Distributed systems

Vivien Quéma (vivien.quema@imag.fr)
Scribe: Tom Cornebize

Contents

Chapter 1

Introduction

Why do we use distributed systems?

- Reliability (the probability that the service is unavailable should be lower if there are more machines to offer it).
- Security (if data is spread accross machines, it should be harder to hack them and get all the data).
- Performances (more processors, more memory, so we should have better performances).

Distributed systems "invented" by L. Lamport (Turing award in 2014).

1.1 Methodology

- **Specification** an API, requirements, ...
- **Assumptions** Assumptions are everywhere in distributed systems. Fischer, Lynch, Paterson (1985): *Impossibility of Distributed Consensus with One Faulty Process*. So, unless some assumptions, we cannot do much intersting things in distributed systems.
  - Number of machines.
  - Relibality of the network: messages loss, reordering.
  - Performances of the network: bounded time to communicate or not.
  - Failure of the machines: crash failures (some machine(s) completely stop working), omission failures (some machine(s) cannot send any messages for some time), crash recovery (never crashes, crash and recover a finite number of times, or an infinite number), arbitrary failures (everything is possible, bugs, malicious behaviours). Currently, most algorithms assume there are no arbitrary failures, since it is very difficult. The current research work is to remove this assumption. The question is also, how many machine do we assume to fail, for each type of failure? Is there a bound on the number of failures?
  - Synchronized clocks (is there a bound on clock drift).
  - Properties of the (if any) failure detector: type of detected failure, probability to detect a failure, probability to falsly suspect a mahcine. In this course, we will either consider $\mathcal{P}$, the perfect failure detector, or assume there is no failure detector.
- **Algorithm**
- **Proof**
- **Theoretical performance assessment**

1.1.1 Perfect failure detector

Let us define $\mathcal{P}$, the perfect failure detector.

Interface Out: $crash(p_i)$, the failure detector notifies the crash of process p_i.

Properties

1. Strong completeness: if a process p_i crashes, then $\mathcal{P}$ will eventually detect it (no false negative).
2. Strong accuracy: if $\mathcal{P}$ notifies the crash of a process p_i, then p_i has actually crashed (no false positive).

Assumptions A first algorithm is to assume that all processes are correct (never crash) and that the network is reliable. Then, $\mathcal{P}$ does not have to do anything.

1. We know all the processes of the system.
2. The network is reliable.
3. Known bound on processing time.
4. Known bound on communication time.
5. No arbitrary failures, neither omission failures or crash recovery.
6. Same clock speed (to be able to measure time consistently).

Algorithm Idea of algorithm:

---

for p_i ***in*** *processes* **do**
    send ping to p_i every X seconds
    if *no response* **then**
        $crash(p_i)$

Correction (it has to run on every process):

1.1.2 Which algorithms to study?

- Consensus (e.g. Paxos).
- Leader election. With a consensus algorithm, we can elect a leader. With a leader, we can reach a consensus. But, we do not need a leader to reach a consensus.
- Failure detectors.
- Synchronization (e.g. mutual exclusion, rendez-vous).
- Point-to-point communication.
- Broadcast and multicast.
- Distributed shared memory (also called a register).

We will not have time to study everything.

π is the set of processes
$alive = \pi$
$still\_alive = \emptyset$
in parallel

- **for** *every X second* **do**
  - **for** $q \in alive$ **do**
    - send ping to q
- **for** *every received ping from q* **do**
  - $still\_alive = still\_alive \cup \{q\}$
- **for** *every Y second* **do**
  - **for** $q \in alive \setminus still\_alive$ **do**
    - notify $crash(q)$
    - $alive = alive \setminus \{q\}$
  - $still\_alive = \emptyset$

Chapter 2

Broadcast

The goal is to send a same message to a set of machines.
What about faulty machines? We consider crash failures.

- Faulty sender. If the sender crash during the broadcast, should all the other machines receive it?
- Faulty receiver. What happens if a sender crashes? Does it impact other senders?

2.1 Best effort broadcast

Interface

in: $broadcast(m)$, broadcast m to all processes.

out: $deliver(m, q)$, used to deliver (to the application) a message m broadcast by q.

Specifications

Validity: if a correct process broadcasts a message m, then all correct processes deliver the message m.

Integrity: a process delivers a message m only once and if m has been broadcast by a process.

Assumptions

- We know all the processes.
- Crash model.
- Reliable network, no bound on communication time.
- No failure detection.
- No assumptions on clocks.

Algorithm We suppose that we have two functions, $rc\_send(m)$ and $rc\_receive(m)$ to send and receive messages through the reliable channel.

- for all processes p, $rc\_send(m)$ to p
- upon $rc\_deliver(m)$ from q, $deliver(m, q)$

This protocol is very usefull, it is widely used. But it has important limitations.

What if a receiver crashes? It may be in an inconsistent state when it is restarted. Even worse, what if a sender crashes during the multicast? Some receivers may have received it but not all.

In consequence:

B1 Two correct machines can deliver different sets of messages (in case one sender is faulty).

B2 A faulty machine can deliver messages that will not be delivered by correct machines (e.g. a faulty sender may deliver its own message and crashing immediately after, without having time to send it).

This is a huge issue in practice. For instance, in distributed databases, this can lead to non-consistent state. The protocol has still important use cases, e.g. live video broadcast.

2.2 Reliable broadcast

We will avoid **B1**.

Interface Same as best-effort broadcast.

Specifications

Validity: if a correct process broadcasts a message m, then this process eventually delivers m.

Agreement: if a correct process delivers a message m, then all correct processes will eventually deliver m.

Integrity: a process delivers a message m only once and if m has been broadcast by a process.

Assumptions We assume we have a best-effort broadcast (with the assumptions this implies).

Algorithm The idea of the algorithm to ensure agreement is that any process which receives a broadcast message and delivers it has to broadcast it (using best-effort broadcast). Thus, we are sure that agreement is fullfiled.

But, how to avoid replication of messages?

A first idea is to have a list of received messages on each process (with the sender ID and a message ID). When we receive a message, we check on this list before re-sending it. However, it requires an infinite memory.

A second idea is to use a sliding windows. Messages are identified by the process ID of the sender and a message ID. Every process has one buffer per other process for the messages it needs to send, and one buffer per other process for received messages. If one process wants to send one of its own messages,

it needs to be in the windows. Each pair of processes then agrees on when to shift the windows. This is similar to TCP protocol.

A drawback of this algorithm is that we send $\mathcal{O}\left(N^2\right)$ messages to do one single broadcast with N processes.

Can we have a faster protocol? Yes, but we need to pay: if we have $\mathcal{P}$, the perfect failure detector, we can decrease largely the number of messages. Now, when the sender of a broadcast crashes, then the processes which have received it broadcast it. When there is no crash, they do nothing. For how long do we need to remember a message? As long as we are not sure that all alive processes have delivered it. To do so, we use again $\mathcal{P}$. From time to time, processes acknowledge to everybody all the messages they have received (reasonable cost since we do not acknowledge every message).

2.3 Uniform reliable broadcast

We will avoid **B2**.

Interface Same as best-effort broadcast.

Specifications

Validity: same as reliable broadcast.

Agreement: if a process delivers a message m, then all correct processes will eventually deliver m.

Integrity: same as reliable broadcast.

Assumptions We assume we have $\mathcal{P}$, the perfect failure detector (with the assumptions this implies).

Algorithm Same idea as reliable broadcast. The initiator does the best effort broadcast. When a process receives a message it re-broadcasts it (best-effort). But now, one process waits to have received a same message from all the alive processes before delivering it (thus we need $\mathcal{P}$).

Remark: as in previous section, we did not write any pseudo-code. It would have been way too messy and too long.

Proof

Validity: let P be a correct process that broadcasts a message m. By validity of best-effort broadcast, all correct processes receive m. Then, each of these processes re-broadcast this message. By validity of best-effort broadcast, P will receive the re-broadcast of all these processes. Eventually, all faulty processes will have crashed and P will have received the re-broadcast of all the correct processes (which will be the only alive at the moment), P will eventually know which are the alive processes (validity of $\mathcal{P}$), so P will deliver the message.

Agreement: let P be a process that delivers a message m. This means that every alive process has sent the re-broadcast. By validity of the best-effort broadcast, every process will eventually receive all these re-broadcast, and thus deliver.

Integrity: (...)

2.4 Total order broadcast

With the previous broadcast algorithms, nodes will deliver the same messages, but not necessarily in the same order. This is required if we want a consistent state in a distributed database (there is a problem if transactions are executed in a different order). Key-word: state machine replication (SMR).

Interface Same as best-effort broadcast.

Specifications

Validity: same as reliable broadcast.

Agreement: same as reliable broadcast, or uniform reliable broadcast (depending on what you want).

Integrity: same as reliable broadcast.

Total Order: if a process delivers m_1 before m_2, then no process can deliver m_2 before m_1.

Remark: there are other ordering properties that can be defined. FIFO order (the messages broadcast by a given process need to be delivered in the broadcasting order), or causal order (used by Facebook, studied later on).

Total order does not imply FIFO order, FIFO order does not imply total order. These are two distinct properties, that can be combined.

We need two algorithms, one for total order broadcast, another for uniform total order broadcast.

Assumptions We assume that we have $\mathcal{P}$, the perfect failure detector, and thus leader election (the leader is the alive process with lowest rank).

Algorithm At any time, one of the processes has a special role, a "sequencer". When one process wants to broadcast, it firstly asks to the sequencer a sequence number. Then, the process broadcasts its message with the sequence number. Now, processes deliver a message only when all previous messages have been delivered. The algorithm has to be adapted for uniformity or not.

The protocol must be carefully designed to ovevrcome any crash. For instance, if one process asks for a sequence number and crashes just after, the message with this sequence number will never be broadcast. But other processes will detect the crash thanks to $\mathcal{P}$ and update their state.

Designing the protocol in every details is tricky, but possible.

Chapter 3

Consensus

The main intuition behind consensus is:

1. Processes can propose values.
2. One value has to be decided (the same for all correct processes).
3. That value must have been proposed.

Consensus is equivalent to total order broadcast.

Assumptions

- Crash recovery.
- Reliable channels.
- Known number of processes.

Remark: we do not assume we have the perfect failure detector. The algorithm has to be safe with any number of faulty processes (i.e. will not take a wrong decision), and live if a majority of processes are correct (i.e. will eventually do a decision).

Algorithm We consider two set of processes, $2f + 1$ acceptors and n proposors.

Acceptors maintain the following state on disk:

- S_N, the last accepted sequence number, initially -1
- A_V the last accepted value, initially $(-1, \perp)$

We consider here unique sequence numbers (note that this not an issue, it is always possible to do this, e.g. based on the rank of the processes).

There are four rules executed by the acceptors.

When one of the proposers wants to reach a consensus, it sends a message $PREPARE(seqno = i)$.

When the acceptors receive this message, they execute rule (1).

When the proposers receive the answer from (1), they execute (2).

Then, acceptors execute (3).

1. If $seqno(PREPARE) \geqslant S_N$ then update S_N on disk and send (OK, A_V), otherwise ignore or send KO.
2. If $(f+1)$ OK are received, then send $ACCEPT(seqno, V)$, where seqno is the one set in the PREPARE message and V is either any value if $(f+1)$ messages with $\perp$ are received, or the value in the A_V having the larger sequence number.
3. If $seqno(ACCEPT) \geqslant S_N$ then update $A_V = (seqno, V)$ on disk and send (OK, A_V), otherwise ignore or send KO.
4. As soon as $f+1$ acceptors accept the same $A_V = (S_N, V)$, then V is decided. Any process learning then can "use" V.

This algorithm is called Paxos, invented by L. Lamport in the 90's.

Nice video about Paxos: `https://www.youtube.com/watch?v=cj9DCYac3dw`.

Paxos can be made more efficient if we allow the perfect failure detector. In this case, we have a distinguished proposer. Then, it is the only one allowed to propose values, so it does not need to send PREPARE messages, it can directly send ACCEPT. If at some point it crashes, then we replace it by some other process.

3.1 Paxos by example

This is a copy of the page `https://angus.nyc/2012/paxos-by-example/`.

It is slightly different than what we saw during the course: here, the proposers include their value in the prepare message (and not only their sequence number).

Distributed consensus algorithms are used to enable a set of computers to agree on a single value, such as the commit or rollback decision typically made using a two- or three-phase commit. It doesn't matter to the algorithm what this value is, as long as only a single value is ever chosen.

In distributed systems this is hard, because messages between machines can be lost or indefinitely delayed, or the machines themselves can fail.

Paxos guarantees that nodes will only ever choose a single value (meaning it guarantees safety), but does not guarantee that a value will be chosen if a majority of nodes are unavailable (progress).

3.1.1 General approach

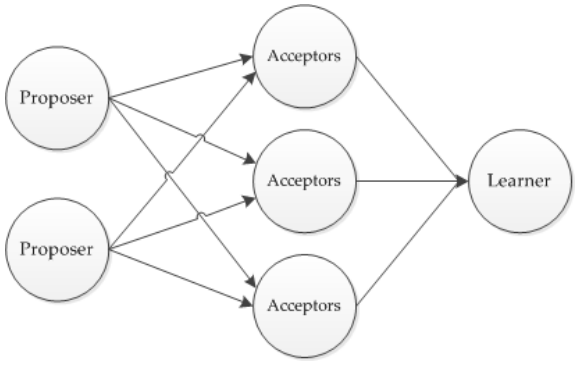

Figure 3.1: Basic Paxos architecture. A number of proposers make proposals to acceptors. When an acceptor accepts a value it sends the result to learner nodes.

A Paxos node can take on any or all of three roles: proposer, acceptor, and learner. A proposer proposes a value that it wants agreement upon. It does this by sending a proposal containing a value to the set of all acceptors, which decide whether to accept the value. Each acceptor chooses a value independently — it may receive multiple proposals, each from a different proposer — and sends its decision to learners, which determine whether any value has been accepted. For a value to be accepted by Paxos, a majority of acceptors must choose the same value. In practice, a single node may take on many or all of these roles, but in the examples in this section each role is run on a separate node, as illustrated above.

3.1.2 Paxos by example

In the standard Paxos algorithm proposers send two types of messages to acceptors: prepare and accept requests. In the first stage of this algorithm a proposer sends a prepare request to each acceptor containing a proposed value, v, and a proposal number, n. Each proposer's proposal number must be a positive, monotonically increasing, unique, natural number, with respect to other proposers' proposal numbers.

In the example illustrated below, there are two proposers, both making prepare requests. The request from proposer A reaches acceptors X and Y before the request from proposer B, but the request from proposer B reaches acceptor Z first.

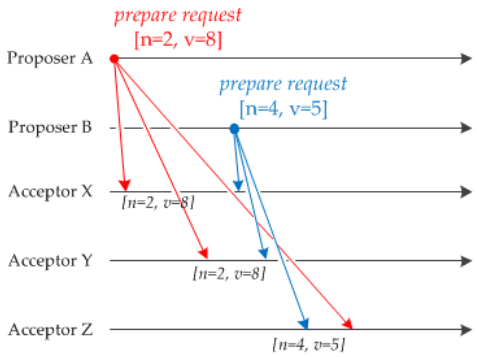

Figure 3.2: Proposers A and B each send prepare requests to every acceptor. In this example proposer A's request reaches acceptors X and Y first, and proposer B's request reaches acceptor Z first.

If the acceptor receiving a prepare request has not seen another proposal, the acceptor responds with a prepare response which promises never to accept another proposal with a lower proposal number. This is illustrated in Figure 3 below, which shows the responses from each acceptor to the first prepare request they receive.

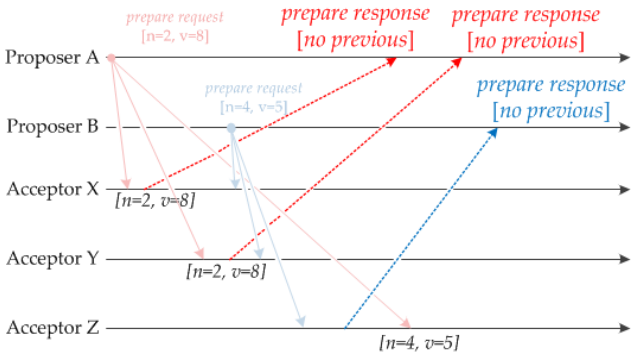

Figure 3.3: Each acceptor responds to the first prepare request message that it receives.

Eventually, acceptor Z receives proposer A's request, and acceptors X and Y receive proposer B's request. If the acceptor has already seen a request with a higher proposal number, the prepare request

is ignored, as is the case with proposer A's request to acceptor Z. If the acceptor has not seen a higher numbered request, it again promises to ignore any requests with lower proposal numbers, and sends back the highest-numbered proposal that it has accepted along with the value of that proposal. This is the case with proposer B's request to acceptors X and Y, as illustrated below:

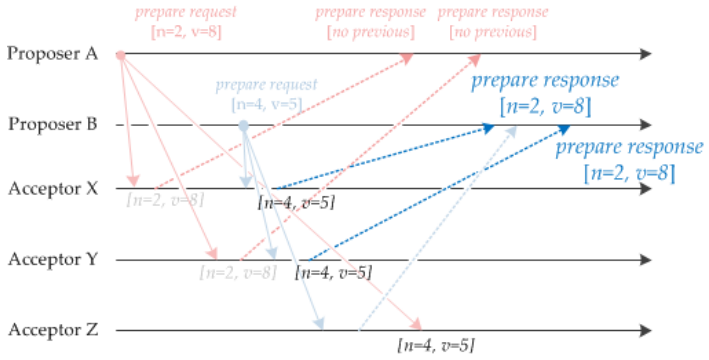

Figure 3.4: Acceptor Z ignores proposer A's request because it has already seen a higher numbered proposal (4 > 2). Acceptors X and Y respond to proposer B's request with the previous highest request that they acknowledged, and a promise to ignore any lower numbered proposals.

Once a proposer has received prepare responses from a majority of acceptors it can issue an accept request. Since proposer A only received responses indicating that there were no previous proposals, it sends an accept request to every acceptor with the same proposal number and value as its initial proposal (n=2, v=8). However, these requests are ignored by every acceptor because they have all promised not to accept requests with a proposal number lower than 4 (in response to the prepare request from proposer B).

Proposer B sends an accept request to each acceptor containing the proposal number it previously used (n=4) and the value associated with the highest proposal number among the prepare response messages it received (v=8). Note that this is not the value that proposer B initially proposed, but the highest value from the prepare response messages it saw.

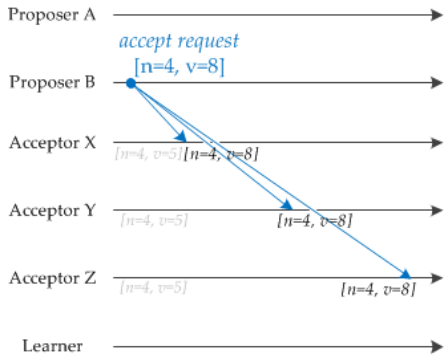

Figure 3.5: Proposer B sends an accept request to each acceptor, with its previous proposal number (4), and the value of the highest numbered proposal it has seen (8, from [n=2, v=8])

If an acceptor receives an accept request for a higher or equal proposal number than it has already seen, it accepts and sends a notification to every learner node. A value is chosen by the Paxos algorithm when a learner discovers that a majority of acceptors have accepted a value, as is illustrated below:

Once a value has been chosen by Paxos, further communication with other proposers cannot change this value. If another proposer, proposer C, sends a prepare request with a higher proposal number than has previously been seen, and a different value (for example, n=6, v=7), each acceptor responds with the previous highest proposal (n=4, v=8). This requires proposer C to send an accept request containing [n=6, v=8], which only confirms the value that has already been chosen. Furthermore, if some minority

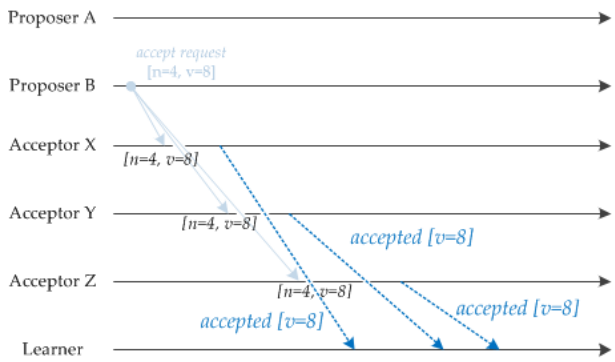

Figure 3.6: Acceptors send their accepted value to the learner

of acceptors have not yet chosen a value, this process ensures that they eventually reach consensus on the same value.

Various efficiency improvements to the standard Paxos algorithm are discussed in the papers by Lamport and Baker et al.. For example, a prepare request is not necessary if the proposer knows that it is the first to suggest a value. The proposal for such a request is numbered 0, so that it will be ignored if any higher numbered requests have been received.

Chapter 4

Performances

The motivation is to study a simple but accurate performance model to predict the performances of distributed algorithms in "standard" deployments.

To do so, we will have a look at examples of very simple broadcasting algorithms (not even implementing best effort broadcast).

To assess the performances of distributed systems, we first need metrics.

- Lantency. The time it takes between the sending of a message and its delivery by the "last" process. We consider here a system without contention (we broadcast a single message).
- Throughput. The number of messages that can be delivered (by any process) per unit of time. We consider here a system with contention (we broadcast many messages).
- Response time. This is latency, in a system with contention.

We will only talk about latency and throughput, because we do not have the theoretical tools to predict accurately the response time.

For instance, we could consider 4 machines, 2 switches, $1Gb/s$ network, P_1 and P_2 connected to a switch, P_3 and P_4 connected to the other switch, then the two switches connected together. We want to broadcast messages of $1Gb$.

With the best effort algorithm studied in the lecture. There are two settings:

- Unicast messages (i.e. point-to-point messages, like in UDP and TCP).
- Multicast messages (i.e. one-to-many messages, like IP multicast).

P_1 has an infinite number of messages to broadcast. Other processes do not have messages to broadcast. Messages are "large" (i.e. greater than $10kB$), so the bottleneck is the network. What is the throughput with the two settings? What is the optimal throughput that can be achieved with unicast messages?

We model using "rounds" (time units). A process can send at maximum one message at the start of each round. One process can receive a message at maximum at the end of each round. In our example, rounds take one second ($1Gb/s$ network and $1Gb$ messages). We consider full duplex links.

With multicast messages, the switch can duplicate the messages "for free". In other word, if four processes are connected to a same switch and one of them is doing a multicast at $1Gb/s$ to the others, then the others will receive at $1Gb/s$ (and not $\frac{1}{3}Gb/s$).

With unicast messages, P_1 sends to P_2 ($1s$), then to P_3 ($1s$), then to P_4 ($1s$). Thus, the total time to do one broadcast is $3s$, so the throughput is $\frac{1}{3}$ message per second. The latency is 3 seconds. Note that crossing a switch does not add a noticeable overhead to the time needed for a unicast.

With multicast messages, there is a latency of 1 second and a throughput of 1 message per second.

How to get a better latency with unicast messages? Make other processes involved to contribute to the broadcast (e.g. once P_2 receives a message, it sends it). In a simpler setting with the four processes connected to a single switch P_1 sends to P_2, then in parallel P_1 sends to P_3 and P_2 to P_4. We have a latency of 2 seconds and a throughput of $\frac{1}{2}$ messages per second.

How to get a better throughput with unicast messages? Again, make other processes involved. In our simpler setting, P_1 sends to P_2, then at the next round P_1 sends to P_2 (for the next broadcast) and P_2 to P_3, and then P_1 again to P_2, P_2 to P_3 and P_3 to P_4. We have a latency of $3s$ and a throughput of 1 message per second.

Be carefull, it shows that the latency and the throughput are not related (which may be counter-intuitive).

To compute the throughput, we look at how many delivers per round are done by the nodes. It is determined by the bottleneck node.

We studied the case where we have only one sender. What if there are $1 < k < N$ senders? What if there are N senders? With N senders, we can have a throughput of $\frac{N}{N-1}$ globally (and thus $\frac{1}{N-1}$ for each broadcast), again by doing a pipeline (one process sends its message every $\frac{1}{N-1}$ round, it forwards the message of another process every other round).

Pro-tip: draw the arrows in a timeline splitted in rounds.

What about our initial setting, with two switches?

Chapter 5

Causality

5.1 Definition

Causality in computer systems was introduced by Lamport in 1978: "Time, clocks and ordering of events in a Distributed System".

At the time, people took "snapshots" in distributed systems, to reduce the impact of crashes. They did snapshots just before sending a message and just after receiving one. However, when machines recovered from crash at some snapshot, there were inconsistencies: one machine has received a message that was never sent according to the snapshot. They called this issue the "domino effect".

Then, Lamport tackled the problem.

Definition 1 (History of a process)**.**

A sequence of events that are totally ordered.

$$H_p = e_1, e_2, \ldots, e_n$$

Or, with a better notation:

$$H_{p_i} = e_i^1, e_i^2, \ldots, e_i^n$$

Definition 2 (Event)**.**

We consider three types of events.

- Local events.
- Message sending.
- Message reception.

Definition 3 (Causality relationship).

An event e_i causally preceeds an event e_j, denoted $e_i \to e_j$, if and only if one of these conditions hold:

1. Events e_i and e_j are local events of the same process p and e_i has happened before e_j.
2. There exists a message m such that $e_i = sending(m)$ and $e_j = receiving(m)$.
3. There exists e' such that $e_i \to e'$ and $e' \to e_j$ (transitivity).

Note that causality is a partial order.

Note: causality actually refines to "potential" causality. More precisely, $e_i \to e_j$ means that e_i is maybe a cause of e_j, but this is not necessarily the case. However, if $e_i \to e_j$ then e_j is not a cause of e_i.

5.2 Encoding the causality relationship

Exercise: we want to timestamp each event with a logical clock $LC(e)$ such that $e_i \to e_j \Rightarrow LC(e_i) < LC(e_j)$. Motvation: if event e_p^n is a bug, all potential causes are events that have a clock strictly smaller than $LC(e_p^n)$.

Every process has a local clock, and we embed it in every event. When we send a message or do a local event, we increment the local clock. When we receive a message, if its local clock is lower than our global clock, we do nothing, otherwise we replace our local clock by this one incremented.

This is called a **Lamport clock**.

5.3 Causal order broadcast

Specifications

Validity: same as (uniform) reliable broadcast.

Agreement: same as (uniform) reliable broadcast.

Integrity: same as (uniform) reliable broadcast.

Causal order: If m_1 and m_2 are such that $sending(m_1) \to sending(m_2)$, then every correct process must deliver m_1 before m_2.

Algorithm Each process P_i has a vector VC_{P_i} of N integers, where N is the number of processes. It is initially set to $(0, \ldots, 0)$.

- When P_i sends a message, it increments $VC_{P_i}[i]$ and timestamps m with $TS(m) = VC_{P_i}$.
- When P_i receives a message m sent by P_k, if $\forall j \neq i, TS(m)[j] \leqslant VC_{P_i}[j] \wedge TS(m)[k] = VC_{P_i}[k] + 1$, then:
  - deliver the message
  - update $VC_{P_i}[k] = TS(m)[k]$
  - process the messages previously received but not yet delivered (maybe some of them can be delivered now)

otherwise, store the message.

Warning: all messages are sent to all processes. If one want to combine this with unicast messages, the algorithm does not work as is.

Remark: causal broadcast is strictly stronger than FIFO broadcast.

5.4 Causally-ordered point-to-point communications

Consider three processes, P_1 sends m_1 to P_3 and then m_2 to P_2. Then, P_2 sends m_3 to P_3. If we do not take care, we can have $sending(m_1) \rightarrow sending(m_2) \rightarrow delivery(m_2) \rightarrow sending(m_3)$ and $sending(m_1) \rightarrow sending(m_3)$.

We apply the same trick than for the broadcast, but instead of using vectors we use matrices. The value (i, j) in the matrix of some process is the knowledge of this process about the number of messages sent from P_i to P_j.

However, if one machine crash, the whole system may be blocked forever. In our example for instance, if m_1 is never received because of a crash of P_1, then m_3 will not be delivered (expected), but no message from P_2 to P_3 will be delivered in the future. The only way to overcome this issue is to have a perfect failure detector $\mathcal{P}$.

Another problem is the huge cost of this, matrices sizes increase quadratically with the number of processes. Solutions for this may be to use sparse matrices (in general, they mostly contain 0) and/or to split the distributed system in sub-regions.