silvia Bonomi



Logical clock

- Physical clock synchronization algorithms try to coordinate distributed clocks to reach a common value
- Physical clock synchronization algorithms are based on estimates of transmission delay but in several system it can be hard to find a good estimation.
- In several applications is important not when things happened but in what order they happened (integer counter in centralized systems)
- However in a Distributed System, each distinct system have its own “logical clock” and you can run into problems if one “clock” gets ahead of others
- **Reliable way of ordering events is required!**
 - We need a rule to synchronize logical clocks



- **Notes:**

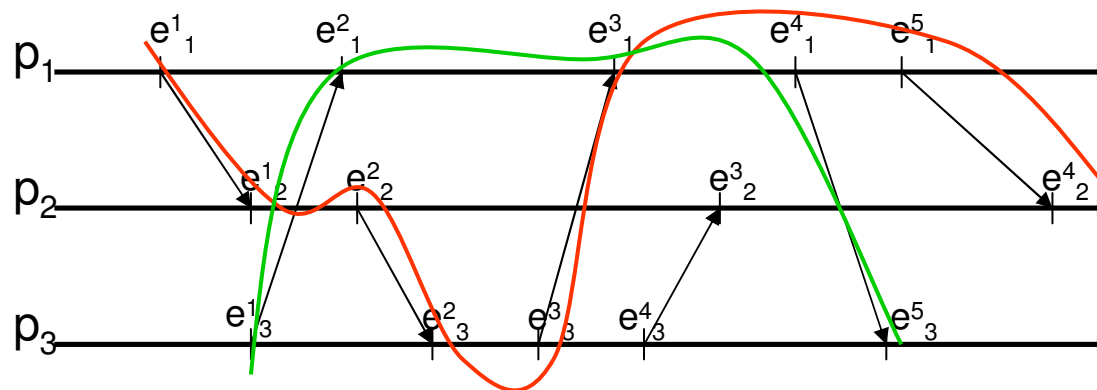
- Two events occurred in the same process p_i happen in the same order as they are observed by p_i
 - When p_i sends a message to p_j the *send* event happens before the *receive* event
- Lamport introduces the *happened-before* relation that captures the causal dependencies between events (*causal order relation*)
 - We note with \rightarrow_i the ordering relation between events in a process p_i
 - We note with \rightarrow the happened-before between any pair of events



Happened-Before Relation: Definition

- Two events e and e' are related by happened-before relation ($e \rightarrow e'$) if:
 - $\exists p_i \mid e \rightarrow_i e'$
 - \forall message m $\text{send}(m) \rightarrow \text{receive}(m)$
 - $\text{send}(m)$ is the event of sending a message m in a process
 - $\text{receive}(m)$ is the event of receipt of the same message m by another process
 - $\exists e, e', e'' \mid (e \rightarrow e'') \wedge (e'' \rightarrow e')$
- **Notes:**
 - There could be a pair of events $\langle e_1, e_2 \rangle$ not related by the happened-before relation: they are *concurrent* ($e_1 \parallel e_2$) and we cannot say whether one event happened-before
 - For any two events e_1 and e_2 in a distributed system, either $e_1 \rightarrow e_2$, $e_2 \rightarrow e_1$ or $e_1 \parallel e_2$

happened-before relation: example



e^j_i is j -th event of process p_i

Note:

e^1_3 and e^1_2 are concurrent



Logical Clock

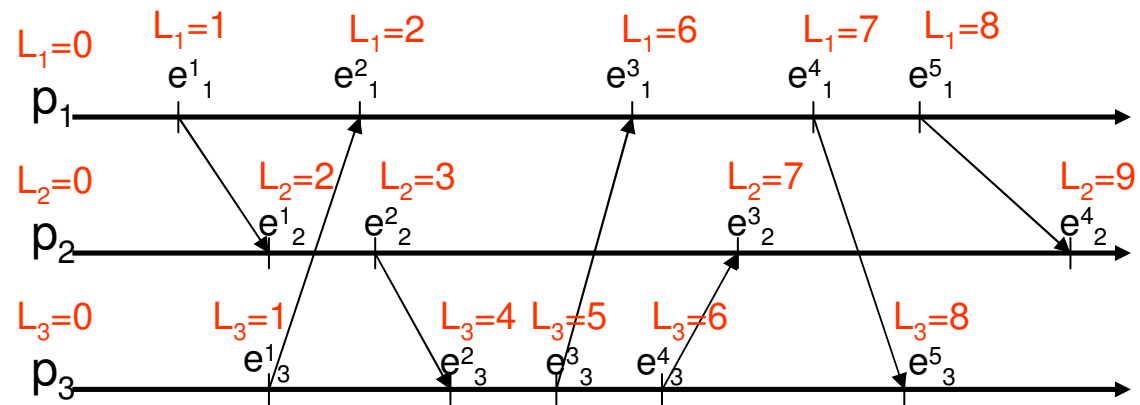
- The Logical Clock, introduced by Lamport, is a software counting register *monotonically* increasing its value
 - Logical clock is not related to physical clock
- Each process p_i employes its logical clock L_i to apply a *timestamp* to events
- $L_i(e)$ is the “logical” timestamp assigned, using the logical clock, by a process p_i to events e .
- **Property:**
 - If $e \rightarrow e'$ then $L(e) < L(e')$
- **Observation:**
 - The ordering relation obtained through logical timestamps is only a partial ordering. Consequently timestamps could not be sufficient to relate two events



Scalar Logical Clock: an implementation

- Each process p_i initializes its logical clock $L_i=0$ ($\forall i = 1 \dots N$);
- for every internal event
 - $L_i = L_i + 1$
- When p_i sends a message m
 - $L_i = L_i + 1$
 - timestamps m with $t=L_i$
 - $send(m)$
- When a message m arrives at p_i with timestamp t
 - $L_i = \max(t, L_i)$
 - $L_i = L_i + 1$
 - Produces an event $receive(m)$

Scalar Clock: example



- e^j_i is j -th event of process p_i
- L_i is the logical clock logico of p_i

Property:

If $e \rightarrow e'$ then $L(e) < L(e')$



Limits of Scalar Logical Clock

- Scalar logical clock can guarantee the following property
 - IF $e \rightarrow e'$ then $L(e) < L(e')$
- But it is not possible to guarantee
 - IF $L(e) < L(e')$ then $e \rightarrow e'$
- **Consequently:**
 - It is not possible to determine, analyzing only scalar clocks, if two events are concurrent or correlated by the happened-before relation.
- Mattern [1989] and Fidge [1991] proposed an improved version of logical clock where events are timestamped with local logical clock and node identifier
 - ***Vector Clock***



Vector Clock : definition

- Vector Clock for a set of N processes is composed by an array of N integer counters
- Each process p_i maintains a Vector Clock V_i and timestamps events by mean of its Vector Clock
- Similarly to scalar clock, Vector Clock is attached to message m (in this case we attach an array of integer)
- Vector Clock allows nodes to order events in happens-before order based on timestamps
 - Scalar clocks: $e \rightarrow e'$ implies $L(e) < L(e')$
 - Vector clocks: $e \rightarrow e'$ **iff** $L(e) < L(e')$



Vector Clock : an implementation

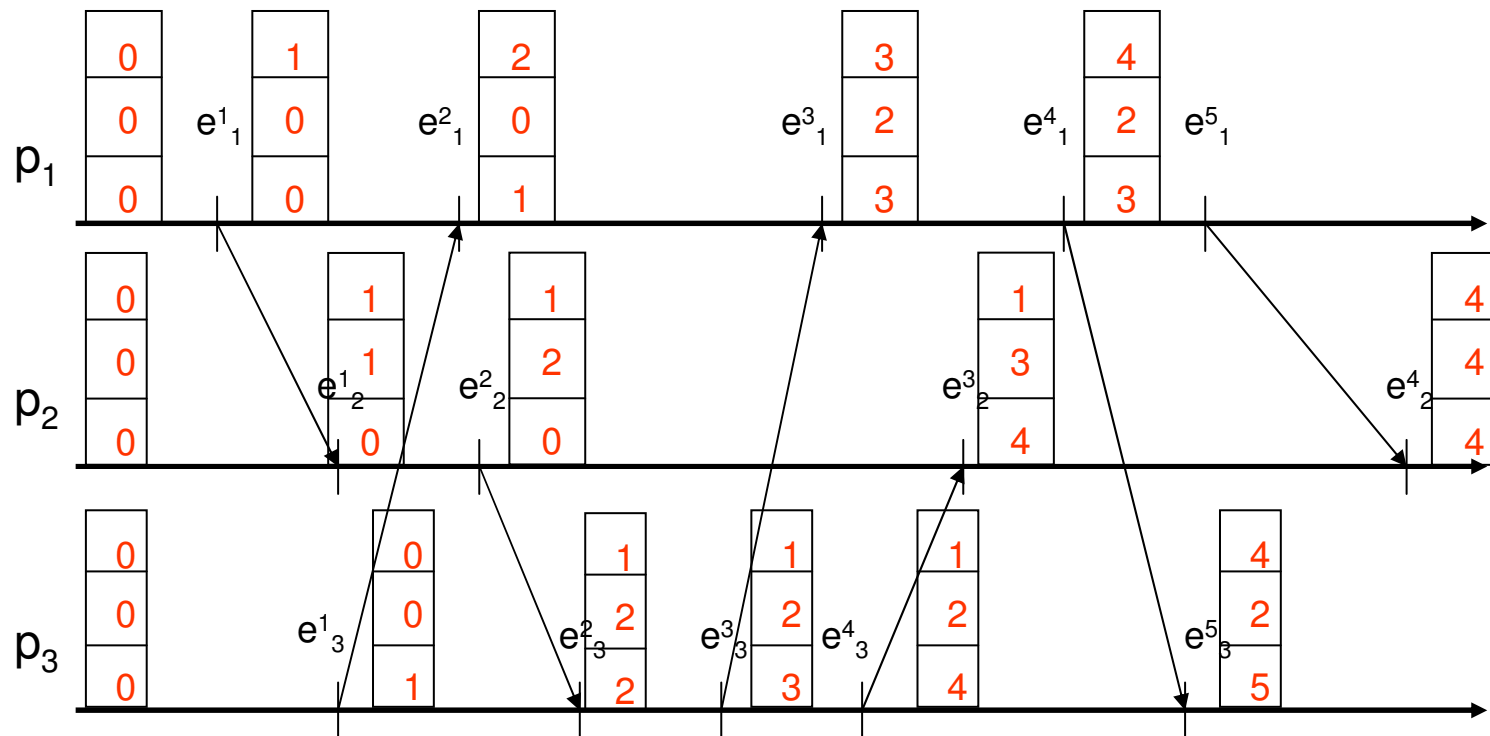
- Each process p_i initializes its Vector Clock V_i
 - $V_i[j]=0 \quad \forall j = 1 \dots N$
- p_i increases $V_i[i]$ of 1 when it generates an internal event
 - $V_i[i]= V_i[i] +1$
- When p_i sends a message m
 - $V_i[i]= V_i[i] +1$
 - Send m with a timestamps $t=V_i$
- When p_i receives a message m containing timestamp t
 - Updates its logical clock $V_i[j] = \max(t[j], V_i[j]) \quad \forall j = 1 \dots N$
 - $V_i[i]= V_i[i] +1$



Vector Clock: properties

- A Vector Clock V_i
 - $V_i[i]$ represents the number of events produced by p_i
 - $V_i[j]$ with $i \neq j$ represents the number of events generated by p_j and known by p_i
- $V = V'$ if and only if
 - $V[j] = V'[j] \quad \forall j = 1 \dots N$
- $V \leq V'$ if and only if
 - $V[j] \leq V'[j] \quad \forall j = 1 \dots N$
- $V < V'$ therefore the event associated to V happened before the event associated to V' if and only if
 - $V \leq V' \wedge V \neq V'$
 - $\forall i = 1 \dots N \quad V'[i] \geq V[i]$
 - $\exists i \in \{1 \dots N\} \mid V'[i] > V[i]$

Vector Clock: an example



A comparison of Vector Clocks

1
0
0

V

1
1
0

V'

$V(e) < V'(e')$ then $e \rightarrow e'$

1
0
0

V

0
0
1

V'

$V(e) \neq V'(e')$ then $e \parallel e'$

Differently from Scalar Clock, Vector Clocks allow to determine if two events are concurrent or related by an happened-before relation



Causal Ordering among broadcasted events

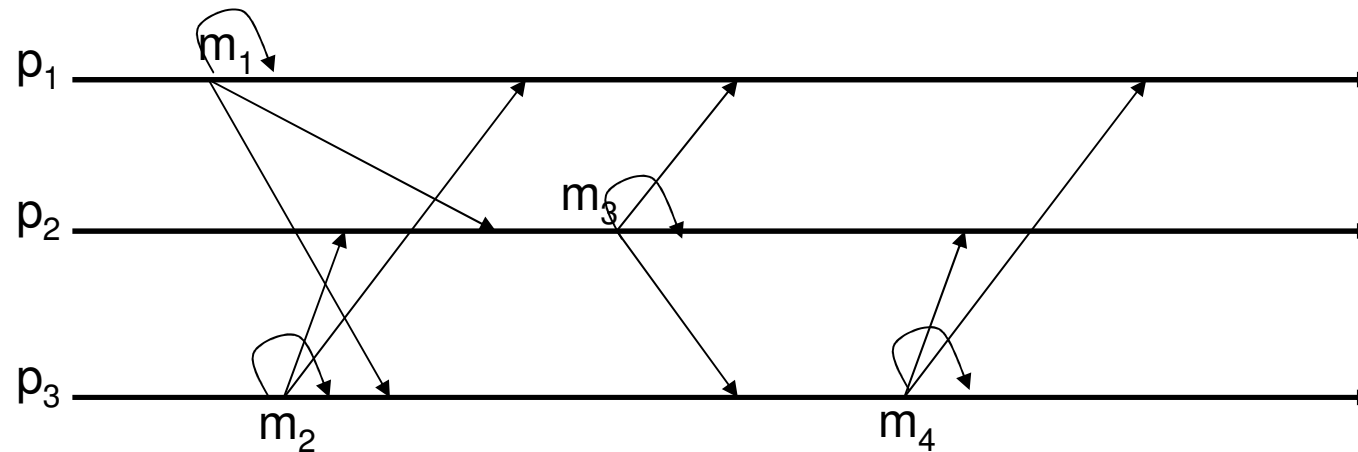
- In a broadcast event a process p_i sends a message m to each other process in the system
- An event $e = \text{broadcast}(m)$ causally comes ahead another event $e' = \text{broadcast}(m')$ if at least one of following condition is true
 - e and e' have been produced by the same process and $\text{broadcast}(m)$ happens before $\text{broadcast}(m')$
 - e and e' have been produced by different processes but e' was produced only after the deliver of m
 - $\exists m'' \mid \text{broadcast}(m) \rightarrow \text{broadcast}(m'') \wedge \text{broadcast}(m'') \rightarrow \text{broadcast}(m')$



Causal Broadcast: Specification

- Causal broadcast was introduced by Birman and Joseph in order to reduce the asymmetry of communication channels as it was perceived by processes.
- **Specification:**
 - (Safety) Let two broadcast messages m and m' such that $broadcast(m) \rightarrow broadcast(m')$ then each process have to deliver m before m'
 - (Liveness) Eventually each message will be delivered
- Therefore, two broadcast message m and m' for which $broadcast(m) \parallel broadcast(m')$ then m and m' can be delivered in different order by different sets of processes

Causal Broadcast: an example



- In this scenario
 - $m_1 \rightarrow m_3 \Rightarrow$ every process has to deliver m_1 before m_3
 - $m_1 \parallel m_2 \Rightarrow m_1$ and m_2 can be delivered in any order
 - $m_1 \rightarrow m_4 \Rightarrow$ every process has to deliver m_1 before m_4



Model of the computation

- Each process step takes a finite time to occur
- Message transfer delays are unpredictable but finite
- Communication channels are reliable
- Computation is failure free



Fifo Broadcast: pseudo-code

- Each process p_i implements the following rule to manage the casual broadcast

- **Procedure broadcast (m)**
 - $m.VC = V_i [i]$ //message timestamp
 - for all $j \in \{1 \dots N\}$
 - Send(m) to p_j // message broadcast
 - and for
 - $V_i [i] = V_i [i] + 1$ // updating local clock

- **Upon receive m from p_j**
 - delay the delivery until $m.VC \leq V_i [j]$
 - if $i \neq j$
 - then $V_i [j] = V_i [j] + 1$ //updating local clock
 - deliver m to the upper layer //deliver event



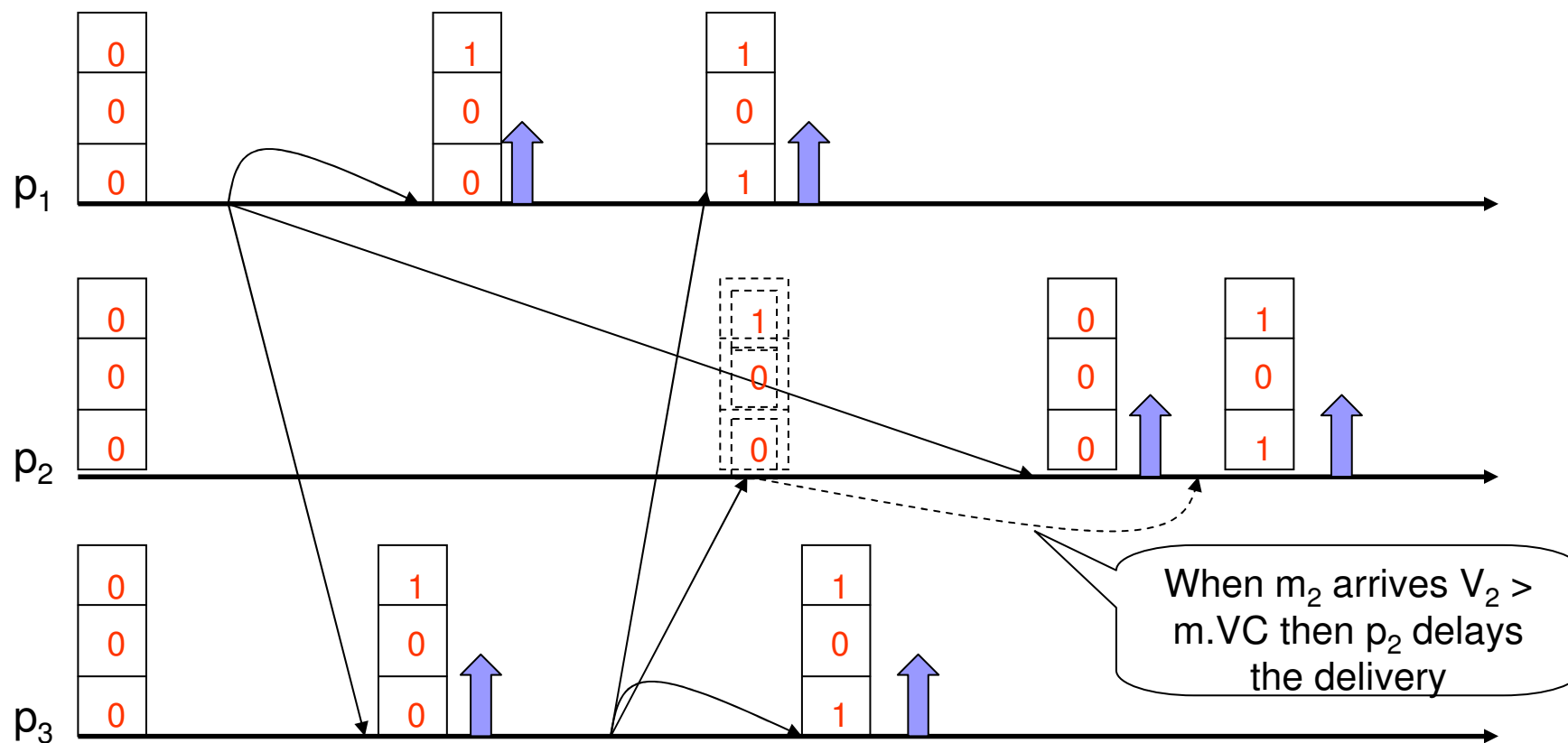
Causal Broadcast: pseudo-code

- Each process p_i implements the following rule to manage the casual broadcast

- **Procedure broadcast (m)**
 - $m.VC = V_i$ // message timestamp
 - for all $j \in \{1 \dots N\}$
 - Send(m) to p_j // message broadcast
 - and for
 - $V_i[i] = V_i[i] + 1$ //updating local clock

- **Upon receive m from p_j**
 - delay the delivery until $\forall k \in \{1 \dots N\} m.VC[k] \leq V_i[k]$
 - if $i \neq j$
 - then $V_i[j] = V_i[j] + 1$ //updating local clock
 - deliver m to the upper layer //deliver event

Causal Broadcast: an example





Causal Broadcast: Safety

■ Property:

- Let two broadcast messages m and m' such that $broadcast(m) \rightarrow broadcast(m')$ then each process have to deliver m before m'

■ Observation:

- if m is the k -th message sent by p_i then $m.Vc[i] = k-1$

- Safety property can be proved by induction using the causal ordering relation among broadcast messages

■ Definition:

- Let two broadcast events b and b' with $b \rightarrow b'$. These events have a causal distance k if \exists a sequence of k broadcast events $b_1 \dots b_k$ such that
 - $\forall i \in \{1 \dots k\} b_i \rightarrow b_{i+1} \wedge (\neg \exists) m^* | b_i \rightarrow m^* \rightarrow b_{i+1}$
 - $b \rightarrow b_1$
 - $b_k \rightarrow b'$



Proof – basic case ($K=0$)

- For two message m, m' such that
 - $\text{broadcast}(m) \rightarrow \text{broadcast}(m')$
 - There does not exist $\text{broadcast}(m'')$ such that $\text{broadcast}(m) \rightarrow \text{broadcast}(m'') \wedge \text{broadcast}(m'') \rightarrow \text{broadcast}(m')$.
- We can have two distinct cases
 - m and m' were produced by the same process
 - m and m' were produced by distinct processes



Case 1 – broadcast produced by the same process

1. p_j is the receiver
2. For line 3 in broadcast procedure
 1. $m'.VC[i] := m.VC[i] + 1$.
if m is the h -th message sent by p_i , $m.VC[i] = h - 1$ and $m'.VC[i] = h$.
3. A process p_j that receives m' verifies the following delivery condition:
 1. $\forall x \in \{1, \dots, n\} m'.VC[x] \leq V_j[x]$
4. This implies $V_i[x]$ is equals to h if and only if the h -th message sent by p_x was delivered by p_i . (line 3 receive thread).
5. Consequently from 2,3,4, m' can be delivered only after the deliver of m .



Case 1 – broadcast produced by distinct processes

- m and m' were sent by distinct processes, respectively p_i e p_j . P_k is the receiver.
- $\text{broadcast}(m) \rightarrow \text{broadcast}(m')$, m' was broadcasted by p_j after the deliver of m .
Without loss of generality $m.VC[i]=h-1$
 - For line 3 of reception thread and for the sending procedure at p_j we have $m'.VC[i]=h$.
- The receiver process p_k respects the following delivery condition:
 - $\forall x \in \{1, \dots, n\} m'.VC[x] \leq V_k[x]$
- This means $m'.VC[i] \leq V_k[i]$, that is $V_k[i] \geq h$
- $V_k[i]$ is equals to h if and only if the h -th message sent by p_i has been delivered by p_k . (line 3 of reception thread thread).
- For 2,3,4, p_k can deliver m' only after the deliver of m



Proof – Inductive step($k > 0$)

- \exists a sequence of k broadcast events $b_1, b_2 \dots b_k$ such that
$$b \rightarrow b_1 \rightarrow b_2 \rightarrow \dots \rightarrow b_k \rightarrow b'$$
- Inductive hypothesis : m has been delivered before m_k
- We have to prove that m_k has been delivered before m' .
 - It follows from the basic case.
- m has been delivered before m' .



Causal Broadcast: Liveness

- **Property:**
 - Eventually each message will be delivered
- Liveness is guaranteed by the following pair of assumptions:
 - The number of broadcast events relative to messages that come before a certain event is finite
 - Channels are reliable

