Distributed System Notes

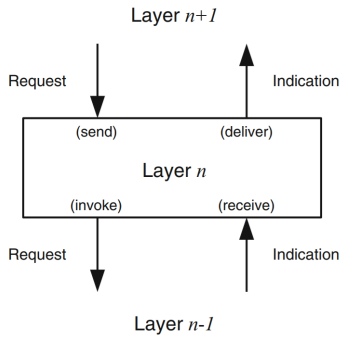
Giuliano Abruzzo

December 28, 2019

Contents

1 Modelling Distributed System

A **distributed system** is a set of *entities, computers, machines* **communicating, coordinating** and **sharing resources** in order to reach a **common goal**, and appearing as a **single computing system**. To explain several situations in *distributed systems*, we will use **distributed abstraction** cause they captures common *properties* of a large range of *systems*, and they prevent reinventing the same solution for variants of the same problem. We will use the Composition Model, where there are several elements represented as:



- **Request**: *events* used by a *component* to **request** a *service* from another *component*;
- **Confirmation**: *events* used by a *component* to **confirm** the *competition* of a *request*;
- **Indication**: *events* used by a *component* to **deliver information** to another *component*;

We will use these three in **modules description** in the *specification*. For a *simple sync* and *async Job Handler* we will have the following pseudocode:

Module 1.1: Interface and properties of a job handler

Module:

Name: JobHandler, **instance** *jh*.

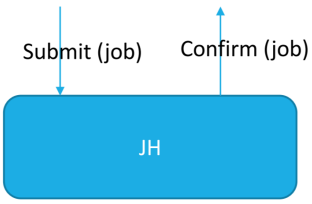
Events:

Request: ⟨ *jh*, *Submit* | *job* ⟩: Requests a job to be processed.

Indication: ⟨ *jh*, *Confirm* | *job* ⟩: Confirms that the given job has been (or will be) processed.

Properties:

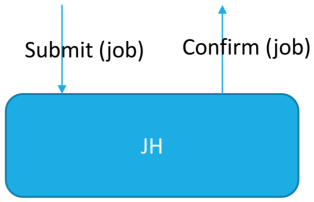
JH1: *Guaranteed response:* Every submitted job is eventually confirmed.



Algorithm 1.1: Synchronous Job Handler

Implements:
JobHandler, **instance** *jh*.

upon event ⟨ *jh*, *Submit* | *job* ⟩ **do**
process(*job*);
trigger ⟨ *jh*, *Confirm* | *job* ⟩;



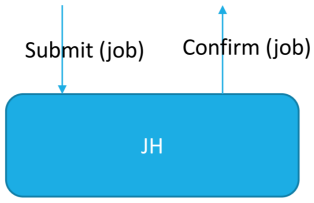
Algorithm 1.2: Asynchronous Job Handler

Implements:
JobHandler, **instance** *jh*.

upon event ⟨ *jh*, *Init* ⟩ **do**
buffer := ∅;

upon event ⟨ *jh*, *Submit* | *job* ⟩ **do**
buffer := *buffer* ∪ {*job*};
trigger ⟨ *jh*, *Confirm* | *job* ⟩;

upon *buffer* ≠ ∅ **do**
job := *selectjob*(*buffer*);
process(*job*);
buffer := *buffer* \ {*job*};



If we need to show a *Job transformation* and *processing abstraction* we have to use another *model*, with **two components**, a **Transformation Handler** and the **Job Handler**, in pseudocode:

Module 1.2: Interface and properties of a job transformation and processing abstraction

Module:

Name: TransformationHandler, **instance** *th*.

Events:

Request: ⟨ *th*, *Submit* | *job* ⟩: Submits a job for transformation and for processing.

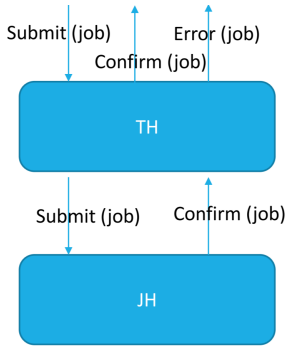
Indication: ⟨ *th*, *Confirm* | *job* ⟩: Confirms that the given job has been (or will be) transformed and processed.

Indication: ⟨ *th*, *Error* | *job* ⟩: Indicates that the transformation of the given job failed.

Properties:

TH1: *Guaranteed response:* Every submitted job is eventually confirmed or its transformation fails.

TH2: *Soundness:* A submitted job whose transformation fails is not processed.



Algorithm 1.3: Job-Transformation by Buffering

Implements:
TransformationHandler, **instance** *th*.

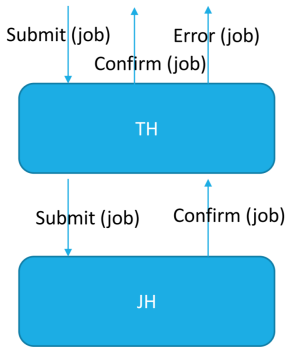
Uses:
JobHandler, **instance** *jh*.

upon event ⟨ *th*, *Init* ⟩ **do**
top := 1;
bottom := 1;
handling := FALSE;
buffer := $[\perp]^M$;

upon event ⟨ *th*, *Submit* | *job* ⟩ **do**
if *bottom* + *M* = *top* **then**
trigger ⟨ *th*, *Error* | *job* ⟩;
else
buffer[*top* mod *M* + 1] := *job*;
top := *top* + 1;
trigger ⟨ *th*, *Confirm* | *job* ⟩;

upon *bottom* < *top* ∧ *handling* = FALSE **do**
job := *buffer*[*bottom* mod *M* + 1];
bottom := *bottom* + 1;
handling := TRUE;
trigger ⟨ *jh*, *Submit* | *job* ⟩;

upon event ⟨ *jh*, *Confirm* | *job* ⟩ **do**
handling := FALSE;



Only the *operations* referred to the **arrows entering** in the **Transformation Handler** are *implemented*, because these are the *operations* the *Transformation Handler* has to handle. So, we write pseudocode which follows the given *specification*, where *operations* and *properties* are listed. The *bottom* and *top* variables are used alongside with the *buffer size* M to ensure that the limit of jobs that can be processed by the *buffer* is not exceeded. Now we have two implements two exercises:

Module 1.1 Interface of a printing module

Module:

Name: Print.

Events:

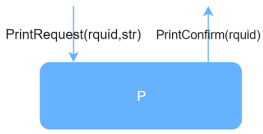
Request: ⟨ *PrintRequest* | rqid, str ⟩: Requests a string to be printed. The token rqid is an identifier of the request.

Confirmation:⟨ *PrintConfirm* | rqid ⟩: Used to confirm that the printing request with identifier rqid succeeded.

Algorithm 1 Print Module

Implements:
Print, **instance:** p;

upon event ⟨ $p, PrintRequest \mid rqid, str$ ⟩ **do**
print(str);
trigger ⟨ $p, PrintConfirm \mid rqid$ ⟩



Module 1.2 Interface of a bounded printing module

Module:

Name: BoundedPrint.

Events:

Request: ⟨ *BoundedPrintRequest* | rqid, str ⟩: Request a string to be printed. The token rqid is an identifier of the request.

Confirmation:⟨ *PrintStatus* | rqid, status ⟩: Used to return the outcome of the printing request: Ok or Nok.

Indication:⟨ *PrintAlarm* ⟩: Used to indicate that the threshold was reached.

Algorithm 1 Bounded Printing Module

Implements:
BoundedPrint, **instance:** bp;

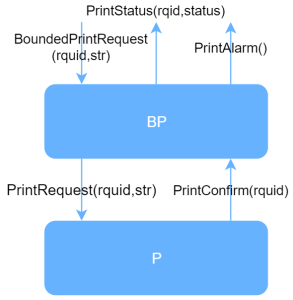
Uses:
Print, **instance:** p;

upon event ⟨ $bp \mid Init$ ⟩ **do**
$bottom := 1$;
$top := 1$;
$handling := FALSE$;
$buffer := [\perp]^{M \times 2}$;

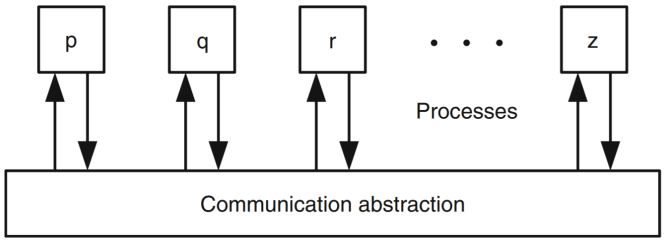
upon event ⟨ $bp, BoundedPrintRequest \mid rqid, str$ ⟩ **do**
if $bottom + M = top$ **then**
trigger ⟨ $bp, PrintAlarm$ ⟩;
trigger ⟨ $bp, PrintStatus \mid status = NOK$ ⟩;
else
$buffer\ [\ top \bmod M + 1] := rquid, str$;
$top = top + 1$;
trigger ⟨ $bp, PrintStatus \mid status = OK$ ⟩

upon $bottom < top \wedge handling = FALSE$ **do**
$rquid, str := buffer\ [\ top \bmod M + 1]$;
$bottom = bottom + 1$;
$handling = TRUE$;
trigger ⟨ $p, PrintStatus \mid rquid, str$ ⟩;

upon event ⟨ $p\ , PrintStatus \mid rquid$⟩ **do**
$handling = FALSE$;



Processes in a *Distributed System* often communicate through **messages**. We can represent a **distributed algorithm** as a series of **automata**, one per *process*, which define how to react to a *message*. The execution of a *distributed algorithm* is represented by a sequence of steps executed by the *processes*:



Distributed Algorithms should have **two fundamental properties**:

- **Safety**: which states that the *algorithm* should not behave in a wrong way;
  - a *safety property* is a property that can be violated at some time t and never be satisfied again after that time;
  - a *safety property* is a property such that, whenever it is violated in some execution E of an *algorithm*, there is a *partial execution* E' of E such that the *property* will be violated in any extension of E;
- **Liveness**: which ensure that eventually something good happens;
  - a *liveness property* is a property of a *distributed system* execution such that, for any time t, there is some hope that the property can be satisfied at some time $t' \geq t$.

There are three different **timing assumptions**:

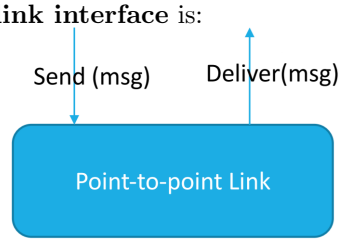
- **Synchronous**: where there is a known *upper bound* on the *time* token for *processing*, *communication*, and a *drift* of a *local lock* with respect to *real time*;
- **Asynchronous**: where there are no *timing assumptions* on *processes* and *links*, but we can use *logical clock* to measure *time* with respect to *communication*;
- **Partially**: where there is an *unknown time* t after which the system behaves as a *synchronous system*. It will be a period long enough to terminate the *distributed algorithm*;

2 Links

Links are used to model the *network component* of a *distributed system*, in fact they connect pairs of *processes*. We have three different types of *link*:

- **Fair-loss links**;
- **Stubborn links**;
- **Perfect links**;

The two **processes** linked can *crash*, and the time taken to execute an *operation* is *bounded*, and the **messages** can be *lost* and can take an indefinite time to reach the *destination*. The **generic link interface** is:



A **message** is typically received at a given port of the *network* and stored in some *buffer*, then some *algorithm* is executed to satisfy the *properties* of the *link abstraction*, before the *message* is actually *delivered*. Remember, **Deliver** is different from *receive*.

2.1 Fair-Loss P2P link

The **Fair-Loss point-to-point link** specification has two *operations*, *Send* and *Deliver* and three *properties*:

- **Fair-loss**: if a correct process p infinitely often sends a message m to a correct process q, then q delivers m an infinite number of times;
- **Finite duplication**: if a correct processes p sends a message m a finite number of times to process q, then m cannot be delivered an infinite number of times by q;
- **No creation**: if some process q delivers a message m with sender p, then m was previously sent to q by p;

The *sender* must take care of the *retransmissions* if it wants to be sure that m is delivered at its *destination* and there is no guarantee that the *sender* can stop the retransmissions of each *message*, and each *message* may be delivered more than once.

2.2 Stubborn P2P link

The Stubborn point-to-point link has has two *operations*, *Send* and *Deliver* and two *properties*:

- **Stubborn delivery**: if a correct process p sends a message m once to a correct process, then q delivers m an infinite number of times;
- **No creation**: *same as fair-loss p2p*;

The implementation of the **Stubborn P2P link** is:

Algorithm 2.1: Retransmit Forever

Implements:
StubbornPointToPointLinks, **instance** *sl*.

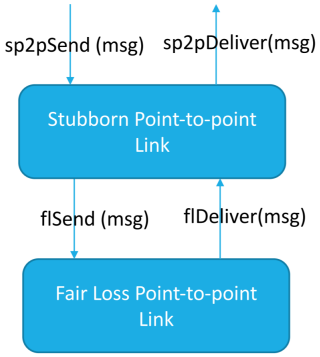
Uses:
FairLossPointToPointLinks, **instance** *fll*.

upon event ⟨ *sl*, *Init* ⟩ **do**
sent := ∅;
starttimer(Δ);

upon event ⟨ *Timeout* ⟩ **do**
forall $(q, m) \in sent$ **do**
trigger ⟨ *fll*, *Send* | q, m ⟩;
starttimer(Δ);

upon event ⟨ *sl*, *Send* | q, m ⟩ **do**
trigger ⟨ *fll*, *Send* | q, m ⟩;
sent := *sent* ∪ $\{(q, m)\}$;

upon event ⟨ *fll*, *Deliver* | p, m ⟩ **do**
trigger ⟨ *sl*, *Deliver* | p, m ⟩;



2.3 Perfect P2P link

The **perfect link** solves all the issues presented above, in fact besides the *No Duplication* and *No Creation* it has another property:

- **Reliable delivery**: if a correct process p sends a message m to a correct process q, then q eventually delivers m;

Algorithm 2.2: Eliminate Duplicates

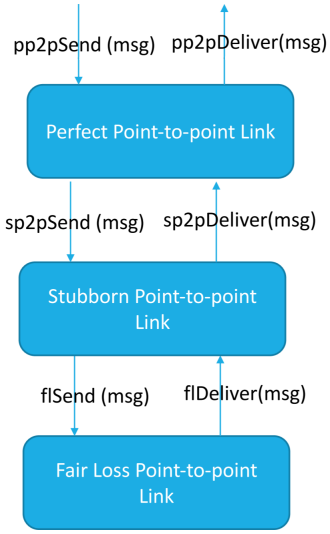
Implements:
PerfectPointToPointLinks, **instance** *pl*.

Uses:
StubbornPointToPointLinks, **instance** *sl*.

upon event ⟨ *pl*, *Init* ⟩ **do**
delivered := ∅;

upon event ⟨ *pl*, *Send* | q, m ⟩ **do**
trigger ⟨ *sl*, *Send* | q, m ⟩;

upon event ⟨ *sl*, *Deliver* | p, m ⟩ **do**
if $m \notin$ *delivered* **then**
delivered := *delivered* ∪ $\{m\}$;
trigger ⟨ *pl*, *Deliver* | p, m ⟩;



We can observe that the *No duplication* is ensured by the last piece of pseudocode, thanks to the variable *delivered*. *No creation* is inherited, and *Reliable delivery* derives from the whole schema, thanks to the *perfect link* in particular which is built by chance to deliver *messages* correctly.

3 Physical Time

In a *distributed system* **processes** run on different *nodes* interconnected by mean of a *network* and cooperate to complete a *computation*. They communicate only through *messages* and, as *ordering* is require in such application, **time** is a *critical factor* for *distributed systems*. Each *process* p_i in a *distributed* system runs on a single *mono-processor machine* with no *shared memory*, and they have a *state* s_i, changed by the *actions* during the *algorithm execution*. Each *process* generate a sequence of *events*:

- **Internal Event**: *events* that transforms the *process state*;
- **External Event**: *send/receive*;
- $\mathbf{e_i^k}$: that represent the *k-th event* of the *process* p_i;

We indicate with:

- $\rightarrow_i$ the **ordering relation** between two events e and $e^{'}$;
- $e \rightarrow_i e^{'}$ if and only if e happened before $e^{'}$;

We call **local history** the sequence of *events* produced by a *process*, a **partial local history** a prefix of a local history, and **global history** the set containing every *local history*. *Events* can be **time-stamped** through **physical clocks** values, in fact in a *single process* is always possible to order *events*, but in a *distributed system* in presence of *network delay*, it is **impossible** to realize a *common clock* shared among every *process*. Anyway, it is possible to use **timestamps** in order to synchronize *physical clocks* through *algorithms* with a certain degree of approximation:

$$C_i(T) = \alpha \cdot H_i(t) + \beta$$

Where $C_i(t)$ represent the **software clock** and $H_i(t)$ represent the **hardware clock**, α and β are two factors which approximate the result closer to guarantee *monotonicity*. This *software clock* is not generally completely *accurate*, in fact it can be different from the *real time* and at any *process* due the precision of the approximation. We have to keep the **granularity**, also called **resolution**, the interval of time between two increments of the *software clock*, of the *software clock* smaller than the *time difference* between two *consequent events* so:

$$T_{resolution} < \Delta T_{between\ two\ notable\ events}$$

There are two parameters that effect **physical clocks**:

- **Skew**: the difference between two *clocks*: $|C_i(t) - C_j(t)|$;
- **Drift rate**: the **gradual misalignment** of once *synchronized clocks* caused by the slight *inaccuracies* of the time-keeping mechanism, for example the *ordinary quartz clocks* deviate of 1 sec in 12 days;

UTC is the international standard for *clock synchronization*, and we can have two types of *synchronization*:

- **External Synchronization**:
  - In which *processes* synchronize their *clock* C_i with an *UTC source* S, in a way such that for each time interval: $|S(t) - C_i(t)| < D$ where D is a *synchronization bound*;
- **Internal Synchronization**:
  - In which all the *processes* synchronize their *clock* C_i between them with respect to D in pairs so: $|C_i(t) - C_j(t)| < D$;

So, the *clocks* that are **internally synchronized** are not necessarily **externally synchronized**, instead the *clocks* that are **externally synchronized** are also **internally synchronized** with a *bound* of $2 \cdot D$.

An hardware clock is **correct** if its *drift rate* is within a *limited bound* of $p > 0$:

$$1 - p \leq \frac{dC}{dT} \leq 1 + p$$

If we have a **correct hardware clock** H we can measure a *time interval* $\left[t, t^{'}\right]$:

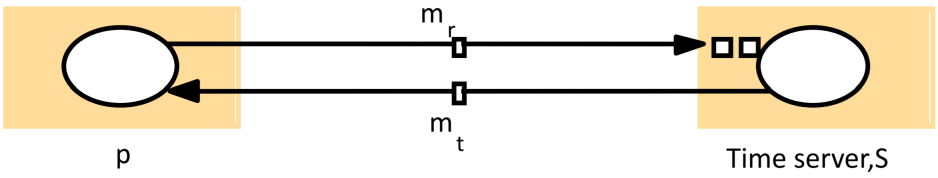
$$(1 - p)(t^{'} - t) \leq H(t^{'}) - H(t) \leq (1 + p)(t^{'} - t)$$

Software clocks have to be **monotone**: $t^{'} > t$, so $C(t^{'}) > C(t)$.

3.1 Synchronization Algorithms

3.1.1 Christian's Algorithm

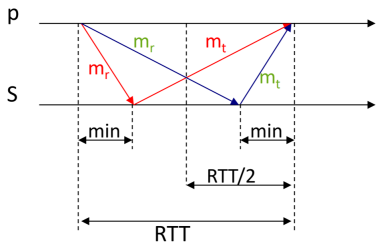
Christian's algorithm is an *external synchronization algorithm* which uses a *time server* S that receives signal from an *UTS source*. It works also in an *asynchronous system* in a probabilistic way. Is based on **message round trip** or **RTT**, and *RTTs* must be small enough in order to obtain *synchronization*.



A process p asks the current time through m_r and receives t in m_t from S, and p will set its time to $t + \frac{RTT}{2}$, where RTT is the round trip time experience by p. Is important to note that the *time server* can crash or can be hacked.

Accuracy of this *algorithm* strongly depends of RTT, and we can have two cases:

- **Case 1**:
  - In this case the **real reply time** is greater than **estimated time** that is $\frac{RTT}{2}$, so in particular is equal to $RTT - min$;
  - $\Delta = estimated - real = \frac{RTT}{2} - (RTT - min) = -(\frac{RTT}{2} - min)$;
- **Case 2**:
  - In this case the **real reply time** is smaller than **estimated time** that is $\frac{RTT}{2}$, so in particular is equal to min;
  - $\Delta = estimated - real = \frac{RTT}{2} - min = +(\frac{RTT}{2} - min)$;



So the the **accuracy** of *Cristian's Algorithm* is $\pm(\frac{RTT}{2} - min)$ where min is the *minimum transmission delay*.

3.1.2 Berkeley's Algorithm

Berkeley's Algorithm is an *internal synchronization algorithm* with a **master/slave structure** and is based on two *steps*. In the first step gathering of all the *clocks* from *processes* and computation of the *difference*, and in the second step there is the computation of the *correction*. The accuracy of this protocol depends on the maximum round-trip time. If the *master* crashes, another one is *elected*, and it's tolerant to *arbitrary behavior*, like a *slave* that sends a wrong value, since the *master* uses a *threshold*.

The **master process** computes the differences Δp_i between the *master clock* and the *clock* of every process p_i (even him self), after it compute the average, avg, of all the differences Δp_i without considering *faulty processes* (a process with a clock which differ from the *master* one more than a *threshold*) and at the end it computes the **correction** of each process (even *faulty*) with the formula: $ADG_{p_i} = avg - \Delta p_i$.

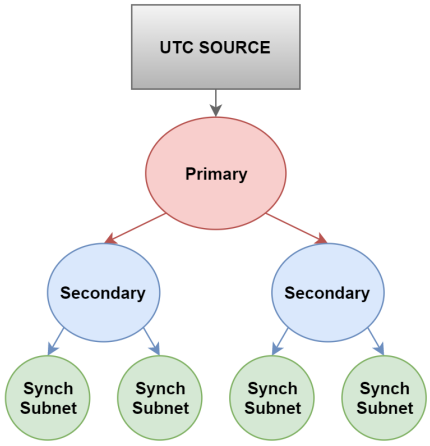
When a **slaves process** receives the *correction*, it is applied to the *local clock*. If the *correction* is negative, the *process* doesn't adjust the value but it slow down its *clock*, since decrementing can cause problems.

3.1.3 Network Time Protocol

The **NTP**, or **Network Time Protocol**, is a time service over *Internet* that synchronizes *clients* with *UTC*. It's a standard for *external clock synchronization* of *distributed systems* and employs several *security mechanisms* and is based on a **remote reading procedure** like the *Cristian's Algorithm*. Furthermore it adds basic algorithm mechanisms for *clustering*, *filtering* and *evaluating data quality*.

It works with a **hierarchy**:

- **Primary server**: connected directly to *UTC sources*;
- **Secondary server**: synchronized to *primary servers*;
- **Synchronization subnet**: lowest *level servers* in *users computers*;



This **hierarchy** in case of *faults* is *reconfigurable*. There are three modes of **synchronization**:

- **Multicast**: in which the *server* periodically sends its *actual time* to its *leaves*;
- **Procedure Call**: in which the *server* replies to request with its *timestamp*;
- **Symmetrical**: in which we synchronize the pairs of *time servers* using messages containing *timing information*;

It's important to note that **physical synchronization** doesn't work in **asynchronous system** as we completely make the *bound logic* useless, in fact *time for answer* is unpredictable.

4 Logical Time

4.1 Logical clock

As we said, **physical clock** are good if we have a precise estimation of *delays*, but this can be hard, and often we want to know in which *order* some *events* happened and not the *exact time* for each of them. Since in a *distributed system* each *system* has its own *logical clock*, if *clocks* are not aligned it's not possible to order *events* generated by different *processes*, so we need a reliable way to order *events* and this is the **logical clock**. These *clocks* are based on the *causal relations* between events, and it's important to define two relations:

- $\rightarrow_i$ is the **ordering relation** between *events* in a *process* p_i;
- $\rightarrow$ is the **happened-before** between any pairs of *events*;

We say that two *events* e and $e^{'}$ are in **happened-before** relation if:

- $\exists\ p_i \mid e \rightarrow_i e^{'}$;
- $\forall$ message $m : send(m) \rightarrow receive(m)$;
  - $send(m)$ is the event of **sending** a message m;
  - $receive(m)$ is the event of **receipt** of the same message m;
- $\exists\ e, e^{'}, e^{''} \mid (e \rightarrow e^{''}) \wedge (e^{''} \rightarrow e^{'})$;

Where the last rule says that the **happened-before** relation is **transitive**. Using these rules we can define a **causal ordered sequence of events**, and if there are two *events* that are not in *happened-before* relation they are **concurrent** ($e \parallel e^{'}$). The **logical clock** is a *software counting register* **monotonically** increasing its value and it's not related to the *physical clock* in any way. We denote with $L_i(e)$ the **logical timestamp** assigned by the *logical clock* by a *process* p_i to the *event* e. There is a property that says:

$$if\ e \rightarrow e^{'}\ then\ L(e) < L(e^{'})$$

There are two main implementations of the **logical clock**:

4.1.1 Scalar Logical Clock

Each *process* p_i initializes its **logical clock** $L_i = 0$, and p_i increases its *logical clock* of 1 when it generate an *event* (send or receive): $L_i = L_i + 1$

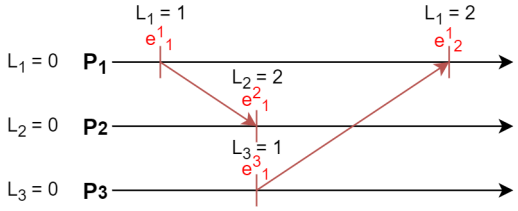
When p_i **sends** m:

- Create an event $send(m)$;
- Increment L_i;
- Timestamp m with a $t = L_i$;

When p_i **receives** m:

- Update $L_i = max(t, L_i)$;
- Produce an event $receive(m)$;
- Increment L_i;

As we said if e $\rightarrow e^{'}$ then $L(e) < L(e^{'})$ but the contrary is not valid, so it possible that the two *events* are *concurrent*. So we cannot determine if two *processes* are in *happened-before* by analyzing *scalar clocks* only.



In fact we can see that $L(e_1^3) < L(e_1^2)$ but: $e_1^2 \parallel e_1^3$;

4.1.2 Vector Logical Clock

Each *process* has an *array* of N integers where N is the number of *processes* that are taken into consideration, this *array* is called **Vector Clock**, and each *process* maintains is own. *Vector Clock* resolves the old problem cause here we have:

$$e \rightarrow e^{'} \textbf{ iff } L(e) < L(e^{'})$$

Each process p_i initialize its **vector clock** V_i: $V_i[j] = 0\ \forall\ j = 1\ ...\ N$, and p_i increases $V_i[i]$ of 1 when it generates an event: $V_i[i] = V_i[i] + 1$;

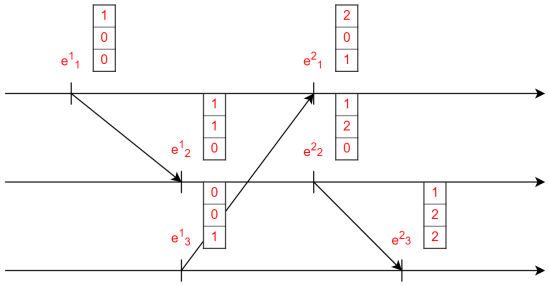
When p_i **sends** m:

- Create an event $send(m)$;
- Increment V_i;
- Timestamps m with a $t = V_i$;

When p_i **receives** m:

- Update $V_i[j] = max(t[j], V_i[j])\ \forall\ j = 1\ ...\ N$;
- Produce an event $receive(m)$;
- Increment V_i;

The implementation is like before, with the difference that the *update* of the vector are done in an *index* according to the *event* that generated the *message*. So $V_i[i]$ represents the numbers of *events* produced by p_i, $V_i[j]$ represents the number of *events* of p_j that p_i knows. Two *events* are in **happened-before** only iff $V \leq V^{'} \wedge V \neq V^{'}$ so it must be $V < V^{'}$.



- event $e_1^1 \begin{bmatrix}1\\0\\0\end{bmatrix} <$ event $e_2^1 \begin{bmatrix}1\\1\\0\end{bmatrix}$ so we have: $e_1^1 \rightarrow e_2^1$
- event $e_1^1 \begin{bmatrix}1\\0\\0\end{bmatrix} \not< $ event $e_3^1 \begin{bmatrix}0\\0\\1\end{bmatrix}$ so we have: $e_1^1 \parallel e_3^1$

Differently from *Scalar Clock*, **Vector Clock** allows to determine if two *events* are *concurrent* or in *happened-before*.

4.2 Logical Time and Distributed Algorithms

We have seen two mechanism to represent **logical time**, the *scalar clock* and the *vector clock*. We will now discuss how **logical-based algorithms** can be implemented in *distributed systems*. But first the **mutual exclusion abstraction** specification are:

- **Mutual Exclusion**: at every time t at most one *process* p is in *critical section*;
- **No-Deadlock**: there always exists a *process* p able to enter in *critical section*;
- **No-Starvation**: every *process* p requesting the *critical section* eventually gets in;

4.2.1 Lamport's algorithm

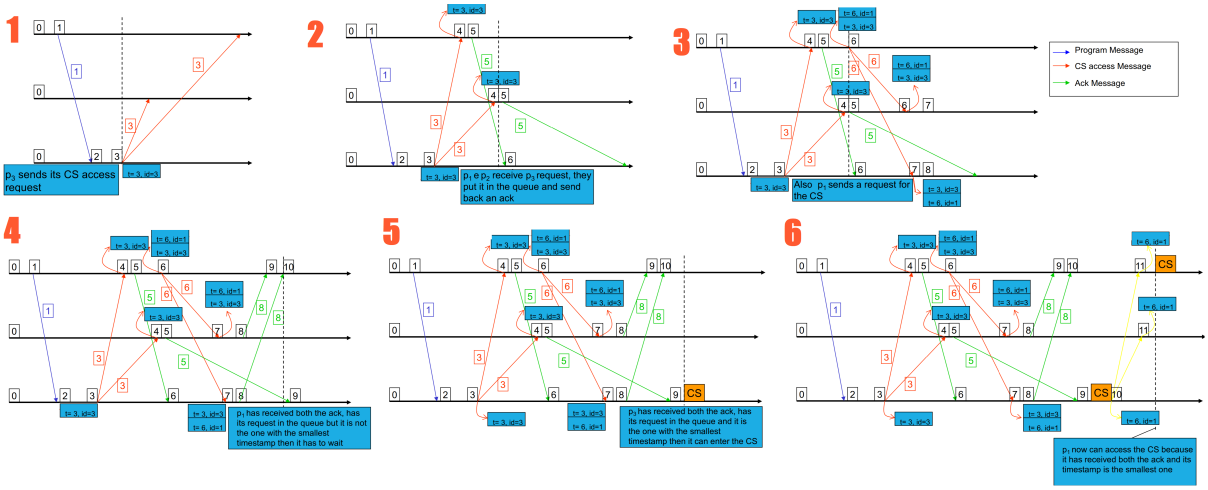
In the **Lamport's algorithm** when a *process* want to enter the *critical section* sends a *request message* to all the other, and a counter, *timestamp*, is used for maintaining an history of the operations and this *counter* is incremented for each *event* and when a *message* (even not related to *mutual exclusion computation*) is *sent* or *received*.

Each *process* p_i uses a **data structure** composed of:

- $\mathbf{C_k}$: a *counter* for process p_i;
- **Q**: a *queue* maintained by p_i where *critical section requests* are stored;

The **algorithms rules** for *process* p_i are:

- **Access the CS**:
  - p_i sends a *request message* attaching C_k to all *processes* and adds its *request* to Q;
- **Request reception from** p_j:
  - p_i puts p_j *request* (*timestamp* included) in its *queue* and sends back an ACK to p_j;
- **p_i enters the CS iff**:
  - p_i has in its queue a *request* with *timestamp* t;
  - t is the smallest *timestamp* in the *queue*;
  - p_i has already received an ACK from another *process* with $t^{'} > t$;
- **Release of the CS**:
  - p_i sends a *release message* to all the other *processes* and deletes its own *request* from the *queue*;
- **Reception of a release message from** p_j:
  - p_j deletes p_i *request* from the *queue*;



The **safety proof** can be explained as follow: let's suppose by **contradiction** that both the *processes* p_i and p_j enter the *critical section*, this means that both have received an ACK from any other *process* and the *timestamp* has to be the smallest in the *queue*:

- $t_i < t_j < ACK_i.ts$;
- $t_j < t_i < ACK_j.ts$

So we have three cases:

- p_j ACK arrives before p_j request then p_i can enter the CS without any problem;
- p_j ACK arrives after p_j request but before p_i ACK then p_i enters the CS without any problem and sends its ACK after executing the CS;
- Both processes receive the ACK when the two requests are in queue but *mutual exclusion* is guaranteed by the *total order* on the timestamps;

Fairness is satisfied because different *requests* can be either in *happened-before* or in *concurrent* relation, so in the first case everything is done with the respect of that *order*, while in the second case the *CS access* can happened in any *order*. In the worst case, this *algorithm* needs $3(N-1)$ messages for the *CS execution*.

4.2.2 Ricart-Agrawala's algorithm

Each *process* has:

- **Replies** initially 0;
- **State** $\in [Requesting, CS, NCS]$;
- **Q** a *queue* for *pending requests* initially empty;
- **Last˙Req**;
- **Num**;

And the algorithm is:

begin
1. State=Requesting
2. Num=num+1; Last_Req=num
3. ∀ i=1…N send REQUEST(num) to pi
4. Wait until #replies=n-1
5. State=CS
6. CS
7. ∀ r∈Q send REPLY to r
8. Q= ∅; State=NCS; #replies=0

Upon receipt REQUEST(t) from pj
1. If State=CS or (State=Requesting and {Last_Req,i}<{t,j})
2. Then insert in Q{t, j}
3. Else send REPLY to pj
4. Num=max(t,num)

Upon receipt of REPLY from pj
1. #replies=#replies+1

This algorithm sends the *processes* into **critical section** based on the *number* of the process, or on the basis of a *deterministic function* that ensures the **total order**.

5 Failure Detection & Leader Election

A system is *synchronous/asynchronous* or *partially synchronous* depending on the **timing assumption**. If they are explicit we talk about a *synchronous system*, otherwise it is *asynchronous*. *Partially systems* are the ones that need *abstract timing* assumptions and we have two choices:

- Put assumption on the *system model* (including *links* and *process*);
- Create a separate *abstractions* that encapsulate *timing assumptions*;

5.1 Failure Detector Abstraction

A **failure detector abstraction** is a *software module* used to detect **faulty processes**, it encapsulate *timing assumptions* of a either *partially synchronous* or *fully synchronous system*. It has two properties:

- **Accuracy**: that represents the ability to avoid *mistakes*;
- **Completeness**: that represents the ability to detect all *failures*;

5.1.1 Perfect Failure Detector

Module 2.6: Interface and properties of the perfect failure detector

Module:

Name: PerfectFailureDetector, **instance** $\mathcal{P}$.

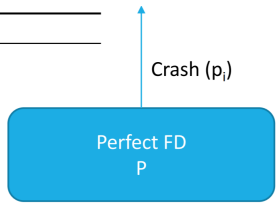
Events:

Indication: $\langle\ \mathcal{P}, Crash \mid p\ \rangle$: Detects that process p has crashed.

Properties:

PFD1: *Strong completeness:* Eventually, every process that crashes is permanently detected by every correct process.

PFD2: *Strong accuracy:* If a process p is detected by any process, then p has crashed.



Algorithm 2.5: Exclude on Timeout

Implements:
PerfectFailureDetector, **instance** $\mathcal{P}$.

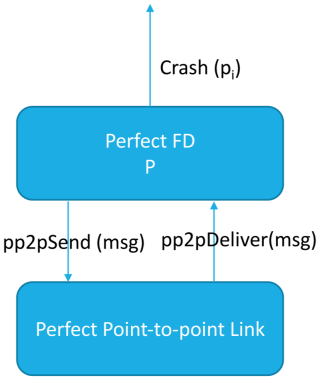
Uses:
PerfectPointToPointLinks, **instance** *pl*.

upon event $\langle\ \mathcal{P}, Init\ \rangle$ **do**
alive := Π;
detected := $\emptyset$;
starttimer(Δ);

upon event $\langle$ *Timeout* $\rangle$ **do**
forall $p \in \Pi$ **do**
if $(p \notin alive) \wedge (p \notin detected)$ **then**
detected := *detected* $\cup\ \{p\}$;
trigger $\langle\ \mathcal{P}, Crash \mid p\ \rangle$;
trigger $\langle$ *pl*, *Send* | *p*, [HEARTBEATREQUEST] $\rangle$;
alive := $\emptyset$;
starttimer(Δ);

upon event $\langle$ *pl*, *Deliver* | *q*, [HEARTBEATREQUEST] $\rangle$ **do**
trigger $\langle$ *pl*, *Send* | *q*, [HEARTBEATREPLY] $\rangle$;

upon event $\langle$ *pl*, *Deliver* | *p*, [HEARTBEATREPLY] $\rangle$ **do**
alive := *alive* $\cup\ \{p\}$;



To prove the **correctness** we have to show that both the *property* are satisfied. In this case they follow from the *perfect point-to-point link*, in fact, if a *process* crashes, it won't be able to send $HEARTBEATREPLY$ any more. If at the *timeout* there is a *process* that doesn't *reply* to *requests* it means that it has *crashed*.

5.1.2 Eventually Perfect Failure Detector

In the **Eventually perfect failure detector** there is an *unknown time* T after that *crashes* can be *accurately detected*. In the *asynchronous period* (so the moments before T), the *failure detector* can make mistake assuming *correct processes* as *crashed*, so the notion of *detection* becomes **suspicious**.

Module 2.8: Interface and properties of the eventually perfect failure detector

Module:

Name: EventuallyPerfectFailureDetector, **instance** $\diamond\mathcal{P}$.

Events:

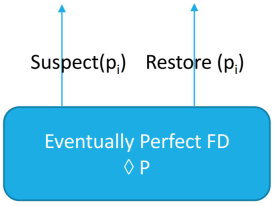
Indication: $\langle\ \diamond\mathcal{P}, Suspect \mid p\ \rangle$: Notifies that process p is suspected to have crashed.

Indication: $\langle\ \diamond\mathcal{P}, Restore \mid p\ \rangle$: Notifies that process p is not suspected anymore.

Properties:

EPFD1: *Strong completeness:* Eventually, every process that crashes is permanently suspected by every correct process.

EPFD2: *Eventual strong accuracy:* Eventually, no correct process is suspected by any correct process.



Algorithm 2.7: Increasing Timeout

Implements:
EventuallyPerfectFailureDetector, **instance** $\diamond\mathcal{P}$.

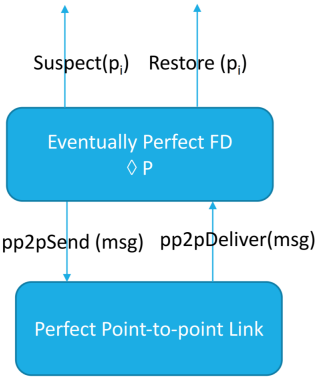
Uses:
PerfectPointToPointLinks, **instance** *pl*.

upon event $\langle\ \diamond\mathcal{P}, Init\ \rangle$ **do**
  $alive := \Pi$;
  $suspected := \emptyset$;
  $delay := \Delta$;
  starttimer($delay$);

upon event $\langle\ Timeout\ \rangle$ **do**
  if $alive \cap suspected \neq \emptyset$ **then**
    $delay := delay + \Delta$;
  forall $p \in \Pi$ **do**
    if $(p \notin alive) \wedge (p \notin suspected)$ **then**
      $suspected := suspected \cup \{p\}$;
      trigger $\langle\ \diamond\mathcal{P}, Suspect \mid p\ \rangle$;
    else if $(p \in alive) \wedge (p \in suspected)$ **then**
      $suspected := suspected \setminus \{p\}$;
      trigger $\langle\ \diamond\mathcal{P}, Restore \mid p\ \rangle$;
    trigger $\langle\ pl, Send \mid p, [\text{HEARTBEATREQUEST}]\ \rangle$;
  $alive := \emptyset$;
  starttimer($delay$);

upon event $\langle\ pl, Deliver \mid q, [\text{HEARTBEATREQUEST}]\ \rangle$ **do**
  trigger $\langle\ pl, Send \mid q, [\text{HEARTBEATREPLY}]\ \rangle$;

upon event $\langle\ pl, Deliver \mid p, [\text{HEARTBEATREPLY}]\ \rangle$ **do**
  $alive := alive \cup \{p\}$;



As we see from the **specification**, a <> P can mistakenly *suspect* a *process* but is able to *restore* it as soon as possible, as it receives a *reply*, also updating the *timeout*. This can happen when the chosen *timeout* is too short. If a *process* q *crashes* and stops to send *replies*, p doesn't change its *judgment* anymore.

5.2 Leader Election

5.2.1 Perfect Leader Election

Sometimes, we may be more interested in knowing if a *process* is **alive** instead of monitoring *failures*. We can use a different oracle which reports a *process* that is alive called **Leader Election** module. In the **perfect leader election** we use the *perfect failure detector*:

Module 2.7: Interface and properties of leader election

Module:

Name: LeaderElection, **instance** *le*.

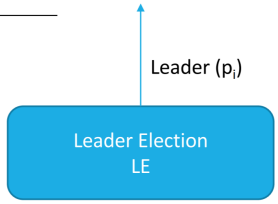
Events:

Indication: ⟨ *le*, *Leader* | *p* ⟩: Indicates that process *p* is elected as leader.

Properties:

LE1: *Eventual detection:* Either there is no correct process, or some correct process is eventually elected as the leader.

LE2: *Accuracy:* If a process is leader, then all previously elected leaders have crashed.



Algorithm 2.6: Monarchical Leader Election

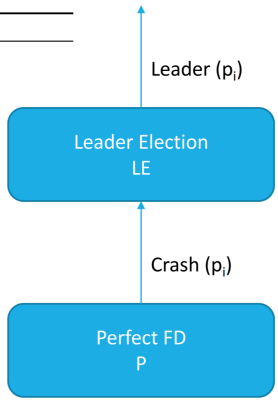
Implements:
LeaderElection, **instance** *le*.

Uses:
PerfectFailureDetector, **instance** $\mathcal{P}$.

upon event ⟨ *le*, *Init* ⟩ **do**
suspected := ∅;
leader := ⊥;

upon event ⟨ $\mathcal{P}$, *Crash* | *p* ⟩ **do**
suspected := *suspected* ∪ {*p*};

upon *leader* ≠ maxrank(Π \ *suspected*) **do**
leader := maxrank(Π \ *suspected*);
trigger ⟨ *le*, *Leader* | *leader* ⟩;



5.2.2 Eventual Leader Election

If the *failure detector* is not perfect we talk about **eventual leader election**:

Module 2.9: Interface and properties of the eventual leader detector

Module:

Name: EventualLeaderDetector, **instance** Ω.

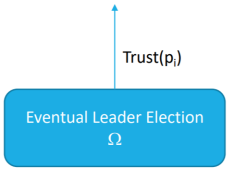
Events:

Indication: ⟨ Ω, *Trust* | *p* ⟩: Indicates that process *p* is trusted to be leader.

Properties:

ELD1: *Eventual accuracy:* There is a time after which every correct process trusts some correct process.

ELD2: *Eventual agreement:* There is a time after which no two correct processes trust different correct processes.



Algorithm 2.8: Monarchical Eventual Leader Detection

Implements:
EventualLeaderDetector, **instance** Ω.

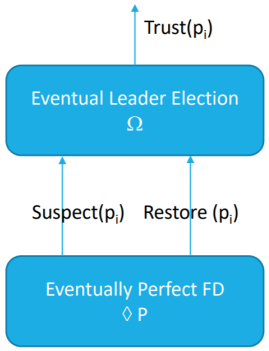
Uses:
EventuallyPerfectFailureDetector, **instance** $\diamond\mathcal{P}$.

upon event $\langle\ \Omega, Init\ \rangle$ **do**
$suspected := \emptyset$;
$leader := \perp$;

upon event $\langle\ \diamond\mathcal{P}, Suspect \mid p\ \rangle$ **do**
$suspected := suspected \cup \{p\}$;

upon event $\langle\ \diamond\mathcal{P}, Restore \mid p\ \rangle$ **do**
$suspected := suspected \setminus \{p\}$;

upon $leader \neq maxrank(\Pi \setminus suspected)$ **do**
$leader := maxrank(\Pi \setminus suspected)$;
trigger $\langle\ \Omega, Trust \mid leader\ \rangle$;



This ensures that, eventually, *correct processes* will elect the same *correct process* as their **leader**. It doesn't guarantee that *leaders* may change in an arbitrary period of time, and that many *leaders* might be elected during the same period of time without having *crashed*. Once a *unique leader* is determined and doesn't change again, we say that the leaser has **stabilized**.

5.2.3 Leader Election with Fair Lossy Links

If we have a **fair-loss link** we need to convert the whole schema and we need to use **crash-recovery** and **timeouts**:

Algorithm 2.9: Elect Lower Epoch

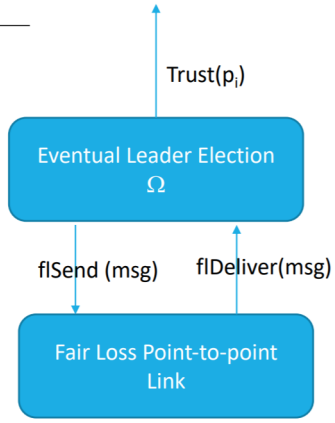
Implements:
EventualLeaderDetector, **instance** Ω.

Uses:
FairLossPointToPointLinks, **instance** *fll*.

upon event $\langle\ \Omega, Init\ \rangle$ **do**
$epoch := 0$;
store($epoch$);
~~$candidates := \emptyset$;~~
trigger $\langle\ \Omega, Recovery\ \rangle$;

upon event $\langle\ \Omega, Recovery\ \rangle$ **do**
$leader := maxrank(\Pi)$;
trigger $\langle\ \Omega, Trust \mid leader\ \rangle$;
$delay := \Delta$;
retrieve($epoch$);
$epoch := epoch + 1$;
store($epoch$);
forall $p \in \Pi$ **do**
trigger $\langle\ fll, Send \mid p, [\text{HEARTBEAT}, epoch]\ \rangle$;
$candidates := \emptyset$;
starttimer($delay$);

keeps track of how many times the process crashed and recovered



The last piece of code, in the *fill deliver event*, we will replace the number of **epoch** of a *process* that crashed again.

Algorithm 2.9: Elect Lower Epoch

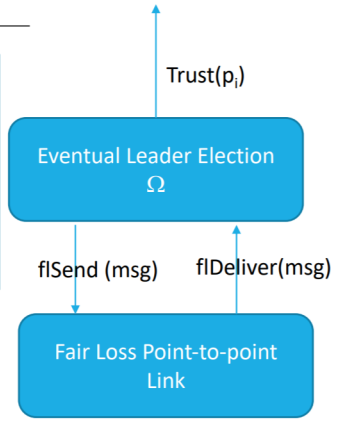
Implements:
EventualLeaderDetector, **instance** Ω.

Uses:
FairLossPointToPointLinks, **instance** *fll*.

upon event ⟨ *Timeout* ⟩ **do**
newleader := select(*candidates*);
if *newleader* ≠ *leader* **then**
delay := *delay* + Δ;
leader := *newleader*;
trigger ⟨ Ω, *Trust* | *leader* ⟩;
forall $p \in \Pi$ **do**
trigger ⟨ *fll*, *Send* | *p*, [HEARTBEAT, *epoch*] ⟩;
candidates := ∅;
starttimer(*delay*);

upon event ⟨ *fll*, *Deliver* | *q*, [HEARTBEAT, *ep*] ⟩ **do**
if exists $(s, e) \in$ *candidates* such that $s = q \wedge e < ep$ **then**
candidates := *candidates* \ {(q, e)};
candidates := *candidates* ∪ (q, ep);

deterministic function returning one process among all candidates (i.e., process with the lowest epoch number and among the ones with the same epoch number the one with the lowest identifier)

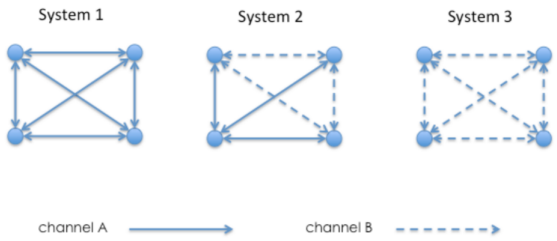


Exercise 1

Ex 1: Let channel A and channel B be two different types of point-to-point channels satisfying the following properties:

- channel A: if a correct process p_i sends a message *m* to a correct process p_j at time *t*, then *m* is delivered by p_j by time $t+\delta$.
- channel B: if a correct process p_i sends a message *m* to a correct process p_j at time *t*, then *m* is delivered by p_j with probability p_{cons} (p_{cons} <1).

Let us consider the following systems composed by 4 processes p_1, p_2, p_3 and p_4 connected trough channels A and channels B.



Assuming that each process p_i is aware of the type of channel connecting it to any other process, answer to the following questions:

1. is it possible to design an algorithm implementing a perfect failure detector in system 2 if only processes having an outgoing channel of type B can fail by crash?
2. is it possible to design an algorithm implementing a perfect failure detector in system 2 if any process can fail by crash?
3. is it possible to design an algorithm implementing a perfect failure detector in system 3?

For each point, if an algorithm exists write its pseudo-code, otherwise show the impossibility.

Exercise 1.1

The answer is **yes**, because we can implement it on the *process* that has all the *links* of **channel A**. So, in that case when the *timeout* will end if a *process* didn't send a *reply* we know for sure that it has **crashed**.

Init:
correct$_i$ = $\{p_1, p_2, p_3, p_4\}$
alive$_i$ = $\varnothing$
detected$_i$ = $\varnothing$
for each $p_j \in$ correct$_i$ **do**:
    trigger $send(HearthBeatRequest, i)$ to p_j
$start(timer_1)$

upon event $deliver(HearthBeatRequest, j)$ from p_j
    trigger $send(HearthBeatReply, i)$ to p_j

upon event $deliver(HearthBeatReply, j)$ from p_j
    alive$_i$ = alive$_i \cup \{p_j\}$

when $timer_1 = 0$
    for each $p_j \in correct_i$ **do**:
        trigger $send(ALIVE_LIST, alive_i, i)$ to p_j
    $start(timer_2)$

when $timer_2 = 0$
    for each $p_j \in correct_i$ **do**:
        if $p_j \notin$ alive$_i \wedge p_j \notin$ detected$_i$
            detected$_i$ = detected$_i \cup \{p_j\}$
            trigger $crash(p_j)$
    alive$_i$ = $\varnothing$
    for each $p_j \in$ correct$_i$ **do**:
        trigger $send(HearthBeatRequest, i)$ to p_j
    $start(timer_1)$

Exercise 1.2

We can't cause once the *process* that has all the **channels A** *fails*, it's not *guaranteed* that all the *failures* will be *detected* since the *channel* has a **probability** to lose a *message* that is not zero.

Exercise 1.3

We can't cause all the *links* are **fair loss**, so we can just implement an **eventually perfect failure detector**.

Exercise 2

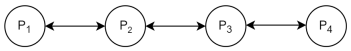
Ex 2: Consider a distributed system composed by n processes $\{p_1,p_2,\ldots,p_n\}$ that communicate by exchanging messages on top of a line topology, where p_1 and p_n are respectively the first and the last process of the network.
Initially, each process knows only its left neighbour and its right neighbour (if they exist) and stores the respective identifiers in two local variables LEFT and RIGHT.
Processes may fail by crashing, but they are equipped with a perfect oracle that notifies at each process the new neighbour (when one of the two fails) through the following primitives:

- **Left_neighbour(p_x):** process p_x is the new left neighbour of p_i
- **Right_neighbour(p_x):** process p_x is the new right neighbour of p_i

Both the events may return a NULL value in case p_i becomes the first or the last process of the line.
Each process can communicate only with its neighbours.

Write the pseudo-code of an algorithm implementing a Leader Election primitive assuming that channels connecting two neighbour processes are perfect.

Exercise 2



The answer is **yes**, because we can implement it on the *process* that has all the *links* of **channel A**. So, in that case when the *timeout* will end if a *process* didn't send a *reply* we know for sure that it has **crashed**.

Uses:
Oracle O_i
Perfect P2P link

Init:
$\text{leader}_i = \perp$
$\text{left}_i = getLeft()$
$\text{right}_i = getRight()$

when $left_i = null$ **do**:
    $\text{leader}_i = p_i$
    trigger $leader(p_i)$
    trigger $send(NewLeader, leader_i)$ to right_i

upon event $deliver(NewLeader, l)$ from left_i
    if $\text{leader}_i \neq l$
        trigger $leader(p_i)$
        trigger $send(NewLeader, leader_i)$ to right_i

upon event $left\_neighbour(p_j)$
    $\text{left}_i = p_j$

upon event $right\_neighbour(p_j)$
    $\text{right}_i = p_j$
    trigger $send(NewLeader, leader_i)$ to right_i

6 Broadcast Communication

6.1 Best Effort Broadcast

We will now focus on the **message broadcasting**. This means that a *process* sends a *message* to all the other ones, and there are several types of *broadcast*, the first one is the **BEB**, or **Best Effort Broadcast**, that ensures *message delivery* only if the *sender* don't crash, if it happens, *processes* may disagree on whether or not *deliver* the *message*.

Module 3.1: Interface and properties of best-effort broadcast

Module:

Name: BestEffortBroadcast, **instance** *beb*.

Events:

Request: ⟨ *beb*, *Broadcast* | m ⟩: Broadcasts a message m to all processes.

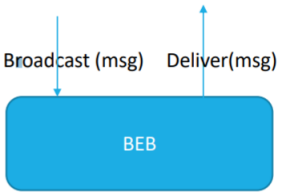
Indication: ⟨ *beb*, *Deliver* | p, m ⟩: Delivers a message m broadcast by process p.

Properties:

BEB1: *Validity:* If a correct process broadcasts a message m, then every correct process eventually delivers m.

BEB2: *No duplication:* No message is delivered more than once.

BEB3: *No creation:* If a process delivers a message m with sender s, then m was previously broadcast by process s.



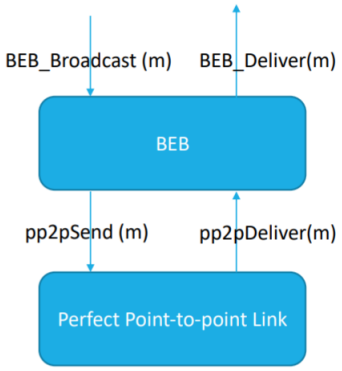
Algorithm 3.1: Basic Broadcast

Implements:
BestEffortBroadcast, **instance** *beb*.

Uses:
PerfectPointToPointLinks, **instance** *pl*.

upon event ⟨ *beb*, *Broadcast* | m ⟩ **do**
  forall $q \in \Pi$ **do**
    trigger ⟨ *pl*, *Send* | q, m ⟩;

upon event ⟨ *pl*, *Deliver* | p, m ⟩ **do**
  trigger ⟨ *beb*, *Deliver* | p, m ⟩;



System model

- Asynchronous system
- perfect links
- crash failures

6.2 Reliable Broadcast

The **Reliable Broadcast** instead is:

Module 3.2: Interface and properties of (regular) reliable broadcast

Module:

Name: ReliableBroadcast, **instance** *rb*.

Events:

Request: ⟨ *rb*, *Broadcast* | m ⟩: Broadcasts a message m to all processes.

Indication: ⟨ *rb*, *Deliver* | p, m ⟩: Delivers a message m broadcast by process p.

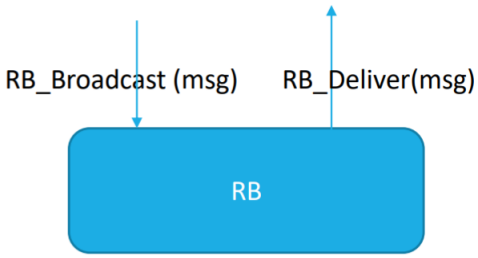
Properties:

RB1: *Validity:* If a correct process p broadcasts a message m, then p eventually delivers m.

RB2: *No duplication:* No message is delivered more than once.

RB3: *No creation:* If a process delivers a message m with sender s, then m was previously broadcast by process s.

RB4: *Agreement:* If a message m is delivered by some correct process, then m is eventually delivered by every correct process.

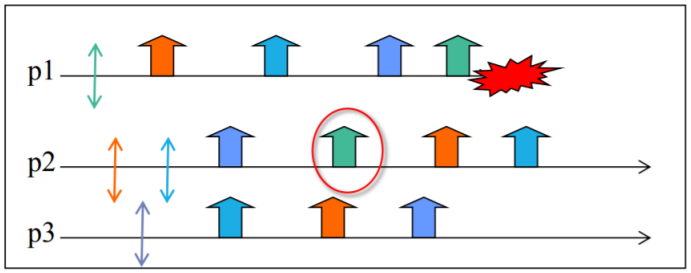


Same as BEB

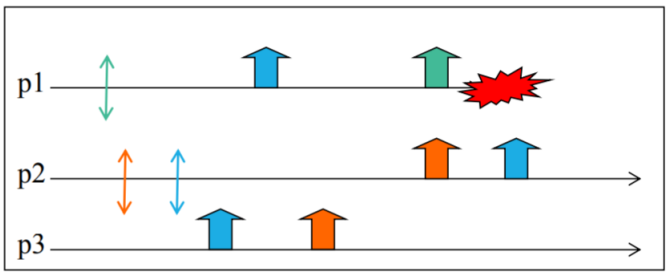
+

Liveness: agreement

In which we have, the same properties of the **BEB**, plus **Agreement** property, now we will see two schemes that hep for the understanding of **BEB** and **RB**:



Satisfies BEB but not RB (violation of the Agreement Property)



Satisfies RB

6.2.1 Reliable Broadcast, Synchronized system

The **Reliable Broadcast** in a **synchronized system** is:

Algorithm 3.2: Lazy Reliable Broadcast

Implements:
ReliableBroadcast, **instance** *rb*.

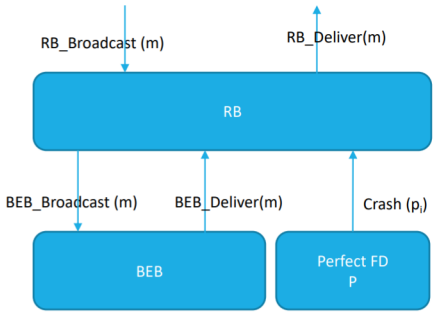
Uses:
BestEffortBroadcast, **instance** *beb*;
PerfectFailureDetector, **instance** $\mathcal{P}$.

upon event ⟨ *rb*, *Init* ⟩ **do**
correct := Π;
from[*p*] := $[\emptyset]^N$;

upon event ⟨ *rb*, *Broadcast* | *m* ⟩ **do**
trigger ⟨ *beb*, *Broadcast* | [DATA, *self*, *m*] ⟩;

upon event ⟨ *beb*, *Deliver* | *p*, [DATA, *s*, *m*] ⟩ **do**
if $m \notin from[s]$ **then**
trigger ⟨ *rb*, *Deliver* | *s*, *m* ⟩;
from[*s*] := *from*[*s*] ∪ {*m*};
if $s \notin correct$ **then**
trigger ⟨ *beb*, *Broadcast* | [DATA, *s*, *m*] ⟩;

upon event ⟨ $\mathcal{P}$, *Crash* | *p* ⟩ **do**
correct := *correct* \ {*p*};
forall $m \in from[p]$ **do**
trigger ⟨ *beb*, *Broadcast* | [DATA, *p*, *m*] ⟩;



The algorithm is Lazy in the sense that it retransmits only when necessary

The first *if* in the *upon event* (*beb*, *deliver*), is used in order to check the presence of *duplicate messages* (if there is *delay* in receiving *messages*) so it guarantees the **no duplication**. So, only the *messages* that are coming from a *crashed process* will be retrasmitted so we ensure the *agreement property*, the *re-broadcast* is done by leaving as *sender* the *original crashed one*. In the *best case* 1 *BEB message* per one *RB message*, so it means that we don't have any crashes in the system and we don't need to re-broadcast, instead in the *worst case* we have $n-1$ BEB messages per one RB, so we have $n-1$ failures, so for each *RB message* we have to *re-broadcast* the *message* $n-1$ times.

6.2.2 Reliable Broadcast, Asynchronous system

If the **failure detector** is not *perfect*, in an *Asynchronous System*, we always have to retransmit the *message* (**eager algorithm**). In this case, the *best case* is the same of the *worst case* so we have n BEB messages per *one* RB *message*:

Algorithm 3.3: Eager Reliable Broadcast

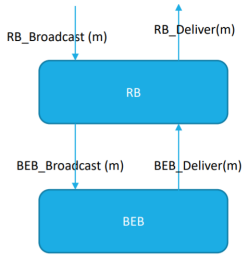
Implements:
ReliableBroadcast, **instance** *rb*.

Uses:
BestEffortBroadcast, **instance** *beb*.

upon event ⟨ *rb*, *Init* ⟩ **do**
delivered := ∅;

upon event ⟨ *rb*, *Broadcast* | *m* ⟩ **do**
trigger ⟨ *beb*, *Broadcast* | [DATA, *self*, *m*] ⟩;

upon event ⟨ *beb*, *Deliver* | *p*, [DATA, *s*, *m*] ⟩ **do**
if $m \notin delivered$ **then**
delivered := *delivered* ∪ {*m*};
trigger ⟨ *rb*, *Deliver* | *s*, *m* ⟩;
trigger ⟨ *beb*, *Broadcast* | [DATA, *s*, *m*] ⟩;



The algorithm is Eager in the sense that it retransmits every message

6.3 Uniform Reliable Broadcast

There is also another type of *Reliable Broadcast*, called **Uniform Reliable Broadcast** or **URB**, where the only difference is that the *Agreement property* becomes **uniform**, which means that *correct processes* must *deliver* also *messages* from *faulty processes*, so the *delivers* of the *crashed processes* are a **subset** of the *delivers* of the *correct processes*:

Module 3.3: Interface and properties of uniform reliable broadcast

Module:

Name: UniformReliableBroadcast, **instance** *urb*.

Events:

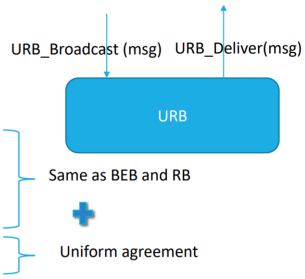
Request: ⟨ *urb*, *Broadcast* | m ⟩: Broadcasts a message m to all processes.

Indication: ⟨ *urb*, *Deliver* | p, m ⟩: Delivers a message m broadcast by process p.

Properties:

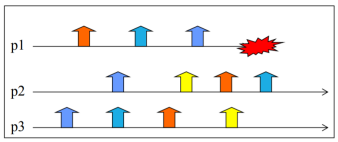
URB1–URB3: Same as properties RB1–RB3 in (regular) reliable broadcast (Module 3.2).

URB4: *Uniform agreement:* If a message m is delivered by some process (whether correct or faulty), then m is eventually delivered by every correct process.

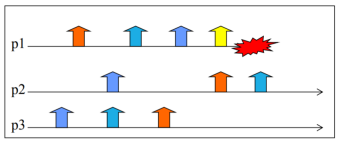


Agreement on a message delivered by any process (crashed or not)!

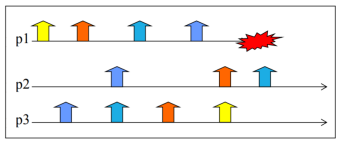
the set of messages delivered by a correct process is a superset of the ones delivered by a faulty one



URB



RB if yellow message is sent by p1
Non-correct otherwise



BEB if yellow message is sent by p1
Non-correct otherwise

6.3.1 Uniform Reliable Broadcast, Synchronous system

Algorithm 3.4: All-Ack Uniform Reliable Broadcast

Implements:
UniformReliableBroadcast, **instance** *urb*.

Uses:
BestEffortBroadcast, **instance** *beb*.
PerfectFailureDetector, **instance** $\mathcal{P}$.

upon event ⟨ *urb*, *Init* ⟩ **do**
delivered := ∅;
pending := ∅;
correct := Π;
forall m **do** *ack*[m] := ∅;

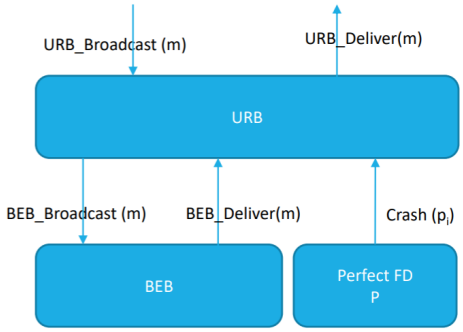
upon event ⟨ *urb*, *Broadcast* | m ⟩ **do**
pending := *pending* ∪ {(*self*, m)};
trigger ⟨ *beb*, *Broadcast* | [DATA, *self*, m] ⟩;

upon event ⟨ *beb*, *Deliver* | p, [DATA, s, m] ⟩ **do**
ack[m] := *ack*[m] ∪ {p};
if $(s, m) \notin$ *pending* **then**
pending := *pending* ∪ {(s, m)};
trigger ⟨ *beb*, *Broadcast* | [DATA, s, m] ⟩;

upon event ⟨ $\mathcal{P}$, *Crash* | p ⟩ **do**
correct := *correct* \ {p};

function *candeliver*(m) **returns** Boolean **is**
return (*correct* ⊆ *ack*[m]);

upon exists $(s, m) \in$ *pending* such that *candeliver*(m) ∧ $m \notin$ *delivered* **do**
delivered := *delivered* ∪ {m};
trigger ⟨ *urb*, *Deliver* | s, m ⟩;



Where *ack* is a *matrix* in which we have for *rows* the *messages* and for *columns* all the *processes* of the system. When the *process* p sends a **BEB messages**, we put in the *pending* a tuple $(id\_sender, id\_message)$. When the *process* receives a message from the *BEB* it insert in the *ack* matrix in the rows of m himself, if the tuple $(id\_sender, id\_message)$ is not in *pending*, the tuple is inserted and the *message* will be **re-broadcasted**. The *candeliver* is a *boolean function* that ensures that the set of the *correct processes* is a subset of the *processes* that receives the *BEB deliver*.

6.3.2 Uniform Reliable Broadcast, Asynchronous system

The *algorithm* is the same of the *synchronous* one but the difference is that we don't have anymore the *perfect failure detector*, so the *candeliver* function is modified, and it return true when the number of *ack received* are at least half of the total. The assumption is that the **majority** of the *processes* are *correct*.

Algorithm 3.5 Majority-Ack Uniform Reliable Broadcast

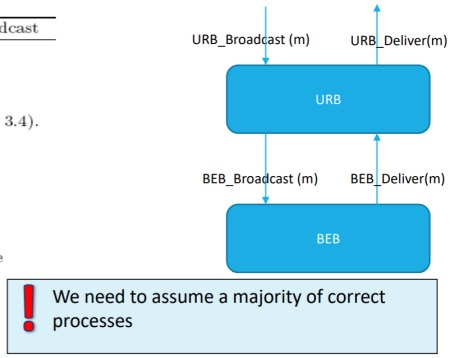
Implements:
UniformReliableBroadcast (urb).

Extends:
All-Ack Uniform Reliable Broadcast (Algorithm 3.4).

Uses:
BestEffortBroadcast (beb).

function canDeliver(m) **returns** boolean **is**
return ($|ack_m| > N/2$);

// Except for the function above, and the non-use of the
// perfect failure detector, same as Algorithm 3.4.



6.4 Probabilistic Broadcast

In the **Probabilistic Broadcast** we have that the message is delivered 99% of the times, so it's not fully reliable. The **probabilistic broadcast** implements a *tree structure* where the *broadcast message* is sent directly to sons, there is also a *hierarchical communication* in which the tree is *hierarchical* and in this case the system loss some *correctness* but in terms of *speed* is more *efficient*.

Module 3.7: Interface and properties of probabilistic broadcast

Module:

Name: ProbabilisticBroadcast, **instance** *pb*.

Events:

Request: $\langle$ *pb*, *Broadcast* | m $\rangle$: Broadcasts a message m to all processes.

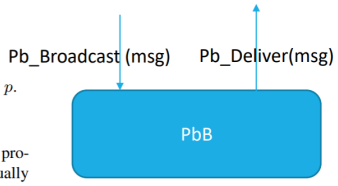
Indication: $\langle$ *pb*, *Deliver* | p, m $\rangle$: Delivers a message m broadcast by process p.

Properties:

PB1: *Probabilistic validity:* There is a positive value ε such that when a correct process broadcasts a message m, the probability that every correct process eventually delivers m is at least $1 - \varepsilon$.

PB2: *No duplication:* No message is delivered more than once.

PB3: *No creation:* If a process delivers a message m with sender s, then m was previously broadcast by process s.



6.4.1 Eager Probabilistic Broadcast

This **broadcast** is used when we work on huge *distributed system*. In fact if we have 100 *nodes*, we could need 100^2 or 100^3 *messages* in the worst case for a *single message delivery*. Instead with this *system* based on the **Gossip Dissemination**, in which a *process* sends a *message* to a set of *random process* and the *processes* that receive will send the *message* to another set of *random process* and this happen for r *rounds*, we cover almost all the *nodes* with a *cost* much lower than the previous.

Algorithm 3.9: Eager Probabilistic Broadcast

Implements:
ProbabilisticBroadcast, **instance** *pb*.

Uses:
FairLossPointToPointLinks, **instance** *fll*.

upon event ⟨ *pb*, *Init* ⟩ **do**
  delivered := ∅;

procedure *gossip*(*msg*) **is**
  forall $t \in$ *picktargets*(k) **do trigger** ⟨ *fll*, *Send* | t, *msg* ⟩;

function *picktargets*(k) **returns** set of processes **is**
  targets := ∅;
  while #(*targets*) < k **do**
    candidate := random($\Pi \setminus$ {*self*});
    if *candidate* ∉ *targets* **then**
      targets := *targets* ∪ {*candidate*};
  return *targets*;

upon event ⟨ *pb*, *Broadcast* | m ⟩ **do**
  delivered := *delivered* ∪ {m};
  trigger ⟨ *pb*, *Deliver* | *self*, m ⟩;
  gossip([GOSSIP, *self*, m, R]);

upon event ⟨ *fll*, *Deliver* | p, [GOSSIP, s, m, r] ⟩ **do**
  if $m \notin$ *delivered* **then**
    delivered := *delivered* ∪ {m};
    trigger ⟨ *pb*, *Deliver* | s, m ⟩;
  if $r > 1$ **then** *gossip*([GOSSIP, s, m, $r-1$]);

The *picktargets* functions picks $k-$ *random processes* from the entire set of the *processes*, the *gossip function* instead is used to send a message in *broadcast* to a subset of *processes* for a number of *rounds*.

7 Consensus

7.1 Regular Consensus

A group of *processes* must agree in a *value* proposed by one of them, they start with different opinions and then they **converge** toward only one of them.

Module 5.1: Interface and properties of (regular) consensus

Module:

Name: Consensus, **instance** c.

Events:

Request: ⟨ c, *Propose* | v ⟩: Proposes value v for consensus.

Indication: ⟨ c, *Decide* | v ⟩: Outputs a decided value v of consensus.

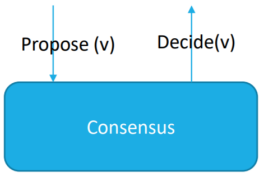
Properties:

C1: *Termination:* Every correct process eventually decides some value.

C2: *Validity:* If a process decides v, then v was proposed by some process.

C3: *Integrity:* No process decides twice.

C4: *Agreement:* No two correct processes decide differently.



We don't deal with **asynchronous systems** cause no *algorithm* can guarantee to reach *consensus* even with one *process crash failure*. In the case of a **synchronous system**, we can implement the **Flooding Consensus**, in which *processes* exchange their values and when all the *processes* make their own proposal available, a value is chosen, but in order to do this we need no *failures* due to the *communication*.

Algorithm 5.1: Flooding Consensus

Implements:
Consensus, **instance** c.

Uses:
BestEffortBroadcast, **instance** *beb*;
PerfectFailureDetector, **instance** $\mathcal{P}$.

upon event $\langle$ c, *Init* $\rangle$ **do**
correct := Π;
round := 1;
decision := $\bot$;
receivedfrom := $[\emptyset]^N$;
proposals := $[\emptyset]^N$;
receivedfrom[0] := Π;

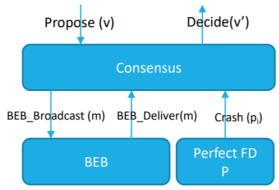
upon event $\langle$ $\mathcal{P}$, *Crash* | p $\rangle$ **do**
correct := *correct* $\setminus$ $\{p\}$;

upon event $\langle$ c, *Propose* | v $\rangle$ **do**
proposals[1] := *proposals*[1] $\cup$ $\{v\}$;
trigger $\langle$ *beb*, Broadcast | [PROPOSAL, 1, *proposals*[1]] $\rangle$;

upon event $\langle$ *beb*, *Deliver* | p, [PROPOSAL, r, ps] $\rangle$ **do**
receivedfrom[r] := *receivedfrom*[r] $\cup$ $\{p\}$;
proposals[r] := *proposals*[r] $\cup$ ps;

upon *correct* $\subseteq$ *receivedfrom*[*round*] $\wedge$ *decision* = $\bot$ **do**
if *receivedfrom*[*round*] = *receivedfrom*[*round* $-$ 1] **then**
decision := min(*proposals*[*round*]);
trigger $\langle$ *beb*, Broadcast | [DECIDED, *decision*] $\rangle$;
trigger $\langle$ c, Decide | *decision* $\rangle$;
else
round := *round* + 1;
trigger $\langle$ *beb*, Broadcast | [PROPOSAL, *round*, *proposals*[*round* $-$ 1]] $\rangle$;

upon event $\langle$ *beb*, *Deliver* | p, [DECIDED, v] $\rangle$ **such that** $p \in$ *correct* $\wedge$ *decision* = $\bot$ **do**
decision := v;
trigger $\langle$ *beb*, Broadcast | [DECIDED, *decision*] $\rangle$;
trigger $\langle$ c, Decide | *decision* $\rangle$;



We can see that **receivedfrom** is an *array* where we insert in the i^{th} position the *processes* from which I delivered in the i^{th} round, instead **proposal** is an array of n entry where in the i^{th} position i will put all the *proposals received* in the *last round* (even my *proposal*). The **propose event** permit to handle my *proposal* by adding it to the array of the *proposal*, and then he send this to all the other thanks to BEB. When we check the **correct array** we are checking if the set of *received proposals* of this *round* is equal to the *received proposals* of the *last round* (so we have only *correct processes*) and if still we don't have decided a variable the *process* will decide it and will send it to all the other thanks to *BEB*. **Last event** handle the situation in which the *variable* was decided by *other process*, in this case I check if the *process* that taken the decision is *alive* and I set the *decided variable* and I *rebroadcast* the decision with *BEB*.

- **Correctness**:
  - **Validity** and **Integrity** follow from the properties on the *communication channels*;
  - **Termination** is ensured because *algorithm* terminates at most after N *rounds*;
  - **Agreement** is satisfied cause the same *deterministic function* is applied to the *same values* by *correct processes*;
- **Performance**:
  - **Best case**: one *communication round*, so $2 \times N^2$;
  - **Worst case**: we have N^2 messages exchanged for N rounds so we have N^3 messages;

7.2 Uniform Consensus

In the **Uniform Consensus** we have the **Uniform Agreement** property which means that also *faulty processes* agree for the decided value:

Module 5.2: Interface and properties of uniform consensus

Module:

Name: UniformConsensus, **instance** *uc*.

Events:

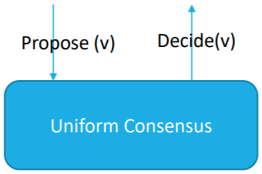
Request: ⟨ *uc*, *Propose* | *v* ⟩: Proposes value *v* for consensus.

Indication: ⟨ *uc*, *Decide* | *v* ⟩: Outputs a decided value *v* of consensus.

Properties:

UC1–UC3: Same as properties C1–C3 in (regular) consensus (Module 5.1).

UC4: *Uniform agreement:* No two processes decide differently.



In this case instead we will check only the **proposal** from the *current round*, so it's very similar to the previous one but here the *decision* is based only on the *current round*. If we are not in the *last round*, we increment the *round* variable and we reset the **receivedfrom array** that is the set that contains the *processes* from which I received the *proposal*.

Algorithm 5.3: Flooding Uniform Consensus

Implements:
UniformConsensus, **instance** *uc*.

Uses:
BestEffortBroadcast, **instance** *beb*;
PerfectFailureDetector, **instance** $\mathcal{P}$.

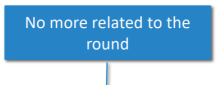
upon event ⟨ *uc*, *Init* ⟩ **do**
correct := *Π*;
round := 1;
decision := ⊥;
proposalset := ∅;
receivedfrom := ∅;

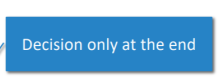
upon event ⟨ $\mathcal{P}$, *Crash* | *p* ⟩ **do**
correct := *correct* \ {*p*};

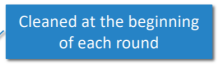
upon event ⟨ *uc*, *Propose* | *v* ⟩ **do**
proposalset := *proposalset* ∪ {*v*};
trigger ⟨ *beb*, *Broadcast* | [PROPOSAL, 1, *proposalset*] ⟩;

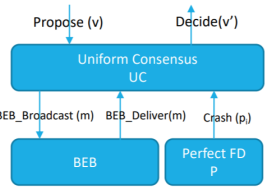
upon event ⟨ *beb*, *Deliver* | *p*, [PROPOSAL, *r*, *ps*] ⟩ **such that** *r* = *round* **do**
receivedfrom := *receivedfrom* ∪ {*p*};
proposalset := *proposalset* ∪ *ps*;

upon *correct* ⊆ *receivedfrom* ∧ *decision* = ⊥ **do**
if *round* = *N* **then**
decision := min(*proposalset*);
trigger ⟨ *uc*, *Decide* | *decision* ⟩;
else
round := *round* + 1;
receivedfrom := ∅;
trigger ⟨ *beb*, *Broadcast* | [PROPOSAL, *round*, *proposalset*] ⟩;









- **Correctness**:
  - **Validity** and **Integrity** follow from the properties of the *best-effort broadcast*;
  - **Termination** is ensured because all *correct processes* reach *round N* and decide in that *round*;
    - The **strong completeness** property of the *failure detector* implies that no *correct process* waits indefinitely for a *message* from a *process* that has crashed, as the *crashed process* is eventually removed from *correct*;
  - **Uniform Agreement** holds cause all *processes* that reach *round N* have the same set of values in their variable *proposalset*;

- **Performance**:
  - We have N *communication steps* and $O(N^3)$ *messages* for all *correct process* to **decide**;

8 Paxos

The **Paxos algorithms** was introduced in order to provide a viable solution to *consensus* in **asynchronous system**, with these **Safety** is always guaranteed, but the *algorithm* makes some progress (**Liveness**) only when the network works for enough time (*partial synchronized*). We have two basic *assumptions*:

- **Agents** can fail by *stopping*, the also operate at *arbitrary speed* and they may *restart*;
  - Since all *agents* may fail after a value is *chosen* and then *restart*, a solution is impossible unless some **information** can be **remembered** by an *agent* that has *failed* and *restarted*;
- **Messages** can take arbitrarily long time to be *delivered*, can be also be *duplicated* or *lost*, but they aren't **corrupted**;

There are three **actors** in **Paxos protocol**:

- **Proposer**: who *propose* a *value*;
- **Acceptors**: *processes* that *commits* on a *final decided value*;
- **Learners**: who *passively assist* to the *decision* and they obtain the *final decided value*;

A model with only one **acceptor** is the simplest one, but we have a problem if it *crash*, so we must have **multiple acceptors**, and in this case a value is accepted when the **majority** of it accepts it.

The problem is that each *acceptor* may receive a different set of *proposals*, a possible solution is that:

- An **acceptor** may accept at most one *value*;

But in this case which value the *acceptor* should accept? A possible solution is that:

- An **acceptor** must accept the first *proposal* it receives;

But in this case we can have a sort of *deadlock* in which the *acceptors* couldn't reach a *majority*. We have to keep track of the different *proposal* by assigning a value v unique, and then the value is chosen when a *proposal* with the same value has been *accepted* by the *majority*.

- If a *proposal* with value v is accepted every *high-numbered proposal* that is accepted by any acceptor has value v;

But what if a new proposal *propose* a new different value that the *acceptor* must accept?

- If a *proposal* with value v is chosen, every *high-numbered proposal* issued by any *proposer* has value v;

Now let's assume that a *proposal* m with value v has been *accepted*, now we have to guarantee that any *proposal* $n > m$ has value v, we could prove it by **induction** assuming that every proposal with number in $[m, n-1]$ has value v. For m to be accepted there is a *majority* of *acceptors* that accept it. Therefore the assumption that m has been accepted implies that: every *acceptor* in the *majority* has accepted a *proposal* with number in $[m, n-1]$ with value v.

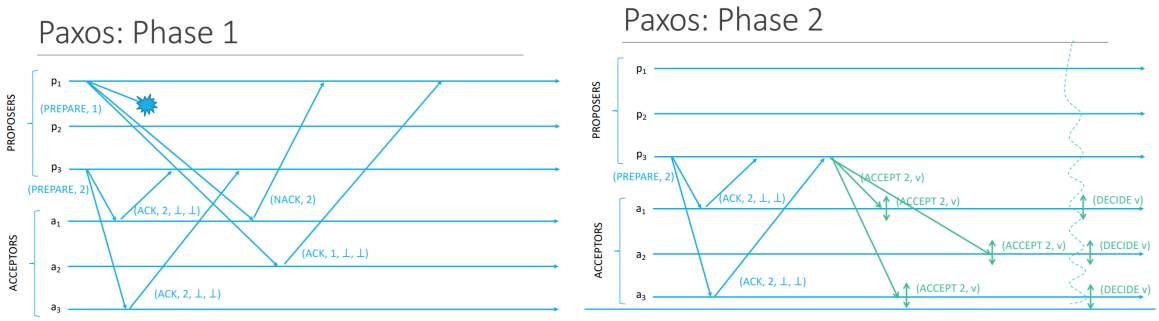
- For any v and n, if a **proposal** with value v and number n is **issued**, then there is a set S consisting of a *majority* of acceptors such that either:
  - No **acceptor** in S has accepted any *proposal* numbered less than n;
  - v is the value of the **highest-numbered proposal** among all proposals numbered less than n accepted by the *acceptors* in S;

So this *condition* consider the situation in which a set of *acceptors* S accept a proposal n with value v, and this can happened in two cases: in the first case, all the previous *proposals* with $id < n$ weren't *accepted*, in the second case, a *proposal* $n^{'} < n$ was already *accepted*, but v is equal to the proposal value $v^{'}$ of $n^{'}$.

To ensure this, we have to ask to *proposer* that wants to *propose* a value numbered n to learn the *highest-numbered value* with number less than n that has been or will be accepted, by any *acceptor* in a *majority*. To learn about a *proposal* we simply have to ask to the *acceptors* to not accept any value numbered less than n. The **Paxos protocol** has two main phases:

- **Phase 1**:
  - A **proposer** chooses a new *proposal* version number n and sends a **prepare request** $(PREPARE, n)$ to a *majority* of *acceptors*;
  - If an **acceptor** receives a *prepare request* it respond with a *promise* not to accept any more proposal numbered less than n and he suggest the value $v^{'}$ of the **highest-numbered proposal** that it has accepted if there is any, else $\perp$:
    $(ACK, n, n^{'}, v^{'})$ if it exists;
    $(ACK, n, \perp, \perp)$ if not;
  - If an *acceptor* receive a **prepare request** with a n lower than the $n^{'}$ from any *prepare request* it has already responded sends out a $(NACK, n^{'})$;
- **Phase 2**:
  - If the *proposer* receives **ACKs** from a majority of *acceptors* then it can issue an **accept request** $(ACCEPT, n, v)$ where n is the number that appears in the *prepare request*, and v is the value of the *highest-numbered proposal* among the responses or the proposal's own proposal if none was received;

– If the *acceptor* receives an **accept request**, it accepts the *proposal* unless it has already responded to a *prepare request* with a number greater than n;
– Whenever *acceptor* **accepts** a *proposal* respond to all *proposal* $(ACCEPT, n, v)$, and the *proposal* that receives $(ACCEPT, n, v)$ from a majority of *acceptors*, decides v and sends a $(DECIDE, v)$ to all the other *learners*. All the *learners* that receive $(DECIDE, v)$, decide v;



9 Ordered Communications

Here we need to define guarantees about the order of deliveries inside group of processes. We have three different types of ordering:

- Delivers respect **FIFO ordering** of the corresponding send;
- Delivers respect **Casual ordering** of the corresponding send;
- Delivery respects a **Total ordering** of deliveries;

Reliable broadcast that we previously studied doesn't have any property on ordering deliveries of messages and this can cause problems in the same communication.

9.1 FIFO broadcast

FIFO Broadcast can be uniform/non uniform

Module 3.8: Interface and properties of FIFO-order (reliable) broadcast

Module:

Name: FIFOReliableBroadcast, **instance** *frb*.

Events:

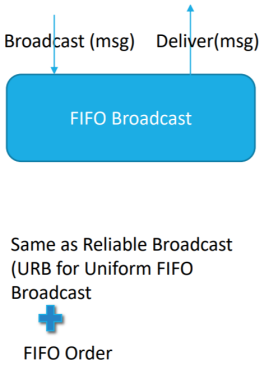
Request: ⟨ *frb*, *Broadcast* | m ⟩: Broadcasts a message m to all processes.

Indication: ⟨ *frb*, *Deliver* | p, m ⟩: Delivers a message m broadcast by process p.

Properties:

FRB1–FRB4: Same as properties RB1–RB4 in (regular) reliable broadcast (Module 3.2).

FRB5: *FIFO delivery:* If some process broadcasts message m_1 before it broadcasts message m_2, then no correct process delivers m_2 unless it has already delivered m_1.



In the *upon event Deliver* there is a **while** next that is used when we receive a *message* with *sn* (*identifier* of the *message* used to control the *order*) equal to the current one, cause *next* is an array used to track how many *messages* are arrived to that *process*. In the **while** we will empty the *pending array* when there are *message* with a *sn* less that the received one, so in this way we respect the **FIFO property**.

Algorithm 3.12: Broadcast with Sequence Number

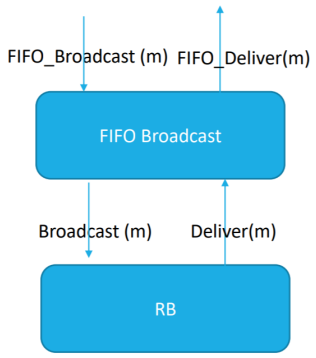
Implements:
FIFOReliableBroadcast, **instance** *frb*.

Uses:
ReliableBroadcast, **instance** *rb*.

upon event ⟨ *frb*, *Init* ⟩ **do**
lsn := 0;
pending := ∅;
next := $[1]^N$;

upon event ⟨ *frb*, *Broadcast* | m ⟩ **do**
lsn := *lsn* + 1;
trigger ⟨ *rb*, *Broadcast* | [DATA, *self*, m, *lsn*] ⟩;

upon event ⟨ *rb*, *Deliver* | p, [DATA, s, m, sn] ⟩ **do**
pending := *pending* ∪ {(s, m, sn)};
while exists $(s, m', sn') \in$ *pending* such that $sn' = next[s]$ **do**
$next[s] := next[s] + 1$;
pending := *pending* \ {(s, m', sn')};
trigger ⟨ *frb*, *Deliver* | s, m' ⟩;



9.2 Casual Order Broadcast

The **Casual Order Broadcast** ensures that *messages* are *delivered* such that they respect all **cause-effect relations**, so is an extension of the *happened-before relation*, so there can be a *message* m_1 that cause a *message* m_2, denoted as: $m_1 \rightarrow m_2$, and this happens when:

- Some *process* p **broadcast** m_1 before it *broadcast* m_2;
- Some *process* p **delivers** m_1 and subsequently *broadcast* m_2;
- There exists some **message** $m^{'}$ such that $m_1 \rightarrow m^{'}$ and $m^{'} \rightarrow m_2$

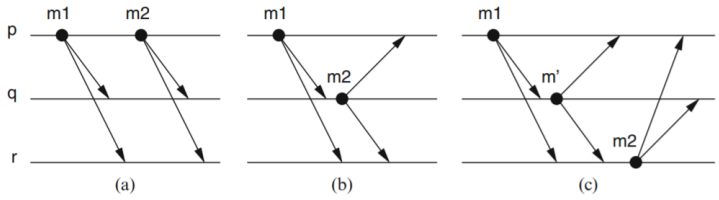


Figure 3.8: Causal order of messages

It's important to note that $Causal\ Broadcast = Reliable\ Broadcast + Causal\ Order$ and that $Causal\ Order = FIFO\ Order + Local\ Order$, where **Local Order** means that if a *process* delivers a *message* m before sending a *message* $m^{'}$, then no *correct process* deliver $m^{'}$ if it has not already delivered m.

Module 3.9: Interface and properties of causal-order (reliable) broadcast

Module:

Name: CausalOrderReliableBroadcast, **instance** *crb*.

Events:

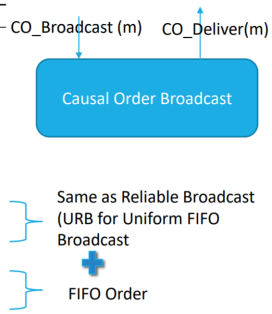
Request: ⟨ *crb*, *Broadcast* | m ⟩: Broadcasts a message m to all processes.

Indication: ⟨ *crb*, *Deliver* | p, m ⟩: Delivers a message m broadcast by process p.

Properties:

CRB1–CRB4: Same as properties RB1–RB4 in (regular) reliable broadcast (Module 3.2).

CRB5: *Causal delivery:* For any message m_1 that potentially caused a message m_2, i.e., $m_1 \rightarrow m_2$, no process delivers m_2 unless it has already delivered m_1.



9.2.1 Waiting Causal Broadcast

Algorithm 3.15: Waiting Causal Broadcast

Implements:
CausalOrderReliableBroadcast, **instance** *crb*.

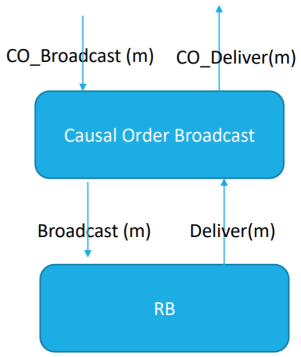
Uses:
ReliableBroadcast, **instance** *rb*.

upon event ⟨ *crb*, *Init* ⟩ **do**
$V := [0]^N$;
$lsn := 0$;
$pending := \emptyset$;

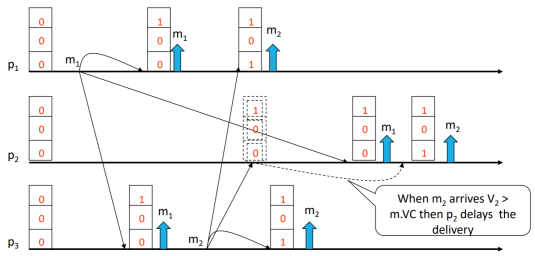
upon event ⟨ *crb*, *Broadcast* | m ⟩ **do**
$W := V$;
$W[rank(self)] := lsn$;
$lsn := lsn + 1$;
trigger ⟨ *rb*, *Broadcast* | [DATA, W, m] ⟩;

The function rank() associates an entry of the vector to each process

upon event ⟨ *rb*, *Deliver* | p, [DATA, W, m] ⟩ **do**
$pending := pending \cup \{(p, W, m)\}$;
while exists $(p', W', m') \in pending$ such that $W' \leq V$ **do**
$pending := pending \setminus \{(p', W', m')\}$;
$V[rank(p')] := V[rank(p')] + 1$;
trigger ⟨ *crb*, *Deliver* | p', m' ⟩;



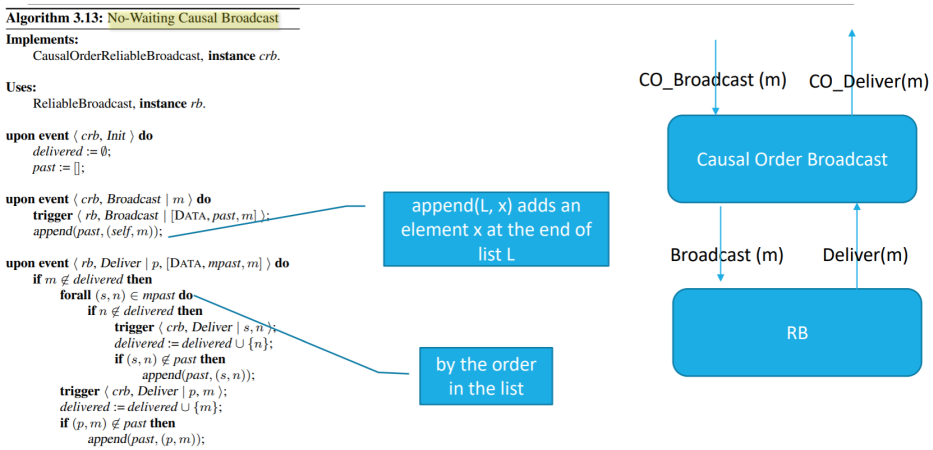
V is the **logical vector**, a vector of dimension N, the number of *processes*. In the **Broadcast event**, I copy the current *logical vector* and in the position of the current *process* I insert the *lsn* before this is incremented. In the **Deliver event** instead, I will enter in the while only if inside the set of *pending messages* there are *messages* with a *logical vector* $W^{'}$ lower than my *logical vector*, so in this way I can deliver them and I respect the condition of the **causal order**. We also in the *Deliver event* increment the **logical clock value** of that *process* cause for the *RB property* a *message* that I am going to deliver was previously sent by a *sender process* so we will increment the value of that *process*.



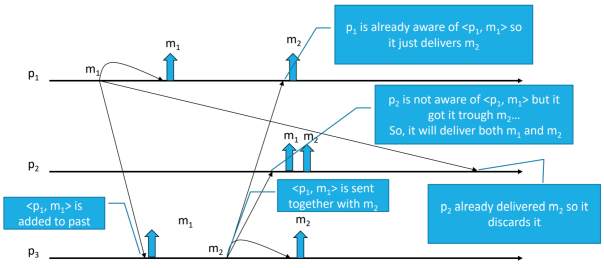
Safety: let two *broadcast messages* m and $m^{'}$ such that $broadcast(m) \rightarrow broadcast(m^{'})$ then each *process* have to deliver m before $m^{'}$
Liveness: eventually each *message* will be delivered and is guaranteed by two assumptions:

- The number of *broadcast events* that precedes a certain *event* is finite;
- *Channels* are reliable;

9.2.2 Non-Waiting Causal Broadcast



The approach of this *algorithm* is **continuous** in fact each time a *message* is *delivered*, the *process* doesn't wait the *missing messages*, so it is always delivered once the *process* is sure that the **past messages** of the received one are *delivered* and then added to my list of *past messages*. In fact the *past* list is a list in which will be inserted all the *message* involved in actions of *deliver* or *broadcast* (by respecting a *causal order*). In the **broadcast event** we will insert in the *past list* the message that will be *broadcasted*. In the **deliver event** instead we will check all the *past list* of the *message* received and for each *message* extracted from it will be checked if *current process* has already *delivered* it and if not this will be *delivered* and inserted in the current *past list*, at the end the *current message*, if is not in the my *past messages*, is *delivered* and inserted in the list.



10 Total Order Broadcast

A **Total Order Broadcast** is a *reliable broadcast* that orders all *messages*, even those from different *senders* and those that are not *causally related*. *Total Order Broadcast* = *Reliable Broadcast* + *Total Order*, from the *reliable* we have that *processes* agree on the same set of *messages* they *deliver*, and from the *total order*, *processes* agree on the same *sequence* of *message*. The *message* is delivered to all or to none of the *processes* and, if the *message* is *delivered*, every other *message* is **ordered** either before or after this *message*.

It's important to note that *Total Order* is **orthogonal** with respect to *FIFO* and *Causal Order*. This means that respecting the *total order* doesn't mean that *FIFO* and *causal order* are respected too, in fact these two are *parallel*, if the *causal order* is respect also *FIFO* is respected, instead with *total order* we cannot make any assumption on *causal* and *FIFO*.

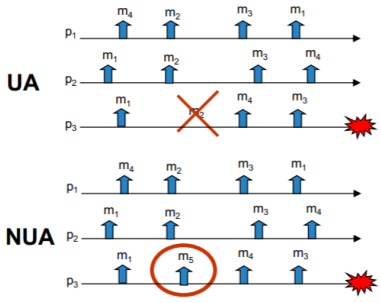
In order to study this part, we need to consider a *system model* composed by a static set of *processes* with *perfect communication channels*, *asynchronous* and *crash-fault based* and we characterize the system in terms of its possible **runs** R. **Total order specifications** are usually composed by four properties:

- A **Validity property** guarantees that *messages* sent by *correct processes* will eventually be *delivered* at least by *correct processes*;
- An **Integrity property** guarantees that no spurious or *duplicate messages* are *delivered*;
- An **Agreement property** ensures that *processes* deliver the same set of *messages*;
- An **Order property** constrains *processes* delivering the same *messages* to deliver them in the same order;

The **total order specifications** with *crash failure* and *perfect channel* are:

- **NUV**: if a *correct process* $TOCAST$ a *message* m then some *correct process* will **eventually deliver** m;
- **UI**: for any *message* m, every *process* -p **delivers** m at most once and only if m was previously $TOCAST$ by some *process*;

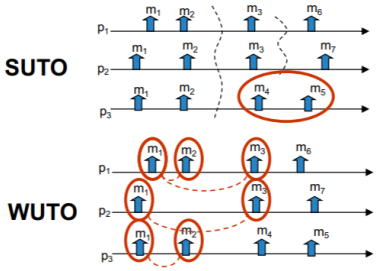
The **Agreement property**:



- **UNIFORM AGREEMENT (UA)**:
  - If a *process* (*correct* or not) $TODelivers$ a message m, then all *correct processes* will eventually $TODeliver$ m;
- **NON-UNIFORM AGREEMENT (NUA)**:
  - If a *correct process* $TODelivers$ a message m, then all *correct processes* will eventually $TODeliver$ m;

So the constrain for **Uniform Agreement** is that *correct processes* always *deliver* the same set of *messages*, and that the set of *messages delivered* by a *faulty process* is a subset of the set of the *correct processes*, instead in *NUA* the set of *faulty* can be completely different.

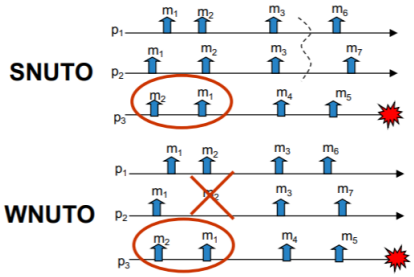
The **Ordering Property** for **Uniform**:



- **STRONG UNIFORM TOTAL ORDER (SUTO)**
  - If some *process* $TODelivers$ some message m before $m^{'}$, then a process $TODelivers$ $m^{'}$ only after it has $TODelivered$ m;
- **WEAK UNIFORM TOTAL ORDER (WUTO)**
  - If process p and process q both $TODeliver$ messages m and $m^{'}$, then p $TODeliver$ m before $m^{'}$ if and only if q $TODeliver$ m before $m^{'}$;

So **SUTO** says that *processes* have the same prefix of the set of *delivered messages* and after an omission (a *message* not *delivered* by someone) we have a *disjointed set* of *delivered messages* (like p_3 in the example image). Instead in **WUTO** there aren't restriction, so the only thing that matter is the *order* of the *deliver* between *processes*.

We have the same property for **Non-Uniform**:



- **STRONG NON-UNIFORM TOTAL ORDER (SNUTO)**
  - If some *correct process* $TODelivers$ some message m before $m^{'}$, then a *correct process* $TODelivers$ $m^{'}$ only after it has $TODelivered$ m;
- **WEAK NON-UNIFORM TOTAL ORDER (WNUTO)**
  - If *correct process* p and q both $TODeliver$ messages m and $m^{'}$, then p $TODeliver$ m before $m^{'}$ if and only if q $TODeliver$ m before $m^{'}$;

10.1 Total Order Algorithm

Algorithm 6.1: Consensus-Based Total-Order Broadcast

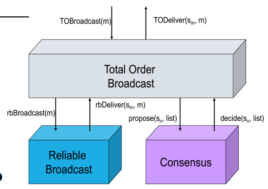
Implements:
TotalOrderBroadcast, **instance** *tob*.

Uses:
ReliableBroadcast, **instance** *rb*;
Consensus (multiple instances).

upon event ⟨ *tob*, *Init* ⟩ **do**
unordered := ∅;
delivered := ∅;
round := 1;
wait := FALSE;

upon event ⟨ *tob*, *Broadcast* | *m* ⟩ **do**
trigger ⟨ *rb*, *Broadcast* | *m* ⟩;

upon event ⟨ *rb*, *Deliver* | *p*, *m* ⟩ **do**
if $m \notin delivered$ **then**
unordered := *unordered* ∪ {(*p*, *m*)};



upon *unordered* ≠ ∅ ∧ *wait* = FALSE **do**
wait := TRUE;
Initialize a new instance *c.round* of consensus;
trigger ⟨ *c.round*, *Propose* | *unordered* ⟩;

upon event ⟨ *c.r*, *Decide* | *decided* ⟩ **such that** *r* = *round* **do**
forall (*s*, *m*) ∈ sort(*decided*) **do**
trigger ⟨ *tob*, *Deliver* | *s*, *m* ⟩;
delivered := *delivered* ∪ *decided*;
unordered := *unordered* \ *decided*;
round := *round* + 1;
wait := FALSE;

So when the list of *messages* is not empty and the *process* is not waiting any decision from the *consensus* it will send his list of **messages unordered**. When it receive a **decision** from the *consensus* it will *deliver* all the *messages* by the *decided order*. It's important to note that the *process* will check if the *consensus round* and his *round* are equal in order to be sure that the *decision* is taken about the actual situation.

10.1.1 UC and URB

When we use **Uniform Consensus** and **Uniform Reliable Broadcast** we obtain a **Total Order**: $TO(UA, SUTO)$:

- Due to **URB** all *processes* (even *faulty*) *delivers* the same set of *messages*, so we obtain UA;
- Due to **UC** all *processes* (even *faulty*) *decide* the same list of *messages*, so message are sorted by a *deterministic rule* and we will have the *same order*;

10.1.2 UC and NURB

When we use **Uniform Consensus** and **Non-Uniform Reliable Broadcast** we obtain a **Total Order**: $TO(NUA, SUTO)$:

- Due to **NURB** all *processes delivers* the same set of *messages*,instead *faulty* can delivers other *messages*, so we obtain NUA;
- Due to **UC** all *processes* (even *faulty*) *decide* the same list of *messages*, so message are sorted by a *deterministic rule* and we will have the *same order*;

10.1.3 NUC and URB

When we use **Not-Uniform Consensus** and **Uniform Reliable Broadcast** we obtain a **Total Order**: $TO(UA, WNUTO)$:

- Due to **URB** all *processes* (even *faulty*) *delivers* the same set of *messages*, so we obtain UA;
- Due to **NUC** all *correct processes decide* the same list of *messages*, so *correct process* will *deliver messages* in the same order, instead *faulty process* will *deliver* (before crash) a different sequence of *messages*.

10.1.4 NUC and NURB

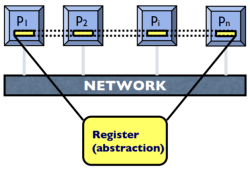
When we use **Not-Uniform Consensus** and **Not-Uniform Reliable Broadcast** we obtain a **Total Order**: $TO(NUA, WNUTO)$:

- Due to **NURB** all *correct processes delivers* the same set of *messages*,instead *faulty* can delivers other *messages*, so we obtain NUA;
- Due to **NUC** all *correct processes decide* the same list of *messages*, so *correct process* will *deliver messages* in the same order, instead *faulty process* will *deliver* (before crash) a different sequence of *messages*.

Consensus / Reliable Broadcast	Uniform	Non Uniform
Uniform	UA SUTO	UA WNUTO
Non Uniform	NUA SUTO	NUA WNUTO

11 Distributed Registers

A **register** is a *shared variable* accessed by *processes* through **read** and **write** *operations*. This abstraction supports the design of *distributed solution* by hiding the complexity of the *message passing system* and the *distribution of the data*.



We have two **operations**:

- **Read operation**: $read() \rightarrow v$, that returns the current value v of the *register*;
- **Write operation**: $write(v)$, that write the value v in the *register* and returns *true* at the end of the operation;

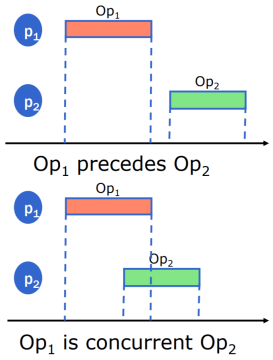
There are three basic assumptions:

- A *register* stores only **positive integers** and it's initialized to 0;

- Each *value* is **univocally identified**;
- *Processes* are **sequential**, so a *process* can invoke only one *operation* per time;

The **notation** of the register is: (X, Y) where X *processes* can **write** and Y *processes* can **read**, so for example $(1, 1)$ is a *register* in which only a *process* can *write* and only a *process* can *read* (these *processes* are decided a priori).

Every *operation* is characterized by two *events*: **Invocation** and **Return**, and each of these *events* occur at a single indivisible point of time. An *operation* is **complete** if both the *invocation* and the *return* events are occured, instead is **failed** if the process *crash* before obtaining a *return*. Given two *operations* o and o', we says that o **precedes** o' if the *response event* of o precedes the *invocation event* of o'. If is not possible to define a *precedence relation* between two operations they are said to be **concurrent**.



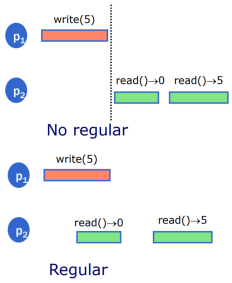
The sequential specification are two:

- **Liveness**: each operation eventually terminates;
- **Safety**: each read operation returns the last value written;

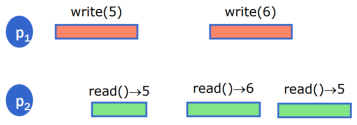
11.0.1 Regular Register

A **regular register** is a *register* $(1, N)$ in which the two following *properties* holds:

- **Termination**: if a *correct process* invokes an *operation*, then the *operation* eventually receives the *confirmation*;
- **Validity**: a *read operation* return the last value *written* or the value *concurrently written*.



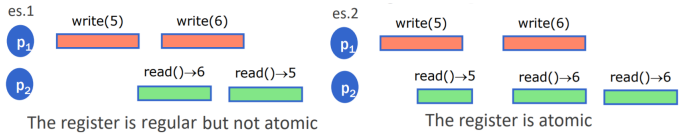
It's important to note that in a *regular register* a *process* can *read* a value v and then a value v' even if the *writer* has written v' and then v, as long as the the *write* and the *read operations* are **concurrent**, but this is not allowed in an *Atomic register*:



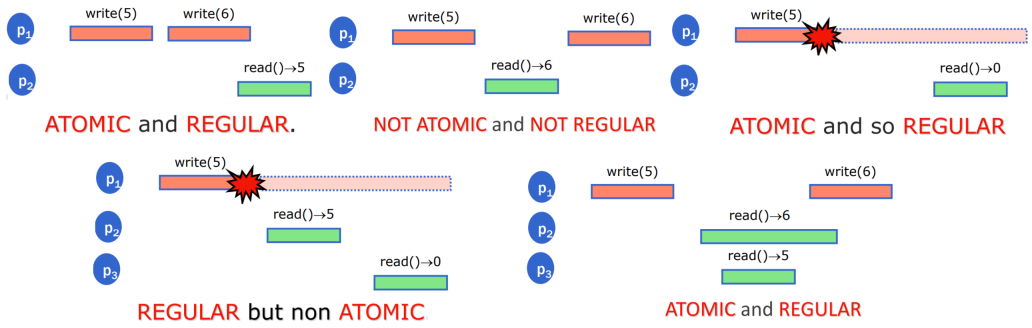
11.0.2 Atomic Register

The **Atomic Register** is a *regular register* with an *ordering property* (that is valid also for *read operations* of different *processes*):

- **Ordering**: if a read return v_2 after a read that it precedes it has returned v_1 then v_1 cannot be written after v_2;



Some examples:



11.1 Regular Register Interface

Let's move to the different types of *implementations* of the *register*, let's begin with the regular register $(1, N)$ that is built in this way:

Module 4.1: Interface and properties of a $(1, N)$ regular register

Module:

Name: $(1, N)$-RegularRegister, **instance** *onrr*.

Events:

Request: $\langle$ *onrr*, *Read* $\rangle$: Invokes a read operation on the register.

Request: $\langle$ *onrr*, *Write* | v $\rangle$: Invokes a write operation with value v on the register.

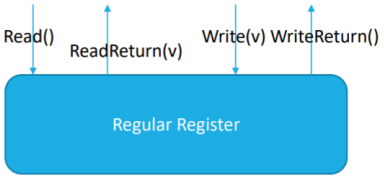
Indication: $\langle$ *onrr*, *ReadReturn* | v $\rangle$: Completes a read operation on the register with return value v.

Indication: $\langle$ *onrr*, *WriteReturn* $\rangle$: Completes a write operation on the register.

Properties:

ONRR1: *Termination:* If a correct process invokes an operation, then the operation eventually completes.

ONRR2: *Validity:* A read that is not concurrent with a write returns the last value written; a read that is concurrent with a write returns the last value written or the value concurrently written.



11.1.1 Read-One-Write-All Algorithm

We will use the **Fail-Stop algorithm: Read-One-Write-All** in which *processes* can *crash* but the *crashes* can be reliably detected by all the other *processes* with the use of a *perfect failure detector*, and it uses a *perfect point-to-point link* and a *Best-effort broadcast (BEB)*. The algorithm idea is that each process stores a *local copy* of the *register* where:

- **Read-one**: where each *read operation* returns the value stored in its local copy of the *register*;
- **Write-all**: where each *write operation* updates the value locally stored at each *process* the *writer* consider to haven't crashed, and a write completes when the *writer* receives an *ack* from each *process* that has not *crashed*;

Algorithm 4.1: Read-One Write-All

Implements:
(1, N)-RegularRegister, **instance** *onrr*.

Uses:
BestEffortBroadcast, **instance** *beb*;
PerfectPointToPointLinks, **instance** *pl*;
PerfectFailureDetector, **instance** $\mathcal{P}$.

upon event ⟨ *onrr*, *Init* ⟩ **do**
$val := \bot$;
$correct := \Pi$;
$writeset := \emptyset$;

upon event ⟨ $\mathcal{P}$, *Crash* | p ⟩ **do**
$correct := correct \setminus \{p\}$;

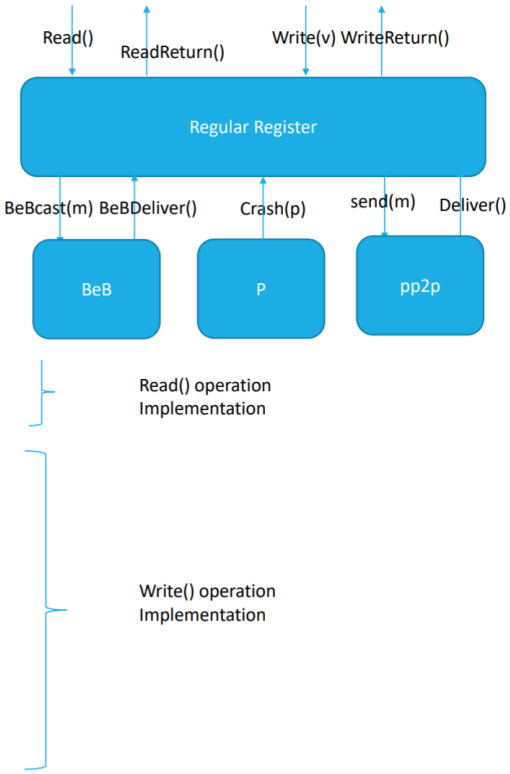
upon event ⟨ *onrr*, *Read* ⟩ **do**
trigger ⟨ *onrr*, *ReadReturn* | val ⟩;

upon event ⟨ *onrr*, *Write* | v ⟩ **do**
trigger ⟨ *beb*, *Broadcast* | [WRITE, v] ⟩;

upon event ⟨ *beb*, *Deliver* | q, [WRITE, v] ⟩ **do**
$val := v$;
trigger ⟨ *pl*, *Send* | q, ACK ⟩;

upon event ⟨ *pl*, *Deliver* | p, ACK ⟩ **do**
$writeset := writeset \cup \{p\}$;

upon $correct \subseteq writeset$ **do**
$writeset := \emptyset$;
trigger ⟨ *onrr*, *WriteReturn* ⟩;



The **BEB instance** will be used in order to *broadcast* to all the *processes* the new variable during the *write operation*. The **pl instance** instead is used when all the *processes* have to send the *ack* back to the *writer*. The *writeset* is used from the *writer* in order to keep track of all *process* that confirms the *receive* of the update of the variable, and when the number of *correct process* is a subset of the process that receives the *ack* the *operation* of write is closed and the *writeset* is set to 0. So for the *write operation* we need at most $2N$ *messages* and for *read operation* 0 *messages* cause it's a local operation.

11.1.2 Fail-Silent Algorithm

The problem with this *algorithm* is that it doesn't ensure *validity* if the *failure detector* is *not perfect*, in fact in this case the *validity property* is not respect. So in this case we can use a different *algorithm* that doesn't use a *failure detector*. This algorithm is called **Fail-silent algorithm: majority voting regular register** and the idea is that each *process* locally stores a copy of the current *value* of the *register* and each *written value* is univocally associated to a *timestamp*, the *writer* and the *reader processes* use a set of *witness process*, to track the last value *written*. We use a **Quorum** that this an *intersection* of any two sets of *witness processes* not empty, and a **Majority Voting**, so each set is constituted by a majority of *processes*:

Algorithm 4.2: Majority Voting Regular Register

Implements:
$(1, N)$-RegularRegister, **instance** *onrr*.
Uses:
BestEffortBroadcast, **instance** *beb*;
PerfectPointToPointLinks, **instance** *pl*.

upon event $\langle$ *onrr*, *Init* $\rangle$ **do**
$(ts, val) := (0, \perp)$;
$wts := 0$;
acks $:= 0$;
rid $:= 0$;
readlist $:= [\perp]^N$;

upon event $\langle$ *onrr*, *Write* | v $\rangle$ **do**
$wts := wts + 1$;
acks $:= 0$;
trigger $\langle$ *beb*, *Broadcast* | [WRITE, wts, v] $\rangle$;

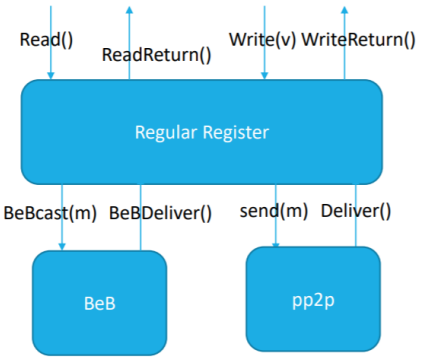
upon event $\langle$ *beb*, *Deliver* | p, [WRITE, ts', v'] $\rangle$ **do**
if $ts' > ts$ **then**
$(ts, val) := (ts', v')$;
trigger $\langle$ *pl*, *Send* | p, [ACK, ts'] $\rangle$;

upon event $\langle$ *pl*, *Deliver* | q, [ACK, ts'] $\rangle$ **such that** $ts' = wts$ **do**
acks $:=$ *acks* $+ 1$;
if *acks* $> N/2$ **then**
acks $:= 0$;
trigger $\langle$ *onrr*, *WriteReturn* $\rangle$;

upon event $\langle$ *onrr*, *Read* $\rangle$ **do**
rid $:=$ *rid* $+ 1$;
readlist $:= [\perp]^N$;
trigger $\langle$ *beb*, *Broadcast* | [READ, *rid*] $\rangle$;

upon event $\langle$ *beb*, *Deliver* | p, [READ, r] $\rangle$ **do**
trigger $\langle$ *pl*, *Send* | p, [VALUE, r, ts, val] $\rangle$;

upon event $\langle$ *pl*, *Deliver* | q, [VALUE, r, ts', v'] $\rangle$ **such that** $r =$ *rid* **do**
readlist[q] $:= (ts', v')$;
if #(*readlist*) $> N/2$ **then**
$v :=$ *highestval*(*readlist*);
readlist $:= [\perp]^N$;
trigger $\langle$ *onrr*, *ReadReturn* | v $\rangle$;



When a *process* need to **write** it will begin to *broadcast* by sending its *value* and its *timestamp* (increased). When we receive a *message* in the **deliver event of the write**, I will check if the *timestamp* received is bigger then the current *timestamp* of the value, and in this case I update the *value* and will send the *ACK* back. When the *writer* receives at least $N/2$ ACK's (since we have the assumption of the *majority* of *correct process*) will trigger the **WriteReturn**. When instead we have a **read operation**, since we don't have a *perfect failure detector* I cannot be sure that my *value* is still *correct*, I need to consult all the other *process* in order to obtain a *quorum* (so to obtain a *variable*). In the **deliver event of the read** we do the *quorum*, in fact the first control is that the r (*timestamp* of the *read*) received is the same of my actual *rid* e will be inserted in the *readlist* and when the number of *processes* in the list is at least the half of the total number of *process* we will trigger the **ReadReturn**. In order to perform a *Write operation* or a *Read operation* we need at most $2N$ *messages*.

11.2 Atomic Register Interface

The **Atomic Register Interface** has the same *properties* of the *regular* plus the **Ordering property**:

Module 4.2: Interface and properties of a $(1, N)$ atomic register

Module:

Name: $(1, N)$-AtomicRegister, **instance** *onar*.

Events:

Request: $\langle$ *onar*, *Read* $\rangle$: Invokes a read operation on the register.

Request: $\langle$ *onar*, *Write* | v $\rangle$: Invokes a write operation with value v on the register.

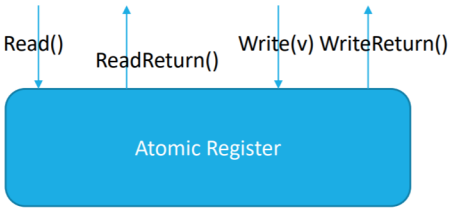
Indication: $\langle$ *onar*, *ReadReturn* | v $\rangle$: Completes a read operation on the register with return value v.

Indication: $\langle$ *onar*, *WriteReturn* $\rangle$: Completes a write operation on the register.

Properties:

ONAR1–ONAR2: Same as properties ONRR1–ONRR2 of a $(1, N)$ regular register (Module 4.1).

ONAR3: *Ordering:* If a read returns a value v and a subsequent read returns a value w, then the write of w does not precede the write of v.



In order to pass from a *Regular Register* $(1, N)$ to an **Atomic Register** $(1, N)$ we have to distinguish two phases:

- We use a *Regular Register* $(1, N)$ to build an **Atomic Register** $(1, 1)$;
- We use a set of *Atomic Registers* $(1, 1)$ to build an **Atomic Register** $(1, N)$;

11.2.1 Regular Register (1,N) to Atomic Register (1,1)

Algorithm 4.3: From $(1, N)$ Regular to $(1, 1)$ Atomic Registers

Implements:
$(1, 1)$-AtomicRegister, **instance** *ooar*.

Uses:
$(1, N)$-RegularRegister, **instance** *onrr*.

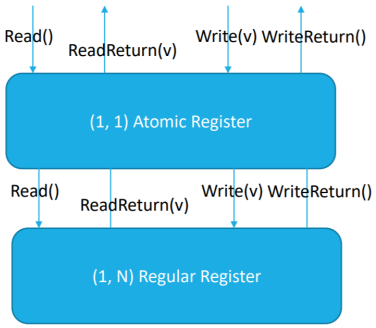
upon event ⟨ *ooar*, *Init* ⟩ **do**
$(ts, val) := (0, \perp)$;
$wts := 0$;

upon event ⟨ *ooar*, *Write* | v ⟩ **do**
$wts := wts + 1$;
trigger ⟨ *onrr*, *Write* | (wts, v) ⟩;

upon event ⟨ *onrr*, *WriteReturn* ⟩ **do**
trigger ⟨ *ooar*, *WriteReturn* ⟩;

upon event ⟨ *ooar*, *Read* ⟩ **do**
trigger ⟨ *onrr*, *Read* ⟩;

upon event ⟨ *onrr*, *ReadReturn* | (ts', v') ⟩ **do**
if $ts' > ts$ **then**
$(ts, val) := (ts', v')$;
trigger ⟨ *ooar*, *ReadReturn* | *val* ⟩;



Where in order to respect the **Ordering property** there is a control on the *timestamp* received, where will be checked if it's greater than the previously *read* value. Each *Write operation* or *Read operation* request a *write/read* on a *regular register*.

11.2.2 Atomic Register (1,1) to Atomic Register (1,N)

Algorithm 4.4: From $(1, 1)$ Atomic to $(1, N)$ Atomic Registers

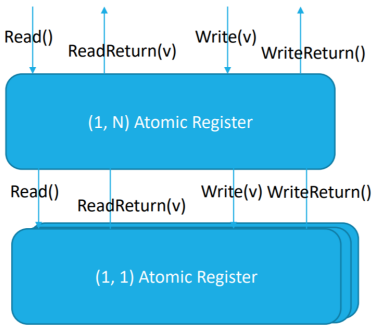
Implements:
$(1, N)$-AtomicRegister, **instance** *onar*.

Uses:
$(1, 1)$-AtomicRegister (multiple instances).

upon event ⟨ *onar*, *Init* ⟩ **do**
$ts := 0$;
acks := 0;
writing := FALSE;
readval := $\perp$;
readlist := $[\perp]^N$;
forall $q \in \Pi, r \in \Pi$ **do**
Initialize a new instance *ooar.q.r* of $(1, 1)$-AtomicRegister with writer r and reader q;

upon event ⟨ *onar*, *Write* | v ⟩ **do**
$ts := ts + 1$;
writing := TRUE;
forall $q \in \Pi$ **do**
trigger ⟨ *ooar.q.self*, *Write* | (ts, v) ⟩;

upon event ⟨ *ooar.q.self*, *WriteReturn* ⟩ **do**
acks := *acks* + 1;
if *acks* = N **then**
acks := 0;
if *writing* = TRUE **then**
trigger ⟨ *onar*, *WriteReturn* ⟩;
writing := FALSE;
else
trigger ⟨ *onar*, *ReadReturn* | *readval* ⟩;



upon event ⟨ *onar*, *Read* ⟩ **do**
forall $r \in \Pi$ **do**
trigger ⟨ *ooar.self.r*, *Read* ⟩;

upon event ⟨ *ooar.self.r*, *ReadReturn* | (ts', v') ⟩ **do**
readlist[r] := (ts', v');
if #(*readlist*) = N **then**
(*maxts*, *readval*) := highest(*readlist*);
readlist := $[\perp]^N$;
forall $q \in \Pi$ **do**
trigger ⟨ *ooar.q.self*, *Write* | (*maxts*, *readval*) ⟩;

In the **Write event** we will use the *writing variable* that is used in order to permit to only one *process* to write at time. In the **WriteReturn event** of the *atomic registers* below when we receive a number of *ACK's* equal to the number of the *processes*, the *process* will check if the variable *writing* is *true* (so this process is the *writer*), and in this case will trigger its own $WriteReturn$, else will trigger its *ReadReturn*. In the **ReadReturn event** of the *atomic registers* below we will add a *tuple* with the *received value* and its *timestamp* in the *readlist*, when the number of *items* in the list is N then we will choose the value with the *maximum timestamp* associated (in order to respect the *ordering property*) and we will send to all the other *processes* the new *variable* with a $Write$ on the *atomic registers* below. So it's important to note that for both **read** and **write operation** we need to use a *write operation* of the N *atomic registers* below, so when a $WriteReturn$ event is received we can have two cases: if the *process* is the *writer* we write in the *register*, if the *process* is not the writer it will *read* thanks to the *ReadReturn*.

11.2.3 Read-Impose Write-All Algorithm

Even the *atomic register* has a modified version of the *Read-One Write-All algorithm* of the *regular register*, and it's called **Read-Impose Write-All Algorithm**. The idea is that the *read operation* writes, and it's called *Read-Impose Write-All* cause a *read operation* imposes to all *correct processes* to update their local copy of the *register* with the value *read*, unless they store a more recent value:

Algorithm 4.5: Read-Impose Write-All

Implements:
$(1, N)$-AtomicRegister, **instance** *onar*.

Uses:
BestEffortBroadcast, **instance** *beb*;
PerfectPointToPointLinks, **instance** *pl*;
PerfectFailureDetector, **instance** $\mathcal{P}$.

upon event $\langle$ *onar*, *Init* $\rangle$ **do**
$(ts, val) := (0, \perp)$;
correct := Π;
writeset := $\emptyset$;
readval := $\perp$;
reading := FALSE;

upon event $\langle$ $\mathcal{P}$, *Crash* | p $\rangle$ **do**
correct := *correct* $\setminus \{p\}$;

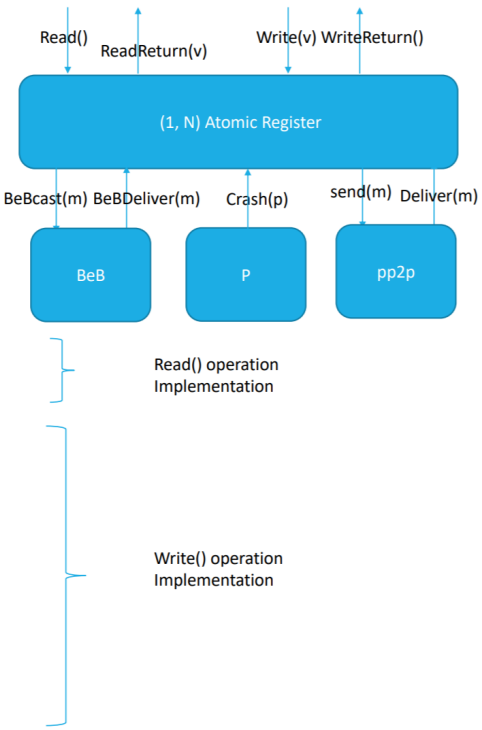
upon event $\langle$ *onar*, *Read* $\rangle$ **do**
reading := TRUE;
readval := *val*;
trigger $\langle$ *beb*, *Broadcast* | [WRITE, *ts*, *val*] $\rangle$;

upon event $\langle$ *onar*, *Write* | v $\rangle$ **do**
trigger $\langle$ *beb*, *Broadcast* | [WRITE, $ts + 1, v$] $\rangle$;

upon event $\langle$ *beb*, *Deliver* | p, [WRITE, ts', v'] $\rangle$ **do**
if $ts' > ts$ **then**
$(ts, val) := (ts', v')$;
trigger $\langle$ *pl*, *Send* | p, [ACK] $\rangle$;

upon event $\langle$ *pl*, *Deliver* | p, [ACK] $\rangle$ **then**
writeset := *writeset* $\cup \{p\}$;

upon *correct* $\subseteq$ *writeset* **do**
writeset := $\emptyset$;
if *reading* = TRUE **then**
reading := FALSE;
trigger $\langle$ *onar*, *ReadReturn* | *readval* $\rangle$;
else
trigger $\langle$ *onar*, *WriteReturn* $\rangle$;



In this **algorithm** we use the *Best Effort Broadcast*, a *Perfect Failure Detector* and a *PP2P*. When a *process* want to **read** a variable from the *register* it will *broadcast* a *write operation* to all the other *processes* its own local variable with its *timestamp*. When a process **receive** a *beb deliver* with a *write*, it will check if the *timestamp* arrived is bigger than its own *timestamp* of that *variable*. and in this case it will *update* the *tuple* and will send back an ACK to confirm the *deliver* of the *broadcast message*. When the *correct processes* is a subset of the **writeset** (the set of *processes* that has send an ACK) if the *process* is a *reader* (with variable $reading = true$) it will trigger the *ReadReturn* else if the *process* is a *writer* it will trigger the $WriteReturn$. Since for any *read operation* the *reader process* ensures that any other *process* has a *timestamp* greater or equal we ensure the **ordering property**. For **Write** and **Read Operation** we have at most $2N$ *messages*.

11.2.4 Read-Impose Write-Majority algorithm

This algorithm called **Read-Impose Write-Majority algorithm** its a variation of the *Majority Voting algorithm* of the *regular register*, in which we don't have any *failure detector* and we assume to have a *majority* of *correct processes*. The idea is to impose to a *majority* of *process* to have the *value read*:

Algorithm 4.6: Read-Impose Write-Majority (part 1, read)

Implements:
(1, N)-AtomicRegister, **instance** *onar*.

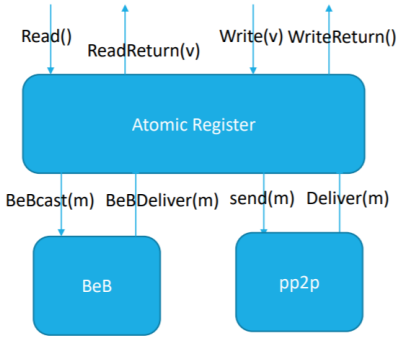
Uses:
BestEffortBroadcast, **instance** *beb*;
PerfectPointToPointLinks, **instance** *pl*.

upon event ⟨ *onar*, *Init* ⟩ **do**
$(ts, val) := (0, \perp)$;
$wts := 0$;
acks := 0;
rid := 0;
readlist := $[\perp]^N$;
readval := $\perp$;
reading := FALSE;

upon event ⟨ *onar*, *Read* ⟩ **do**
rid := *rid* + 1;
acks := 0;
readlist := $[\perp]^N$;
reading := TRUE;
trigger ⟨ *beb*, *Broadcast* | [READ, *rid*] ⟩;

upon event ⟨ *beb*, *Deliver* | p, [READ, r] ⟩ **do**
trigger ⟨ *pl*, *Send* | p, [VALUE, r, ts, val] ⟩;

upon event ⟨ *pl*, *Deliver* | q, [VALUE, r, ts', v'] ⟩ **such that** $r = rid$ **do**
readlist[q] := (ts', v');
if #(*readlist*) > $N/2$ **then**
(*maxts*, *readval*) := *highest*(*readlist*);
readlist := $[\perp]^N$;
trigger ⟨ *beb*, *Broadcast* | [WRITE, *rid*, *maxts*, *readval*] ⟩;



upon event ⟨ *onar*, *Write* | v ⟩ **do**
rid := *rid* + 1;
$wts := wts + 1$;
acks := 0;
trigger ⟨ *beb*, *Broadcast* | [WRITE, *rid*, wts, v] ⟩;

upon event ⟨ *beb*, *Deliver* | p, [WRITE, r, ts', v'] ⟩ **do**
if $ts' > ts$ **then**
$(ts, val) := (ts', v')$;
trigger ⟨ *pl*, *Send* | p, [ACK, r] ⟩;

upon event ⟨ *pl*, *Deliver* | q, [ACK, r] ⟩ **such that** $r = rid$ **do**
acks := *acks* + 1;
if *acks* > $N/2$ **then**
acks := 0;
if *reading* = TRUE **then**
reading := FALSE;
trigger ⟨ *onar*, *ReadReturn* | *readval* ⟩;
else
trigger ⟨ *onar*, *WriteReturn* ⟩;

Since in this *algorithm* we don't use any *failure detector*, in addiction to a **write timestamp** we need a **read timestamp** that will be incremented in both operation of *read* and *write*. In the

Read event we have a *broadcast* used to report to all the other *processes* that my *process* need a *quorum* on this variable. When we receive at least $N/2$ responses the *process* will take only the *messages* with $r = rid$ (so the current read) and will take the value with *highest timestamp* and will trigger a *broadcast* that imposes to all the other *process* to change its own *variable* to the current one decided by the **quorum**. In the **deliver** of the *write event* instead to maintain the *ordering property* will take the value with a *timestamp* bigger than the actual and will send an ACK. So, like for the *read event,* if the number of ACK received are at least $N/2$ if the *process* is a *reader* will trigger the *ReadReturn* else if the *process* is a *writer* it will trigger the $WriteReturn$. Since the *read* imposes the *write* of the value read to a *majority of processes* and to the property of *intersection* of *quorum* the **ordering property** is respected. For *write operation* we need at most $2N$ *messages*, instead for *read operation* we need at most $4N$ *messages*, cause the *read* does two *broadcast* one for obtaining a *quorum* and the other for impose its value to all the other *processes*.

12 Software Replication

In *distributed system*, **Software Replication** is used for *fault tolerance purposes*, so for guarantee the *availability* of a *service* (an *object*) despite *failures*. If we consider p the **failure probability** of an object O, the **availability** of O is $1 - p$. If we **replicate** an *object* O on n *nodes* now its availability is $(1 - p)^n$. So now the *system model* is composed by a set of *processes* that are connected with a *PP2P* and they may *crash*.

These **processes** interacts wit a set of **objects** X located at different sites managed by *processes*:

- Each object has a state, that can be accessed through operations;
- An operation by a process p_i on an object $x \in X$ is a pair invocation/response:
  - The **operation invocation** is: $x\ op(arg)\ p_i$ where arg is the argument of the operation;
  - The **operation response** is: $x\ ok(res)\ p_i$ where res is the result of the operation;
  - The **pair invocation/response** is: $x\ op(arg)\ p_i / x\ ok(res)\ p_i$
- After issuing an *invocation* a *process* is **blocked** until it receives the *matching response*;

In order to tolerate **process crash failures**, a *logical object* must have several *physical replicas* located in different sites, we assume that a *process* p_i crashes when an *object* x_i crashes. It's important that *replication* is **transparent** to the *client processes*, this means that *client* think that it is interacting with the same *server*, even if in reality it is interacting with a *correct copy* of it.

There are three **consistency criteria**:

- **Linearizability**;
- **Sequential consistency**;
- **Causal consistency**;

The first two criteria composes the **Strong Consistency**, instead the third one is the **Weak Consistency**.

If we call the **precedence relation**: $\prec$ and the **concurrency relation**: $\|$,we say that an *execution* E is **linearizable** if there exists a *sequence* S including all *operations* of E such that:

- for any operation O_1 and O_2 such that $O_1 \prec O_2$, then O_1 appears before O_2 in the sequence S;
- the sequence S is **legal**, so for every *object* in the *sub-sequence* of S they have to respect the *sequential specification* of the object;

There is a sufficient condition for **linearizability**: *replicas* must agree on the *set of invocations* they handle and on the *order* according to which they handle these invocations:

- **Atomicity**: Given an *invocation* x $op(arg)$ p_i, if one *replica* of the object x handles this *invocation*, then every *correct replica* of x also handles that *invocation*;
- **Ordering**: Given two *invocations* x $op(arg)$ p_i and x $op(arg)$ p_j if two *replicas* handle both the *invocations*, they handle them in the same *order*;

There are two main techniques that implement *linearizability*: **primary backup** and **active replication**.

12.1 Primary Backup

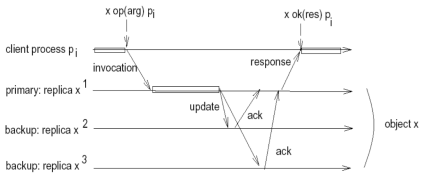
Primary:

- Receives *invocations* from *clients* and sends back the answers;
- Given an *object* x, then $prim(x)$ returns the *primary* of x;

Backup:

- Interacts with $prim(x)$;
- Used to guarantee *fault tolerance* by replacing a *primary* when crashes;

12.1.1 No-Crash scenario



So before send back the *response* to the *client*, the *primary replica* sends an *update* to all the other *correct backups* and only after that it gets an *ACK* from them it will send a *response* to the client. **Linearizability** is guaranteed since the *order* in which $prim(x)$ receive *client invocations* define the *order* of the *operation* on the object.

12.1.2 Crash scenario

There are three different scenarios:

- **Scenario 1**, Primary fails after the *client* receives the *answer*, and there are two cases:
  - The *client* doesn't receive the *response* due to *PP2P*, if the *response* is lost client retransmits the *request* after a timeout, in this case the new *primary* will recognize the *request* already issued and will send back the result without updating the *replicas*;

  - The *client* receives the answer, so it's fine;

- **Scenario 2**, Primary fails before sending *update messages*:
  - The *client* doesn't get any answer and will resend the *request* after a timeout, so the new *primary* will handle the request as new;

- **Scenario 3**, Primary fails after sending *update messages* but before receiving all ACK:
  - In order to guarantee *atomicity*, the update its received either by all or by no one, when a *primary* fails there is need to elect another one among all the replicas.

12.2 Active Replication

Here all the *replicas* have the same *role*, and each *replica* is *deterministic* (so if they have same *state* and same *input* they produce the same *output*). So the *client* will receive the same *response* from all the *replicas*. In this way the *client* doesn't need to wait the *response* of all *replicas*, he will took the *response* sent by the first *correct replica* it receives. In this case in order to ensure **linearizability**:

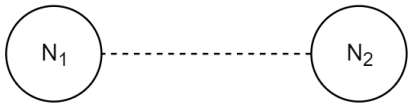
- **Atomicity**: if a *replica* executes an *invocation*, all *correct replicas* execute the same *invocation*;

- **Ordering**: no two *correct replicas* have to execute two *invocations* in different order;

So we need a **total order broadcast**, even for the *clients*. Obviously **Active Replication** doesn't need **recovery action** upon the *failure* of a *replica*.

13 Cap theorem

CAP theorem (**Consistency**, **Availability**, **Partition tolerance**) states that we can choose only two of these in a *Distributed System* in case of *failures*, so we cannot guarantee all of these at the same moment. To see why this rule holds, we image a situation in which we have *two nodes* connected each other:



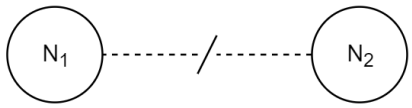
Data are replicated across the *nodes*, so they have the same *dataset*. Now we see what **C**,**A** and **P** means in this situation:

- **Consistency**: if the *dataset* in N_1 is changed, then we need to change also the *dataset* in N_2 so that it look the same in both of them;

- **Availability**: as long as both N_1 and N_2 are up and running, I should be able to *query/update* data on any of them;

- **Partition tolerance**: if the *link* between N_1 and N_2 fails, I should till be able to *query/update* my *dataset*;

The best way to understand how the *theorem* works is to see what happens during a **network partition**:



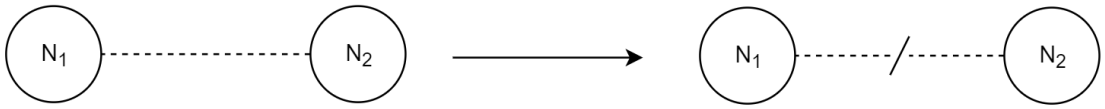
So we can see that N_1 and N_2 cannot *communicate* anymore, so let's assume that we have some means to *discover partition*. Now if someone talks to N_1 and changes the dataset, N_1 cannot propagate these changes to N_2:

- If we choose **consistency**, we have to block all the updates on the *system* (both *nodes*) but this makes it *unavailable*;
- If we choose **availability**, we make different updates on the nodes this erases *consistency*;

We can see that we have to sacrifice one of the them, one decent solution is to reduce availability to one single node and then update the other when the link is re-available.

So, we can choose between **CP** or **AP**, since choosing **CA** is obviously nonsense, cause in this case we don't have any info on that system so we can't work on it. It is also important to say that we can also choose the **C-level** and **A-level** so we are not constrained into chasing one or the other. For example we can be *read available* on any node but not *update available*, or be *available* on only one *node* and apply some *post-partition recovery*. Or we can choose eventual *consistency*, if our app is okay in using *slightly old data* in some nodes, to improve *availability*. Now let's write a formal proof of the **CAP Theorem by contradiction**:

Let's assume that there exists a *system* that is *consistent*, *available* and *partition tolerant*. Then we will *partition* the system like:



Next, we have to *update* the value on N_1 to v_1 and since the system is *available* we can do it. Next we will *read* the value of N_2 but it returns v_0 cause the *link* is broken and N_1 cannot pass v_1 to N_2. This is an *inconsistent scheme* so we have a **contradiction**.

14 Byzantine Tolerant Broadcast

Byzantine processes are *process* that may deviate arbitrarily from the *instructions* that an *algorithm* assign to them, or act as if they were deliberately preventing the *algorithm* from reaching its goals. The basic step to fight them is to use some *cryptography mechanisms* to implement *perfect links abstraction*, but alone it doesn't allow to tolerate *Byzantine processes*.

Module 2.5: Interface and properties of authenticated perfect point-to-point links

Module:

Name: AuthPerfectPointToPointLinks, **instance** *al*.

Events:

Request: ⟨ *al*, *Send* | *q*, *m* ⟩: Requests to send message m to process q.

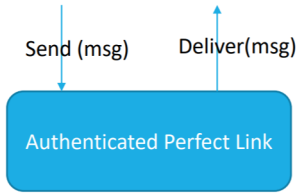
Indication: ⟨ *al*, *Deliver* | *p*, *m* ⟩: Delivers message m sent by process p.

Properties:

AL1: *Reliable delivery:* If a correct process sends a message m to a correct process q, then q eventually delivers m.

AL2: *No duplication:* No message is delivered by a correct process more than once.

AL3: *Authenticity:* If some correct process q delivers a message m with sender p and process p is correct, then m was previously sent to q by p.



Same as Perfect point-to-point links

14.1 Byzantine Consistent Broadcast

Module 3.11: Interface and properties of Byzantine consistent broadcast

Module:

Name: ByzantineConsistentBroadcast, **instance** *bcb*, with sender s.

Events:

Request: ⟨ *bcb*, *Broadcast* | *m* ⟩: Broadcasts a message m to all processes. Executed only by process s.

Indication: ⟨ *bcb*, *Deliver* | *p*, *m* ⟩: Delivers a message m broadcast by process p.

Properties:

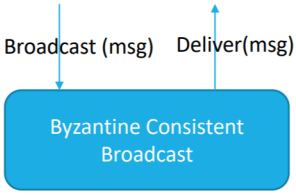
BCB1: *Validity:* If a correct process p broadcasts a message m, then every correct process eventually delivers m.

BCB2: *No duplication:* Every correct process delivers at most one message.

BCB3: *Integrity:* If some correct process delivers a message m with sender p and process p is correct, then m was previously broadcast by p.

BCB4: *Consistency:* If some correct process delivers a message m and another correct process delivers a message m', then $m = m'$.

The specification refers to a single broadcast event!



The **consistency property** is very important cause it is referred to one single *broadcast event*, so every *correct process* delivers the same *message*. In the **broadcast event** we will send a *message* to all the *processes* and this *message* contains the *id* of the *sender*, this is used in the *deliver event* where we will check if the *message* is from the *sender* and in this case the *process* will *resend* the *message* to all the other *processes* with *sender* himself with an **echo**. When a *process* receive an

echo message, they will add in the *echo* array the *message* in the position of the *process* that sent it. When the number of *processes* with the message m in the *echo array* is bigger than $\frac{N+f}{2}$ we can finally *deliver* the *broadcast*.

Algorithm 3.16: Authenticated Echo Broadcast

Implements:
ByzantineConsistentBroadcast, **instance** *bcb*, with sender s.

Uses:
AuthPerfectPointToPointLinks, **instance** *al*.

upon event $\langle$ *bcb*, *Init* $\rangle$ **do**
sentecho := FALSE;
delivered := FALSE;
echos := $[\bot]^N$;

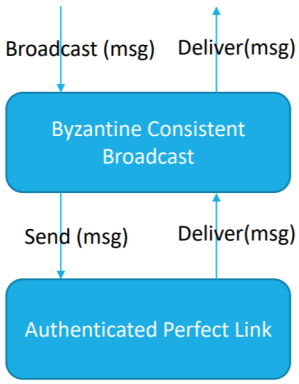
upon event $\langle$ *bcb*, *Broadcast* | m $\rangle$ **do** // only process s
forall $q \in \Pi$ **do**
trigger $\langle$ *al*, *Send* | q, [SEND, m] $\rangle$;

upon event $\langle$ *al*, *Deliver* | p, [SEND, m] $\rangle$ **such that** $p = s$ **and** *sentecho* = FALSE **do**
sentecho := TRUE;
forall $q \in \Pi$ **do**
trigger $\langle$ *al*, *Send* | q, [ECHO, m] $\rangle$;

upon event $\langle$ *al*, *Deliver* | p, [ECHO, m] $\rangle$ **do**
if *echos*[p] = $\bot$ **then**
echos[p] := m;

upon exists $m \neq \bot$ such that $\#(\{p \in \Pi \mid echos[p] = m\}) > \frac{N+f}{2}$
and *delivered* = FALSE **do**
delivered := TRUE;
trigger $\langle$ *bcb*, *Deliver* | s, m $\rangle$;

Correctness is ensured if N>3f



We know that in a **normal quorum** we need at least $\frac{N}{2}$ *correct processes*, but in this case since we have to deal with f *byzantine processes* we need at least $\frac{N+f}{2}$. The number of *correct processes* in such a system becomes: $\frac{N+f}{2} - f$ that is equal to $\frac{N-f}{2}$. So if we have to be sure that two *byzantine quorums* returns at least one *correct process* we will consider the *edge case* in which we have two *disjoint quorums* of $\frac{N-f}{2}$, they will have more than $N - f$ (sum of the two *quorums* quantity) *correct processes* to have at least one intersection. The $N - f$ is the number of *correct processes* I need to consider the *system correct* and has to be $> \frac{N+f}{2}$ so we have $N > 3f$ for which *correctness* in ensured.

14.2 Byzantine Reliable Broadcast

Module 3.12: Interface and properties of Byzantine reliable broadcast

Module:

Name: ByzantineReliableBroadcast, **instance** *brb*, with sender s.

Events:

Request: $\langle$ *brb*, *Broadcast* | m $\rangle$: Broadcasts a message m to all processes. Executed only by process s.

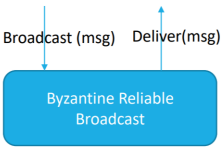
Indication: $\langle$ *brb*, *Deliver* | p, m $\rangle$: Delivers a message m broadcast by process p.

Properties:

BRB1–BRB4: Same as properties BCB1–BCB4 in Byzantine consistent broadcast (Module 3.11).

BRB5: *Totality:* If some message is delivered by any correct process, every correct process eventually delivers a message.

The specification refers to a single broadcast event!



When the **quorum** of a *receiving process* is reached the *process* will send a **ready message** to all the other *processes* with its *id* and the *message* received from the *echo*. When a **deliver event**

of an *echo message* arrives the *ready array* is filled with the *message* in the position of the *sender process*. The **ready message** of the *broadcast* can be send even when the number of *ready messages* delivered is higher than f, this cause in this case at least one *process* is *correct*. At the end in order to deliver the **broadcast message** we need to check if the number of *processes* from which I received the *ready message* is at least $> 2f$ and this is made in order to avoid *byzantine processes* that sends twice a *ready message*.

Algorithm 3.18: Authenticated Double-Echo Broadcast

Implements:
ByzantineReliableBroadcast, **instance** *brb*, with sender s.

Uses:
AuthPerfectPointToPointLinks, **instance** *al*.

upon event ⟨ *brb*, *Init* ⟩ **do**
sentecho := FALSE;
sentready := FALSE;
delivered := FALSE;
echos := $[\perp]^N$;
readys := $[\perp]^N$;

upon event ⟨ *brb*, *Broadcast* | m ⟩ **do** // only process s
forall $q \in \Pi$ **do**
trigger ⟨ *al*, *Send* | q, [SEND, m] ⟩;

upon event ⟨ *al*, *Deliver* | p, [SEND, m] ⟩ **such that** $p = s$ **and** *sentecho* = FALSE **do**
sentecho := TRUE;
forall $q \in \Pi$ **do**
trigger ⟨ *al*, *Send* | q, [ECHO, m] ⟩;

upon event ⟨ *al*, *Deliver* | p, [ECHO, m] ⟩ **do**
if *echos*[p] = $\perp$ **then**
echos[p] := m;

upon exists $m \neq \perp$ such that $\#(\{p \in \Pi \mid echos[p] = m\}) > \frac{N+f}{2}$
and *sentready* = FALSE **do**
sentready := TRUE;
forall $q \in \Pi$ **do**
trigger ⟨ *al*, *Send* | q, [READY, m] ⟩;

upon event ⟨ *al*, *Deliver* | p, [READY, m] ⟩ **do**
if *readys*[p] = $\perp$ **then**
readys[p] := m;

upon exists $m \neq \perp$ such that $\#(\{p \in \Pi \mid readys[p] = m\}) > f$
and *sentready* = FALSE **do**
sentready := TRUE;
forall $q \in \Pi$ **do**
trigger ⟨ *al*, *Send* | q, [READY, m] ⟩;

upon exists $m \neq \perp$ such that $\#(\{p \in \Pi \mid readys[p] = m\}) > 2f$
and *delivered* = FALSE **do**
delivered := TRUE;
trigger ⟨ *brb*, *Deliver* | s, m ⟩;

15 Byzantine Tolerant Consensus

Since byzantine processes may invent values or claim to have proposed different values we need to adapt the validity property of the consensus. So, we restrict the specification only to correct processes and we define two different type of validity weak and strong:

Weak Byzantine Consensus:

Module 5.10: Interface and properties of weak Byzantine consensus

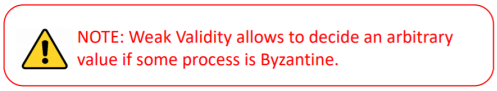
Properties:

WBC1: *Termination:* Every correct process eventually decides some value.

WBC2: *Weak validity:* If all processes are correct and propose the same value v, then no correct process decides a value different from v; furthermore, if all processes are correct and some process decides v, then v was proposed by some process.

WBC3: *Integrity:* No correct process decides twice.

WBC4: *Agreement:* No two correct processes decide differently.



Strong Byzantine Consensus:

Module 5.11: Interface and properties of (strong) Byzantine consensus

Module:

Name: ByzantineConsensus, **instance** *bc*.

Events:

Request: ⟨ *bc*, *Propose* | v ⟩: Proposes value v for consensus.

Indication: ⟨ *bc*, *Decide* | v ⟩: Outputs a decided value v of consensus.

Properties:

BC1 and **BC3–BC4:** Same as properties WBC1 and WBC3–WBC4 in weak Byzantine consensus (Module 5.10).

BC2: *Strong validity:* If all correct processes propose the same value v, then no correct process decides a value different from v; otherwise, a correct process may only decide a value that was proposed by some correct process or the special value $\square$.

15.1 Byzantine Generals Problem

The **byzantine generals problem** is a problem of *consensus* for *byzantine processes* in which we have a *general* (called **commander**) that can communicate to the other *generals* (called **lieu-**

tenant) to *attack* or to *retreats*, in which *commander* and *lieutenants* can be *traitors* (*byzantine*), and in order to win all *loyal general* attack or retreat, and after the *commander* send the order *lieutenants* can communicate between them. So we have two goals:

- All **loyal general** decide upon the same plan of action;
- The *traitors* cannot cause the *loyal general* to adopt a *bad plan*;

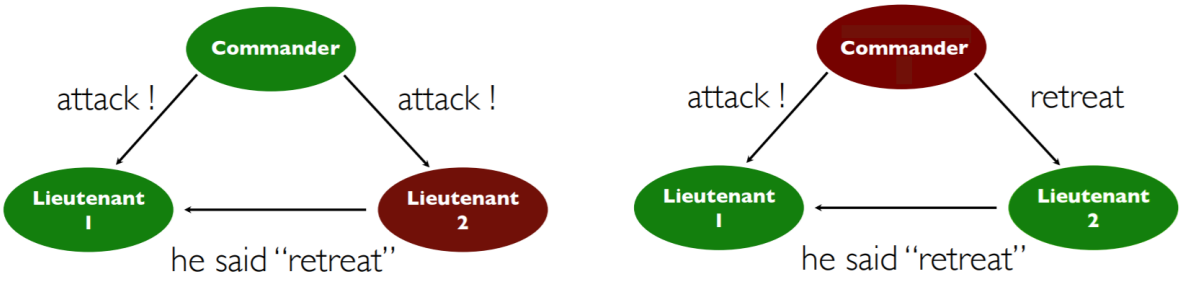
So we can rephrasing the goal as:

- Any two *loyal general* use the same value of $v(i)$, where $v(i)$ is the information communicated by the i^th *general*;
- If the i^th *general* is *loyal*, then the value he sends must be used by every *loyal general* as the value of $v(i)$;

So the property are:

- All *loyal lieutenants* obey the same order;
- If the *commander* is *loyal*, then every *loyal general* obeys the order he sends;

Like from the the *correctness* of the past section we need $N \geq 3f + 1$ of **loyal generals** in order to let this work:



So in this **system model** we have a *reliable communication channels*, the *message source* is known, *message omissions* can be detected and the *default decision* for *lieutenants* is $RETREAT$. We will use a *recursion algorithm* and we define a set of protocols: $OM(f)$ for which:

- $OM(0)$
  - The **commander** sends his value to every *lieutenant*;
  - Each **lieutenant** uses the value received or $RETREAT$ if he receives no value;
- $OM(f)$ with $f > 0$:
  - The *commander* sends his value to every *lieutenant*;
  - For each v_i, where $v(i)$ is the value of the *lieutenant* i received by the *commander*, the *lieutenant* send its value to all the other $N - 2$ *lieutenants* ($N - 2$ to everybody minus himself and the *commander*);

– For each value received (not counting duplicate) and by considering also is own value, the *lieutenant* uses the **value majority** too choose is own result;

So in less words, when the *lieutenants* receives the *variable* from the *commander* they starts to *resend* the value between them. When a *lieutenant* receive a message from another *lieutenant* they add the *variable* in one *array* at the position of the *sender* and at the end they choose the *majority* in base of the content of the *array* and if its not possible to choose a *majority* the *lieutenant* decide to *retire*.

15.1.1 Byzantine Generals Problem with Authentication Codes

The problem is since this a **recursive algorithm** we have a large number of *messages* and it's very *complex* so a solution can be to use **messages authentication codes**:

- The *commander* signs and sends his value $(v : 0)$ to every *lieutenant*.
- For each i:
  - If *lieutenant* i receives a *messages* $(v : 0)$ from the *commander* and has not received yet any order then: $V_i = \{v\}$ and sends $v : 0 : i$ to every *lieutenant*;
  - If *lieutenant* i receives a *messages* $v : 0 : j_1 : \ ... \ : j_k$ and v is not in V_i then: adds v to V_i and if $k < f$ sends $v : 0 : j_1 : \ ... \ : j_k$ to every *lieutenant* other than $j_1 : \ ... \ : j_k$
- For each i : when *lieutenant* i receives no more *messages*, he obeys the order $choice(V_i)$;

16 Byzantine Tolerant Registers

16.1 Safe Register

A **safe register** is a *register* in which we have a **validity property** that *states* that if we have a *read operation* that is not *concurrent* with a *write* will return the last value written:

Module 4.5: Interface and properties of a $(1, N)$ Byzantine safe register

Module:

Name: $(1, N)$-ByzantineSafeRegister, **instance** *bonsr*, with writer w.

Events:

Request: $\langle$ *bonsr*, *Read* $\rangle$: Invokes a read operation on the register.

Request: $\langle$ *bonsr*, *Write* $\mid v \rangle$: Invokes a write operation with value v on the register. Executed only by process w.

Indication: $\langle$ *bonsr*, *ReadReturn* $\mid v \rangle$: Completes a read operation on the register with return value v.

Indication: $\langle$ *bonsr*, *WriteReturn* $\rangle$: Completes a write operation on the register. Occurs only at process w.

Properties:

BONSR1: *Termination:* If a correct process invokes an operation, then the operation eventually completes.

BONSR2: *Validity:* A read that is not concurrent with a write returns the last value written.

16.1.1 Byzantine Tolerant Safe Register

We have N servers with 1 writer and n readers:

Module 4.5: Interface and properties of a $(1, N)$ Byzantine safe register

Module:

Name: $(1, N)$-ByzantineSafeRegister, **instance** *bonsr*, with writer w.

Events:

Request: $\langle$ *bonsr*, *Read* $\rangle$: Invokes a read operation on the register.

Request: $\langle$ *bonsr*, *Write* $\mid v \rangle$: Invokes a write operation with value v on the register. Executed only by process w.

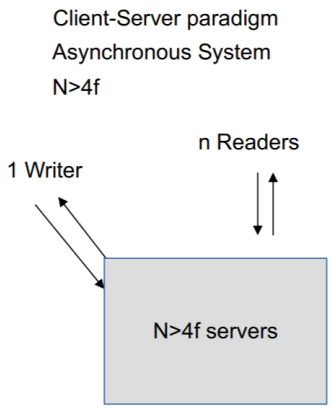
Indication: $\langle$ *bonsr*, *ReadReturn* $\mid v \rangle$: Completes a read operation on the register with return value v.

Indication: $\langle$ *bonsr*, *WriteReturn* $\rangle$: Completes a write operation on the register. Occurs only at process w.

Properties:

BONSR1: *Termination:* If a correct process invokes an operation, then the operation eventually completes.

BONSR2: *Validity:* A read that is not concurrent with a write returns the last value written.



We have to assure that once *writer* returns from a *write operation*, then any following *read operation* returns the last *written value*, so a *writer* sends a request to *servers* and waits for enough *ACK messages* to be sure that enough *correct servers* delivers it. Same for the *read operation*, that sends a *read request* and waits for enough *reply messages* to be able to read the newest value.

The *quorum* that works for *Byzantine Broadcast* here is not enough since *Safe Register* has a stronger semantic, cause we require that a *write operations* is visible to all once it terminates so we need a **Masking Quorum**: $N > 4f$ and quorum: $\frac{N+2f}{2}$.

Algorithm 4.14: Byzantine Masking Quorum

Implements:
$(1, N)$-ByzantineSafeRegister, **instance** *bonsr*, with writer w.

Uses:
AuthPerfectPointToPointLinks, **instance** *al*.

upon event $\langle$ *bonsr*, *Init* $\rangle$ **do**
$(ts, val) := (0, \perp)$;
$wts := 0$;
$acklist := [\perp]^N$;
$rid := 0$;
$readlist := [\perp]^N$;

upon event $\langle$ *bonsr*, *Write* $\mid v \rangle$ **do** // only process w
$wts := wts + 1$;
$acklist := [\perp]^N$;
forall $q \in \Pi$ **do**
trigger $\langle$ *al*, *Send* $\mid q$, [WRITE, wts, v] $\rangle$;

upon event $\langle$ *al*, *Deliver* $\mid p$, [WRITE, ts', v'] $\rangle$ **such that** $p = w$ **do**
if $ts' > ts$ **then**
$(ts, val) := (ts', v')$;
trigger $\langle$ *al*, *Send* $\mid p$, [ACK, ts'] $\rangle$;

upon event $\langle$ *al*, *Deliver* $\mid q$, [ACK, ts'] $\rangle$ **such that** $ts' = wts$ **do**
1 $acklist[q] := \text{ACK}$;
if $\#(acklist) > (N + 2f)/2$ **then**
$acklist := [\perp]^N$;
trigger $\langle$ *bonsr*, *WriteReturn* $\rangle$;

upon event $\langle$ *bonsr*, *Read* $\rangle$ **do**
$rid := rid + 1$;
$readlist := [\perp]^N$;
forall $q \in \Pi$ **do**
trigger $\langle$ *al*, *Send* $\mid q$, [READ, rid] $\rangle$;

upon event $\langle$ *al*, *Deliver* $\mid p$, [READ, r] $\rangle$ **do**
trigger $\langle$ *al*, *Send* $\mid p$, [VALUE, r, ts, val] $\rangle$;

upon event $\langle$ *al*, *Deliver* $\mid q$, [VALUE, r, ts', v'] $\rangle$ **such that** $r = rid$ **do**
$readlist[q] := (ts', v')$;
2 **if** $\#(readlist) > \frac{N+2f}{2}$ **then**
$v :=$ byzhighestval($readlist$);
$readlist := [\perp]^N$;
trigger $\langle$ *bonsr*, *ReadReturn* $\mid v \rangle$;

So we can see that in the **deliver event** of the **write**, when a new *write request* arrives we check if the *sender* is the *writer process* and in this case if the value of the *receiver* is older than the new

value sent, and in this case the *receiver* will update it and after it sends an ACK. When the *writer* receives the ACK it will add in the *acklist* the ACK received, and if the number of ACK received is bigger than $\frac{N+2f}{2}$ (so reachs the *quorum*) we can trigger the *WriteReturn*. When a process want to **read** a value, will trigger a *read request*, and all *processes* will send to him the *value* requested. When the *sender* receives at least $\frac{N+2f}{2}$ values, will select the *variable* that occurs more than f and with the *highest timestamp*, in this way we respect that the *read* returns the most recent value *written*.

16.2 Regular Register

The **regular register** is an evolution of the *Majority voting Algorithm*, and we will discuss two implementation, with *cryptography* and without it.

16.2.1 Regular Register with cryptography

In the *cryptography one*, we have that the *writer* signs the **timestamp-value pair**, so it will send to all the *process* the pair (ts, v) signed, so the *reader* will verifies the *signature* on each *pair* received and will ignore those with invalid *signatures*:

Algorithm 4.15: Authenticated-Data Byzantine Quorum

Implements:
$(1, N)$-ByzantineRegularRegister, **instance** *bonrr*, with writer w.
Uses:
AuthPerfectPointToPointLinks, **instance** *al*.

upon event ⟨ *bonrr*, *Init* ⟩ **do**
$(ts, val, \sigma) := (0, \perp, \perp)$;
$wts := 0$;
$acklist := [\perp]^N$;
$rid := 0$;
$readlist := [\perp]^N$;

upon event ⟨ *bonrr*, *Write* | v ⟩ **do** // only process w
$wts := wts + 1$;
$acklist := [\perp]^N$;
$\sigma := sign(self, bonrr \| self \| \text{WRITE} \| wts \| v)$;
forall $q \in \Pi$ **do**
trigger ⟨ *al*, *Send* | q, [WRITE, wts, σ] ⟩;

upon event ⟨ *al*, *Deliver* | p, [WRITE, ts', v', σ'] ⟩ **such that** $p = w$ **do**
if $ts' > ts$ **then**
$(ts, val, \sigma) := (ts', v', \sigma')$;
trigger ⟨ *al*, *Send* | p, [ACK, ts'] ⟩;

upon event ⟨ *al*, *Deliver* | q, [ACK, ts'] ⟩ **such that** $ts' = wts$ **do**
$acklist[q] :=$ ACK;
if $\#(acklist) > (N + f)/2$ **then**
$acklist := [\perp]^N$;
trigger ⟨ *bonrr*, *WriteReturn* ⟩;

i.e. > 2f

Assumption
N>3f

upon event ⟨ *bonrr*, *Read* ⟩ **do**
$rid := rid + 1$;
$readlist := [\perp]^N$;
forall $q \in \Pi$ **do**
trigger ⟨ *al*, *Send* | q, [READ, rid] ⟩;

upon event ⟨ *al*, *Deliver* | p, [READ, r] ⟩ **do**
trigger ⟨ *al*, *Send* | p, [VALUE, r, ts, val, σ] ⟩;

upon event ⟨ *al*, *Deliver* | q, [VALUE, r, ts', v', σ'] ⟩ **such that** $r = rid$ **do**
if $verifysig(q, bonrr \| w \| \text{WRITE} \| ts' \| v', \sigma')$ **then**
$readlist[q] := (ts', v')$;
if $\#(readlist) > \frac{N+f}{2}$ **then**
$v :=$ highestval($readlist$);
$readlist := [\perp]^N$;
trigger ⟨ *bonrr*, *ReadReturn* | v ⟩;

i.e. > 2f

In the **write event** we can see that the *sender* will *encrypt* the *message* and will be sent to all the *processes*. The *quorum*, since we use some *cryptography mechanisms*, is bigger than $\frac{N+f}{2}$ so we have to assume than $N > 3f$. When a *process* want to **read** some values, it will trigger the *read request* to all the *processes*, that they will respond with them *variable encrypted* and the *timestamp*. So the *reader* will check if the **signature** is correct and in this case will add the message to its *readlist*, and when this is bigger than the *quorum* will trigger the *readreturn* with the highest value (*timestamp*) in the *readlist*.

16.2.2 Regular Register without cryptography

It's important to note that sometime just one phase for *writing* is not enough cause we can have some cases in which a *reader* is not able to choose a *value*, so we need two phases:

- **Pre-write**: where the *writer* sends $PREWRITE$ messages with the current *value-timestamp pair*, and then it waits until it receives $PREACK$ messages from $N - f$ *processes*;
- **Write**: where the *writer* sends $WRITE$ messages with the current *value-timestamp pair*, and then it waits until it receives ACK messages from $N - f$ *processes*;

So every *process* stores two **timestamp/value pairs**, in this way they can detect *faulty process*:

Algorithm 4.16: Double-Write Byzantine Quorum (part 1, write)

Implements:
$(1, N)$-ByzantineRegularRegister, **instance** *bonrr*, with writer w.
Uses:
AuthPerfectPointToPointLinks, **instance** *al*.

upon event ⟨ *bonrr*, *Init* ⟩ **do**
$(pts, pval) := (0, \perp)$;
$(ts, val) := (0, \perp)$;
$(wts, wval) := (0, \perp)$;
preacklist := $[\perp]^N$;
acklist := $[\perp]^N$;
rid := 0;
readlist := $[\perp]^N$;

upon event ⟨ *bonrr*, *Write* | v ⟩ **do** // only process w
$(wts, wval) := (wts + 1, v)$;
preacklist := $[\perp]^N$;
acklist := $[\perp]^N$;
forall $q \in \Pi$ **do**
trigger ⟨ *al*, *Send* | q, [PREWRITE, $wts, wval$] ⟩;

upon event ⟨ *al*, *Deliver* | p, [PREWRITE, $pts', pval'$] ⟩
such that $p = w \wedge pts' = pts + 1$ **do**
$(pts, pval) := (pts', pval')$;
trigger ⟨ *al*, *Send* | p, [PREACK, pts] ⟩;

upon event ⟨ *al*, *Deliver* | q, [PREACK, pts'] ⟩ **such that** $pts' = wts$ **do**
preacklist[q] := PREACK;
if $\#(preacklist) \geq N - f$ **then**
preacklist := $[\perp]^N$;
forall $q \in \Pi$ **do**
trigger ⟨ *al*, *Send* | q, [WRITE, $wts, wval$] ⟩;

upon event ⟨ *al*, *Deliver* | p, [WRITE, ts', val'] ⟩
such that $p = w \wedge ts' = pts \wedge ts' > ts$ **do**
$(ts, val) := (ts', val')$;
trigger ⟨ *al*, *Send* | p, [ACK, ts] ⟩;

upon event ⟨ *al*, *Deliver* | q, [ACK, ts'] ⟩ **such that** $ts' = wts$ **do**
acklist[q] := ACK;
if $\#(acklist) \geq N - f$ **then**
acklist := $[\perp]^N$;
trigger ⟨ *bonrr*, *WriteReturn* ⟩;

upon event ⟨ *bonrr*, *Read* ⟩ **do**
rid := *rid* + 1;
readlist := $[\perp]^N$;
forall $q \in \Pi$ **do**
trigger ⟨ *al*, *Send* | q, [READ, *rid*] ⟩;

upon event ⟨ *al*, *Deliver* | p, [READ, r] ⟩ **do**
trigger ⟨ *al*, *Send* | p, [VALUE, $r, pts, pval, ts, val$] ⟩;

upon event ⟨ *al*, *Deliver* | q, [VALUE, $r, pts', pval', ts', val'$] ⟩ **such that** $r = rid$ **do**
if $pts' = ts' + 1 \vee (pts', pval') = (ts', val')$ **then**
readlist[q] := $(pts', pval', ts', val')$;
if exists (ts, v) in an entry of *readlist* **such that** *authentic*$(ts, v, readlist)$ = TRUE
and exists $Q \subseteq readlist$ **such that**
$\#(Q) > \frac{N+f}{2} \wedge$ *selectedmax*(ts, v, Q) = TRUE **then**
readlist := $[\perp]^N$;
trigger ⟨ *bonrr*, *ReadReturn* | v ⟩;
else
trigger ⟨ *al*, *Send* | q, [READ, r] ⟩;

TRUE if *readlist* contains a pair (ts,v) that is found in the entries of more than f processes

readlist contains a pair (ts,v) such there is a Byzantine quorum (Q) of entries in *readlist* whose *highest* timestamp/value pair, selected among the pre-written or written pair of the entries, is (ts, v)

The **write event** is same as usual with a *quorum* of $N - f$ and with two phases of *write* and *pre-write*. When a *reader* want to **read** something will send a *request*, and all the *processes* will respond with a *message* that contains **two pairs**, one pair *value-timestamp* of the *pre-write*, and the same pair for the *write*. So the *reader* will check if these two pairs are equal, or if the *timestamp* of the the *pre-write* is bigger of one unit than the *timestamp* of the *write*, so for this condition we

have the case in which that *process* received the *pre-write* but not the *write*. After the *reader* will check if there are *tuple* than are more than f *byzantine processes* and if there exist a *subset* of the *readlist* for which the *quorum* composed by the **highest timestamps** (one of the two *timestamps* bigger) is reached. If the *quorum* is not reached then the *read request* is *re-sended*.

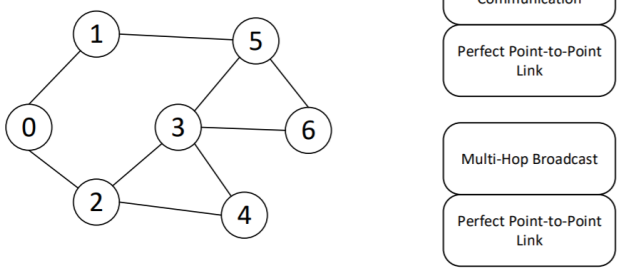
So the *termination property* must be relaxed in to **finite-write termination**. Instead of requiring that every *operation* of a *correct process* eventually terminates, a *read operation* that is **concurrent** with *infinitely* many *write operations* may not terminate.

17 Byzantine Tolerant Broadcast, Multi-Hop Networks

The **Distributed System** is an *abstraction*, in which we have, a set of spatially separate *entities*, each of these with a certain *computational power*, that are able to *communicate* and to *coordinate* among themselves for reaching a common goal. It's defined by a set of assumptions like:

- **Process assumptions**: all *correct processes*, *crash failures*, *crash with recovery*, *byzantine*, etc...;
- **Link assumptions**: *Fair-loss*, *Perfect link*, etc...;

And we have different **Communication Network**, or *Network Topology*, that shows how *processes* are interconnected between each other, so defines the set of *processes* which can directly exchange *messages*. The *Perfect Point-to-Point Link* is a *link* in which we have *no duplication*, *no creation*, and *reliable delivery*, that means that if a *correct process* p sends a *message* m to a *correct process* q then q eventually *delivers* m. In the **Multi-Hop** instead we have that a *message* may cross *hop* to reach its *destination*. So the *communication* and the *broadcast* is based on the *PP2P*:



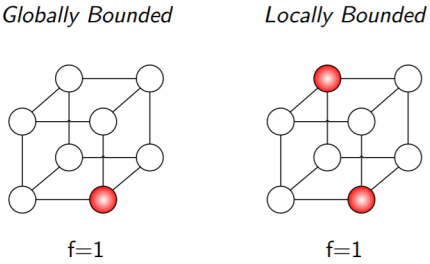
In order to transmit a *message* in a **multi-hop network** there are three different ways:

- **Flooding**: in which the *message* is sent to all the *processes* of the *network*;
- **Routing**: in which the *message* is sent by using a *spanning tree*, so we have a *route* in order to reach a *destination process*;
- **Gossip**: that uses *probabilistic*;

Since we can have some **byzantine processes** that can block or change the *message* in the *multi-hop network* we need to implement a *byzantine reliable broadcast*, in which for *Complete Communication Network*, we have *validity*, *no duplication*, *integrity* and *totality*, and instead for *Multi-Hop Network*, we need to have a *correct source*, *safety* (*integrity*) and *liveness* (*validity*).

We can distinguish two different types of *failure models*:

- **Globally Bounded**: that counts the total number of *byzantine processes* in whole system;
- **Locally Bounded**: that counts the number of *byzantine processes* to which every other *correct process* is directly connected;



17.1 Globally Bounded Failure Model, Multi-Hop network

So with **Globally Bounded Failure Model** we have:

- *n processes*;
- *Not-complete communication network*;
- *Processes* can be *correct* or *byzantine*;
- We can have *f faulty processes*;
- *Processes* have no global knowledge (so they don't know about each other);
- *Authenticated Perfect Channels*, in which we have *reliable delivery*, *no duplication*, *no creation* and *authenticity*;

A *graph* is called $k - connected$ if and only if it contains k *independent path* between any two *vertexes*, and the *menger theorem* (**Vertex cut - Disjoint path**), says that minimum number of *vertexes* separating two *nodes* p and q is equal to the maximum number of *disjoint* $p - q$ *paths* in the *graph*, so this means that is equal to minimum number of *node* that we need to delete from the *graph* in order to get a *min-cut*.

We can use this theorem, since we know there are at most f *byzantine processes* over the *system*, so if a *message* comes from $f + 1$ *disjoint paths* it can be *safely accepted*.

17.1.1 Dolev's Algorithm

In order to verify it we can use the **Dolev's Algorithm**, where the idea is to leverage the *authenticated channels* to collect the *ID's* of the *processes* traversed by a *message*, so the format will be: $(source, content, traversed\_processes)$. This *algorithm* is divided in two parts:

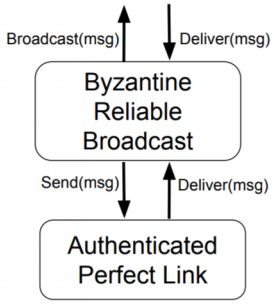
- **Propagation Algorithm**:
  - The *source process* s sends the *message* msg to all of its *neighbors* $msg := \langle s, m, \varnothing \rangle$;
  - A *correct process* p saves and relays the *message* sent by a *neighbor* q to all other *neighbors* not included in the $traversed\_processes$ appending to it the id of q so: $msg := \langle s, m, traversed\_processes \cup q \rangle$;
- **Verification Algorithm**:
  - If a *process* receives copies of msg carrying the same m and s where it is possible to identify $f+1$ *disjoint path* (from s to current node) among $traversed\_processes$ then m is delivered by the *process*;

```
Implements:
       ByzantineReliableBroadcast, instance brb, with sender s.
Uses:
       AuthPerfectPointToPointLinks, instance al.
       Asynchronous System

upon event ⟨ brb, Init ⟩ do
    received := Ø
    delivered := Ø

upon event ⟨ brb, Broadcast | ⟨p,m⟩ ⟩ do
    forall q ∈ Neighbors_p do
        trigger ⟨ al, Send | q, [ ⟨p,m⟩ ,Ø] ⟩;

upon event ⟨ al, Deliver | q̂, [ ⟨s,m⟩ , tp] ⟩ do
    new_tp := tp ∪ q̂
    if q̂ == s then
        trigger ⟨ brb, Deliver | ⟨s,m⟩ ⟩;
        delivered := delivered ∪ ⟨s,m⟩
    else
        received[ ⟨s,m⟩ ] := received[ ⟨s,m⟩ ] ∪ new_tp
        if ⟨s,m⟩ ∉ delivered and check_disjoint_paths(f+1, received[ ⟨s,m⟩ ]) == True then
            trigger ⟨ brb, Deliver | ⟨s,m⟩ ⟩;
            delivered := delivered ∪ ⟨s,m⟩
    forall q ∈ Neighbors_p such that q ∉ new_tp do
        trigger ⟨ al, Send | q, [ ⟨s,m⟩ , new_tp] ⟩;
```



Since all the *messages* generated by a *byzantine process* are labeled with its *ID*, it is not possible to generate $traversed\_processes$ with **minimum cut** greater than f, so the **dolev algorithm** enforces **safety**. Instead the **liveness** depends on the *network topology*.

The **byzantine realible broadcast** can achieved in a *static network* G composed of n *processes* and f *byzantine processes*, if and only if the **vertex connectivity** of G is at least $2f + 1$. The **vertex connectivity** of a graph is the minimum number of *nodes* whose *deletion* disconnects it.

We have two types of **complexity**:

- **Message Complexity**: exponential in the number of *processes*;
- **Delivery Complexity**: solve an **NP-complete problem**;

So, it's not practically employable, and we can have even another problem: a *byzantine processes* can start *flooding* the *network* with lots of *messages*, so we have *denial of service* problem and so no *liveness*. A possible solution is to restrict the *capability* of every *process*, so we can use a **Bounded Channel Capacity**, where every *process* can send only a bounded number of *messages* in a time window, so in this way we decrease the input for the *NP-complete problem*.

The **Dolev algorithms** relays any *message* to any *process* not already included in $traversal_processes$ so we can optimize it:

- Optimization 1:
  - If a *process* p has delivered a *message* msg than p can relay msg with an empty $traversed\_processes$ without affecting the *safety property*;
- Optimization 2:
  - A *process* p has not to relay *messages* carrying m to *processes* q_i that have already delivered m;

```
Optimized Dolev Byzantine Reliable Broadcast

Implements:
       ByzantineReliableBroadcast, instance brb, with sender s.
Uses:
       AuthPerfectPointToPointLinks, instance al.
       Synchronous System

upon event ⟨ brb, Init ⟩ do
    received := Ø
    delivered := Ø
    neight_del := Ø

upon event ⟨ brb, Broadcast | ⟨p,m⟩ ⟩ do
    forall q ∈ Neighbors_p do
        trigger ⟨ al , Send | q, [ ⟨p,m⟩ ,Ø] ⟩;

upon event ⟨ al, Deliver | q̂, [ ⟨s,m⟩ , tp] ⟩ do
    if q̂ == s then
        trigger ⟨ brb, Deliver | ⟨s,m⟩ ⟩;
        delivered := delivered ∪ ⟨s,m⟩
        received[ ⟨s,m⟩ ] := received[ ⟨s,m⟩ ] ∪ Ø
    else
        if tp = Ø then
            neight_del[ ⟨s,m⟩ ] := neight_del[ ⟨s,m⟩ ] ∪ q̂
        new_tp := tp ∪ q̂
        received[ ⟨s,m⟩ ] :=  received[ ⟨s,m⟩ ] ∪ new_tp
        if ⟨s,m⟩ ∉ delivered and check_disjoint_paths(f+1, received[ ⟨s,m⟩ ]) == True then
            trigger ⟨ brb, Deliver | ⟨s,m⟩ ⟩;
            delivered := delivered ∪ ⟨s,m⟩
            received[ ⟨s,m⟩ ].clear()
            received[ ⟨s,m⟩ ] := received[ ⟨s,m⟩ ] ∪ Ø

upon exists ⟨s,m⟩ in received do
    [s,m,tp] := received[ ⟨s,m⟩ ].random_pop()
    forall q ∈ Neighbors_p - neight_del[ ⟨s,m⟩ ] such that q ∉ tp and q != s do
        trigger ⟨ al , Send | q, [ ⟨s,m⟩ , tp] ⟩;
```

On **asynchronous systems**, the *message complexity* is still *exponential*, instead in **Synchronous Systems** with specific topologies it has been show that they are very effective.

In case we have a **routed network**, so with fixed routes between every pair of *processes*, we have that the *source* broadcasts a *message* along $2f + 1$ *disjoint routes*, so any other *process* relays a *message* only if it respects the *planned routes*. So we need at least $2f+1$ *vertex connected network*, and we have a *Quadratic Message Complexity* (every *edge* is traversed once), and a *Linear Delivery Complexity* (counting the copies of message).

If we have instead a **Digital Signature**, the *source* will digitally signs a *message* to broadcast,

and any *correct process* just relays the received *messages*, in this case we need at least $f+1$ *vertex connected network*.

17.2 Locally Bounded Failure Model, Multi-Hop network

With **Locally Bounded Failure Model** we have:

- n *processes*;
- *Not-complete communication network*;
- *Processes* can be *correct* or *byzantine*;
- We can have f *faulty processes* in the *neighborhood* of every *process*;
- *Processes* have no global knowledge (so they don't know about each other);
- *Authenticated Perfect Channels*;

17.2.1 CPA Algorithm

The idea is that since we have at most f *faulty process* in every *neighborhood* a *process* waits for $f+1$ copies of the same *message* in order to *deliver*. So the *source* will *broadcast* the *message*, and all the *neighbors* of the *source* will directly *accept* and *relays* the *message*, instead when a *process* receives the *message* from $f+1$ *distinct neighbors* will *accepts* and *relays* the *message*.

So every *process* relays a *message* only if it has been *delivered*, and since at most f *faulty process* are present in the neighborhood the *CPA algorithm* enforces **safety**, instead **liveness** depends on *network topology*. The **Message Complexity** is *quadratic* (since every *edge* is traversed once), and the **Delivery Complexity** is *linear* (counting the copies of a *message*).

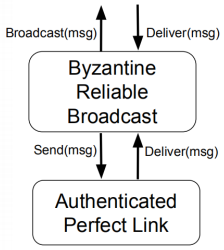
```
Certified Propagation Algorithm (CPA) Byzantine Reliable Broadcast

Implements:
      ByzantineReliableBroadcast, instance brb, with sender s.
Uses:
      AuthPerfectPointToPointLinks, instance al.
      Asynchronous System

upon event ⟨ brb, Init ⟩ do
      received := Ø
      delivered := Ø

upon event ⟨ brb, Broadcast | ⟨p,m⟩ ⟩ do
      forall q ∈ Neighbors_p do
            trigger ⟨ al, Send | q, ⟨p,m⟩ ⟩;

upon event ⟨ al, Deliver | q̂, ⟨s,m⟩ ⟩ such that ⟨s,m⟩ ∉ delivered do
      if q̂ == s then
            trigger ⟨ brb, Deliver | ⟨s,m⟩ ⟩;
            delivered := delivered ∪ ⟨s,m⟩
      else
            received[ ⟨s,m⟩ ] := received[ ⟨s,m⟩ ] ∪ p
            if |received[ ⟨s,m⟩ ]| > f then
                  trigger ⟨ brb, Deliver | ⟨s,m⟩ ⟩;
                  delivered := delivered ∪ ⟨s,m⟩
            forall q ∈ Neighbors_p do
                  trigger ⟨ al, Send | q, ⟨s,m⟩ ⟩;
```



17.2.2 MKLO

MKLO is the partition of *nodes* in *levels*, in which the *source* is placed in L_0, and the *neighbors* of the *source* are in L_1, instead any other *level* is placed in the first level such that it has at least k *neighbors* in the previous levels. If we use MKLO the CPA algorithm liveness is guaranteed. So the **Correctness** is:

- *Necessary condition*: MKLO with $k = f + 1$;
- *Sufficient condition*: MKLO with $k = 2f + 1$;

And the *strict condition* is that MKLO with $f+1$ removing any possible placement of the *byzantine processes*;

17.3 Byzantine Reliable Broadcast in Planar Networks

A **Planar Graph** is a *graph* in which *edges* do not cross and in which Z is the maximum number of *edges* per **polygon** and D is the minimum number of *nodes* between two *byzantine nodes*. It is possible to achieve *Byzaint Reliable Broadcast* in a *4-connected planar graph* if and only if $D > Z$. The implementation is:

- Every *process* saves in $Rec(q)$ the last *message* received from a neighbor q;
- The *source* s *multicasts* an information m;
- The *neighbors* of the *source* wait until they receive m from s then they *deliver* m and *multicast* $\langle s, m, \varnothing \rangle$;
- When $\langle s, m, S \rangle$ is received from a neighbor q with $q \notin S$ and $|S| \leq Z - 3$ then $Rec(q) := \langle s, m, S \rangle$ and multicast $\langle s, m, S \cup \{q\} \rangle$
- When $\exists m, p, q, S$ such that $q \neq p, q \neq S, Rec(q) = \langle s, m, \varnothing \rangle$ and $Rec(p) = \langle s, m, S \rangle$ then deliver m and multicast $\langle s, m, \varnothing \rangle$ and stop;

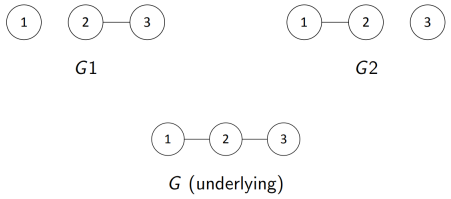
Every *process* keeps a copy of the last *message* received from every *neighbors*, so **linear memory** is required on every *process* so we need at most 1 *byzantine* arbitrarily place or *spatial condtion* $D > 4$.

18 Byzantine Tolerant Broadcast, Dynamic Networks

In **Dynamic Networks** we have continuously changes like the set of *processes* that composes the *system* (*Churns*) and also the *communication network*. In this system *processes* can directly exchange *messages* with a subset of all *processes*, and this subset can change over the time and *processes* can be isolated for a while. So we need a *model* that changes over time, and one of the most general is the **TVG model**.

18.1 TVG Model

The **TVG model** or **Time Varying Graph** is a *Graph* $G := (V, E, \rho, \zeta)$ where V is the set of *nodes*, E the set of *edges*, ρ is the **presence function** (a true/false *function* that indicate if there is that *edge* or not), ζ is the **latency function** (a *function* that given an *arch* return the latency). G can also be described as a sequence of *static graphs* (snapshots). The **underlying graph** is the *graph* on which all the *snapshots graphs* are based on:



A sequence of distinct nodes $(p_1, \ ... \ , p_n)$ is a **Journey**, or *Dynamic Path*, from p_1 to p_n if there exists a sequence of dates $(t_1, \ ... \ , t_n)$ such that $\forall \ i \in \{1, \ ... \ , n-1\}$ we have:

- $e_i = (p_i, p_{i+1}) \in E$, so there is an edge connecting p_i to p_{i+1};
- $\forall \ t \in [t_i, t_i + \zeta(e_i, t_i)] \, , \rho(e_i, t) = 1$, so p_i can send a message to p_{i+1} at date t_i;
- $\zeta(e_i, t_i) \leq t_{i+1} - t_i$ the aforementioned message is received by data t_{i+1};

There are different *classes* of **TVG**:

- Class 1, **Temporal Source**: $\exists u \in V : \forall v \in V, u \rightarrow v$, broadcast feasible from at least one node;
- Class 2, **Temporal Sink**: $\exists u \in V : \forall v \in V, v \rightarrow u$, function whose input is spread over all nodes;
- Class 3, **Connectivity over time**: $\exists u, v \in V, v \rightarrow u$, every node can reach all other;
- Class 5, **Recurrent connectivity**: $\forall v, u \in V, \forall t \in \tau, \exists \jmath \in j^*_{u,v}, departure(\jmath) > t$, routing can always be achieved;
- Class 6, **Recurrence of edges**: $\forall e \in E, \forall t \in \tau, \exists t^{'} > t, \rho(e, t^{'}) = 1$ and G is connected;
- Class 7, **Time-bounded recurrence of edges**: $\forall e \in E, \forall t \in \tau, \exists t^{'} \in [t, t + \delta] \, , \rho(e, t^{'}) = 1$ and G is connected;
- Class 8, **Periodicity of edges**: $\forall e \in E, \forall t \in \tau, \forall k \in \mathbb{N}, \rho(e, t + kp)$ some $p \in \mathbb{T}$ and G is connected;
- Class 10, **T-interval connectivity**: $\forall i \in \mathbb{N}, \forall T \in \mathbb{N}, \exists G^{'} \subseteq G : V_{G^{'}} = V_G$, where $G^{'}$ is connected and $\forall j \in [i, i + T - 1] \, , G^{'} \subseteq G_j$;

The **Broadcast latency** on **1-interval connected networks** is $O(n)$ so $\forall t$ at least one not *informed node* is connected to an *informed node*, so the *complexity* is equal to the *static multi-hop network*. The **Byzantine Reliable Broadcast Specification** is:

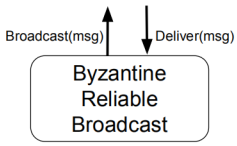
Module:

Name: Byzantine Reliable Broadcast, **instance** *brb*, with source *s*.

Events:

Request: $\langle brb, Broadcast|m\rangle$: Broadcasts a message m to all processes. Executed only by process s.

Indication: $\langle brb, Deliver|p, m\rangle$: Delivers a message m broadcast by p.



Properties:

RB1: *Safety*: If some correct process delivers a message m with source p and process p is correct, then m was previously broadcast by p.

RB2: *Liveness*: If a correct process p broadcasts a message m, then every correct process eventually delivers m.

Consistency and *Totality* of Byzantine Reliable Broadcast for complete network are not considered.

Like for the *multi-hop network* we can have two different types of *failure models*, **globally bounded** and **locally bounded**.

18.2 Globally Bounded Failure Model, Dynamic network

- n *processes*;
- Dynamic Communication Network, TVG;
- *Processes* can be *correct* or *byzantine*;
- We can have f *faulty processes*;
- *Processes* have no global knowledge (so they don't know about each other);
- *Authenticated Perfect Channels*;

Unlike the *multi-hop network* in which *Vertex Cut* = *Disjoint Paths*, in **dynamic network** the *Vertex Cut* $\geq$ *Disjoint Paths*. So, the idea is to extend the *Dolev Algorithm*, by using the same *propagation algorithm*, but we check for a **dynamic min-cut** with size $f + 1$ and in which every *message* is re-transmitted every time a *process* detects a *network change* in its neighborhood.

The *Correctness* is guaranteed by the fact that *byzantine processes* can not generate *traversed_processes* with *minimum cut* lower than f, *liveness* instead between two endpoints, the *byzantine reliable communication* from process p to q, is achievable if and only if the **dynamic minimum cut** between p and q is at least $2f + 1$. The *complexity* is the same of the *Dolev* one, exponential for *message complexity*, NP-complete for *delivery complexity*.

For *static distributed systems* the *vertex connectivity* of a graph can *polynomial* verified through a *max-flow algorithm*, instead in *dynamic distributed systems* the computation of a **dynamic min-cut** is a NP-complete problem.

18.3 Locally Bounded Failure Model, Dynamic network

- *n processes*;
- Dynamic Communication Network, TVG;
- *Processes* can be *correct* or *byzantine*;
- We can have *f faulty processes* in the neighborhood of every process;
- *Processes* have no global knowledge (so they don't know about each other);
- *Authenticated Perfect Channels*;

We can use the **CPA algorithm** on **Dynamic Distributed Systems** without further changes. The **safety property** is still guaranteed, in fact every *process* relays a *message* only if it has been *delivered*, at most *f faulty process* are present in the *neighborhood*. The **liveness property** in *static networks* requires the existence of a *partition MKLO* to be guaranteed. In *dynamic networks* a *MKLO* is not enough, in fact an *edge* may disappear while transmitting a *message* and the order of appearance of *edges* matters, so we have to use the **TMKLO** that is the **temporal minimum k-level ordering**:

$$RCD + MKLO$$

Where **RCD** is the *edge* appearances that allow to *reliably deliver* a *message* transmitted over a channel. and the *necessary condition* with *TMKLO* is $k = f + 1$ instead the *sufficient condition* is $k = 2f + 1$, and the computation of the *TMKLO* is done only if we have a full *TVG* knowledge.

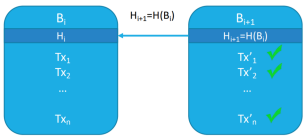
19 DLT & Blockchain

A **Ledger** is a written or *computerized record* of all the *transactions* a business has completed. A *Distributed Ledger* is a *database* replicated across several locations or among multiple participants used to store *record* of *transactions*. The *key properties* of a *Distributed Ledger* are *Consistency*, *Integrity* and *Availability*.

A **Blockchain** is a *decentralized, distributed data structure* used to record *transactions* (aggregated in **blocks**) across many computers. Every *Blockchain* represents a *distributed ledger* but the vice versa is not true. A **transaction** is an instance of buying or selling something, so it's an exchange or an *interaction* between *people*. Every *block* in a *Blockchain* contains a set of *transaction*, where the clients generate *transactions* and submit them to *nodes* implementing the *Blockchain*. *Transactions* are collected by *processes*, so they need to be **valid** in order to respect the *ledger specification*, so they remain *unconfirmed* until they are inserted in a *block* that is successfully attached to the chain, so when enough *transactions* have been collected a *block* is created and propagated in to the *network* to be attached to the chain.

In order to **validate** a *transaction*, *nodes* need to read the *last state* of the leader and check if the *current transaction* is correct with respect to the *semantics* of the *ledger*, so **ordering** the *transactions* in the *blocks* is fundamental to check and guarantee the *validity* of the *ledger*. If we want to attach a *block* to the chain we need to:

- check if all the *transactions* in the *block* are *valid*;
- compute the *hash* $H()$ of the last *block* attached to the *chain*;
- include this *hash* in the *current block* to generate a *pointer* and attach the *current block*;
- we can have some problems like *concurrency* in the *block creation process* and *asynchronous systems*.



So we need **consensus** between *nodes* in order to attach a new *block* to the *Blockchain*. The *Blockchain* can be classified in base of:

- **How is accessing data**:
  - **Public**: so no restrictions on reading *blockchain data* and submitting *transactions* for inclusion into the *blockchain*;
  - **Private**: direct access to *blockchain data* and *submitting transactions* is limited to a predefined list of entities;
- **How is managing data**:
  - **Permissioned**: *transaction processing* is performed by a predefined list of subjects with known identities;
  - **Permissionless**: there aren't restrictions on identities of *transaction processors*;

So for the *consensus* there are two ways in base of what *Blockchain* we have, for a **Public Permissionless Blockchain** (in which every *process* can access *data*, submit *transactions* and *process* them) we will use the **Proof of X**, instead for a **Private Permissioned Blockchain** (in which only *authorized processes* can access *data*, submit *transactions* and *process* them) we will use the **BFT Consensus**.

19.1 Private Permissioned Blockchain

In **private permissioned blockchain** we have that:

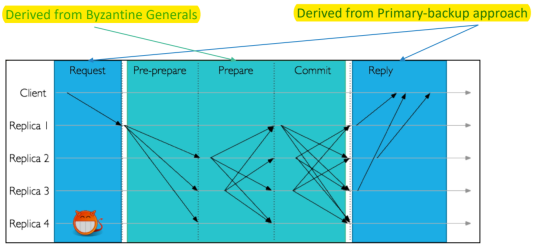
- **Transactions** are submitted only by known *clients* and only known *clients* can access it;
- **Blocks** are created and attached by known and *authorized processes*;

PBFT or **Practical Byzantine Fault Tolerance** is the current adopted *protocol*, designed to support *replication* and merges the *primary-backup approach* with the *Byzantine General Consensus approach*. So in our **system model** we have an *asynchronous distributed system*, where *nodes* are connected by *network* and *messages* can be lost, duplicated, or no order, and we can have some *node*

failures (*byzantine*). We will use come *Cryptographic techniques* like *public key signatures, MAC*, in order to avoid *spoofing* or attack and all *replicas* know each others *public keys* to verify *signatures*. A **strong adversary** can coordinate the action of *faulty nodes*, and can *delay communication* and *delay correct nodes* (not indefinitely) but cannot subvert the *cryptographic techniques* used. We have two properties:

- **Safety**:
  - *Satisfies Linearizability*, like a *centralized system* that executes operations atomically one at a time;
  - Regardless of *faulty clients*, but they can do strange (legal) operations;
  - We can limit damage by *access control mechanisms*;
- **Liveness**:
  - *Clients* receive *responses* to their *request* if at most $\frac{n-1}{3}$ *replicas* are *faulty* and *delay*(t) doesn't grow faster than t indefinitely;

The *protocol* is divided in five phases: **request, pre-prepare, prepare, commit and reply**.



Like the **Byzatine Consensus**, all the *replicas* will exchange *messages* between them until a *majority* on the *block* that will be added to the chain. The basic idea is that a *client* sends a *request* to invoke an operation to the *primary*, the *primary* will *multicasts* the *request* to the *backups*, *replicas* execute the *request* and send a *reply* to the *client*, so the *client* waits for $f+1$ replies from different *replicas* with the same result. If the *primary* is *not correct* it is substituted with a new one as soon as a misbehavior is detected. The *system lifetime* is characterized by a series of **views** (*numbered*) each with a different *primary*.

During the **Request Phase**: the *client* c requests the execution of operation o issuing the request: $\langle Request, o, t, c \rangle_{\sigma c}$, that is sent to the *last primary* known by the *client* (if we don't receive a *response*, after a timeout the *request* is multicast to all replicas) a timestamp t is used to guarantee exactly-once semantic. All the replica have an **internal state**: with an *id* i (between 0 and $N-1$) a *service state*, a *view number* v and a *log* of accepted messages.

In the **Pre-Prepare Phase**: the *primary* p will send to all *backups* the message $\left\langle \langle Pre\_Prepare, c, n, d \rangle_{\sigma p}, m \right\rangle$, where c is the *client id*, n a *sequence number* for the current *view* assigned by the *primary*, d is a *digest* of the *client request*, and m is the *client request*. So the *backup* will accept the *pre-prepare message* (by putting it into the log) only if: the *signature* of the two messages (*pre prepare* and *request*) are verified and d is the *digest* of m, it is currently in *view* v, it has not accepted a *pre-prepare*

message for *view* v and *sequence number* n containing a different *digest*, the sequence number n is between a *low watermark* h and a *high watermark* H.

In the **Prepare Phase**: a *backup* i will send to all *nodes* the *message* $\langle Prepare, v, n, d, i\rangle_{\sigma i}$, and another *node* will accept it only if, the *signature* are correct, if it is currently in *view* v and n is between h and H. So this predicate m, v, n, i is true only if the *replica* i has in its *log*, the *message* m a *pre-prepare* for m and at least $2f$ *prepare messages* from different *backups* that match the *pre-prepare*. So matching among *messages* is done checking v n and d. The *pre-prepare* and *prepare phase* guarantee that *non-faulty replicas* agree on a *total order* of execution for *requests* in a same view, so they ensure that $prepared(m, v, n, i)$ is true then $prepared(m^{'}, v, n, j)$ is false for any *non-faulty replica* j and any $m^{'}$ such that the *digest* of m is different from the *digest* of $m^{'}$, this is due to the *intersection* among *quorums* of *nodes* that accept the *prepare messages*.

In the **Commit Phase**: as soon as $prepared(m, v, n, i)$ becomes true the replica i will send to all the message $\langle Commit, v, n, d, i\rangle_{\sigma i}$, another node will accept a commit messages only if the *signature* are correct, it is in *view* v and n is between h and H. The predicate $commited(m, v, n)$ is true only if $prepared(m, v, n, i)$ is true for all i in a set of $f+1$ *non faulty replicas*. So, the *predicate* $commited-local(m, v, n, u)$ is true only if $prepared(m, v, n, i)$ is true and i has accepted $2f+1$ commits from different *replicas* that match the pre-prepare for m. The *commit phase* guarantees that if $committed-local(m, v, n, i)$ is true for some *non-faulty* i, then $committed(m, v, n)$ is true, because if i received $2f+1$ commits, then $prepared(m, v, n, i)$ is true for at least $f+1$ non-faulty nodes.

In the **Replay Phase**: a *replica* i will execute the request as soon as: $committed-local(m, v, n, i)$ is true and the state of i reflects the sequential execution of all *requests*, the response is then sent to the client $\langle Reply, v, t, c, i, r\rangle_{\sigma i}$, so the client will wait for $f+1$ replies with valid signatures and the same values for t and r.

19.2 Public Permissionless Blockchain

In **public permissionless blockchain** we have that: *transactions* can be submitted by everyone and every *process* can read the *chain*, *blocks* can be created and attached by every *process* in the system. *Public permissioned blockchains* are characterized by *lack of trust* in other *processes* and possibly large scale so *PBFT protocol* doesn't work. The idea is that *processes* start a competition and only the winner can attach its *block* to the *blockchain* so they implement a **randomized leader election**. The **proof-of-work** is a mathematical challenge to *solve* a *block*. **Mining** is finding a number $hash(block) < target$, this *target* is a *threshold* decided apriori by the *system*, and the first *node* who solves a *block* can propose it as next in the *blockchain*, the other *nodes* which receive the *block* will recompute the *hash* to check the *validity*. If two or more *branches* may arrive together we have a *temporal disagree* and in this case the *network* has to converge to the longest one. A miner needs a high *computational power* in order to compute a thousands of *hash* per second.

Transaction rate depends on two parameters:

- **Block Size**: how many *transactions* in a *block*;
- **Block Interval**: time to wait for a *block* to propagate to all the *nodes*;

We have two important *metrics*:

- **Throughput**: how many *transactions* for second;
- **Latency**: how much time for complete a *transaction*;

Increasing the *block size* will improve the *throughput* but the bigger *block* will take longer to propagate in the *network*. Instead decreasing the *block size* will improve the *latency* but will lead the system to *instability* caused by *disagreement*.

Miner is *encouraged* cause solving a *block* gives **coins** to the *node* that found the *proof-of-work*, and some *transactions* can have an *additional fee* given to the *node* who will mines the *block* that containing it. *Rewards* are an incentive for *nodes* to keep them supporting the *blockchain* and to keep *nodes* honest, and its a way to distribute *coins* into circulation.

It's impossible for an **attacker** to change a *transaction* in a specific *block* in the *blockchain*, cause an *attacker* should be quicker than the rest of the whole *network*, and in that case he should be able to *re-mine* $n + 1$ *blocks* (all *blocks* next to the specific *block*) quicker than the rest of the *network*, if so the *attacker* could obtain the *longest* and *modified blockchain* and all *network* would converge to it, but the *attacked* should have the 50% of the *computational power* of the *network*. Since the last *blocks* are less secure (in order to obtain the *longest branch*) an *attacker* should wait for 5 or 6 blocks in order to make an attack successfully and this make its probability too low, so this solution protect **Integrity** and **Double Spending Fraud**.

There is another way to *proof-of-work*, that is called **proof-of-stake**, while *proof-of-work* was *very secure* but waste a lot of *electric power*, **Proof-of-Stake** is secure without *mining* so we don't waste energy. Instead of *mine* a *block* the creator of the next *block* is chosen in a deterministic way according to its *wealth*, and the *reward* are not related to the created block but according to your *wallet*, the longer you keep the *coin* in the *wallet* the more the *reward* is high. The probability of **mint** (instead of *mine*) is proportional to your *wallet*, so *minting* will require a lot of *coin* in order to attack the *network* and it's very hard to *mint* to consecutive *blocks*.

20 Publish/Subscribe Systems

In the **publish/subscribe communication paradigm** we have:

- **Publisher**: who produce data in form of **events**;
- **Subscribers**: declare interests on *published data* with *subscriptions*;
- Each **subscription** is a *filter* on the set of published *events*;
- An **Event Notification Service** (**ENS**) notifies to each *subscriber* every published *event* that matches at least one of its *subscriptions*;

This paradigm is used in various cases, we can have a **many to many communication** in which various *producers* and *consumers* can *communicate* all at the same time, each *information* can be delivered at the same time to various *consumers*, all the *interacting parties* don't know each other, the *information delivery* is mediated through a *third party*, so we don't need *synchronization* between the *interacting parties*.

Events represent information structured following an **event schema**, that is *fixed a-priori* and known to all the *participants*, and it defines a set of *fields* or *attributes*, each constituted by a *name* and a *type*. So given an *event schema*, an *event* is a collection of *values*, one for each *attribute* defined in the *schema*. So we can see it as a *table* in which we have *name*, *type of value* and *allowed values* for each *attribute* of the *event*.

A **subscription** is, generally speaking, a *constraint* expressed on the **event schema**. The **Event Notification Service** will notify an *event* e to a *subscriber* x only if the *values* that define the *event* satisfy the *constraint* defined by one of the *subscriptions* s issued by x and in this case we say that e **matches** s. **Subscriptions** can take various forms, for example it can be a *conjunction* of *constraints* each expressed on a *single attribute* in which each *constraint* can be an $>=<$ *operator* applied on an *integer attribute*, or *complex* as a *regular expression* applied to a *string*.

From an abstract point of view the **event schema** defines an **n-dimensional event space** (where n is the number of *attributes*), in this *space* each *event* e represents a **point** and each *subscription* s identifies a **subspace**, and an *event* e **matches** the *subscription* s iff the corresponding *point* is in the *portion* of the *space* of s.

20.1 Types of Publish/Subscribe

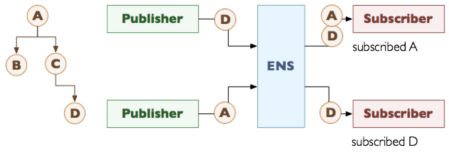
Depending on the **subscription model** used we distinguish various flavors of **publish/subscribe**:

20.1.1 Topic-based selection

In which the *data* published in the *system* is *unstructured* but each *event* is **tagged** with the *identifier* of a **topic** it is published in, so the *subscribers* issue a *subscription* that contains the *topics* they are interested in. So a *topic* can be thus represented as a *virtual channel* connecting *producers* to *consumers*.

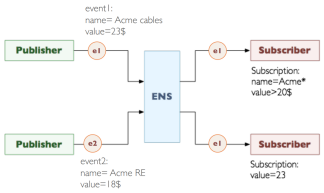
20.1.2 Hierarchy-based selection

In which each *event* is *tagged* with the *topic* it is published in like before, but here *topics* are organized in a **hierarchical structure** which express a notion of *containment* between *topics*. When a *subscriber* subscribe a *topic*, it will receive all the *events* published in that *topic* and in all the *topics* present in the corresponding *sub-tree*.



20.1.3 Content-based selection

In which all the *data* published in the *system* are **mostly structured**, so each *subscription* can be expressed as a **conjunction of constrains** expressed on **attributes**, and the **ENS** will filter out the *useless events* before notifying a *subscriber*.

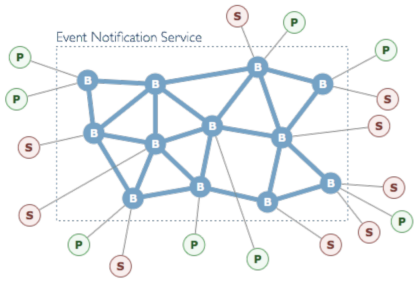


20.2 Event Notification Service

The **Event Notification Service** can be *implemented* in two ways:

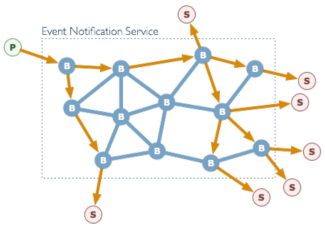
- **Centralized service**: the *ENS* is implemented on a *single server*.
- **Distributed service**: the *ENS* is constituted by a set of *nodes*, called **event brokers**, which *cooperate* to implement the *service*;

The **distributed service** is usually preferred for *large setting* where **scalability** is a *fundamental issue*.



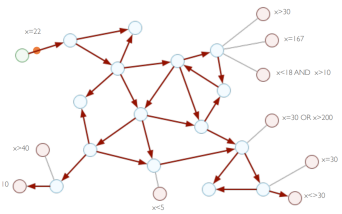
20.3 Event routing mechanism

These **event brokers** forms an *overlay network*, so each *Client* (*publisher* or *subscriber*) accesses the *service* through a *broker* that masks the *system complexity*. We need to introduce an **event routing** mechanism that *routes* each *event* inside the *ENS* from the *broker* where it is *published* to the *brokers* where it must be *notified*.



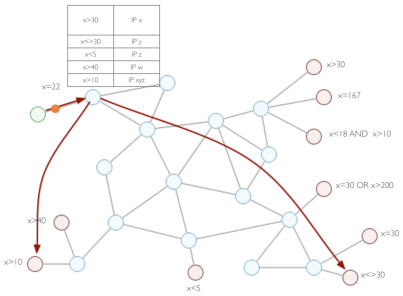
20.3.1 Event Flooding

In which each *event* is **broadcasted** from the *publisher* in whole the *system*, this implementation is *straightforward* but very **expensive**, so we have *highest message overhead* and no *memory overhead*.



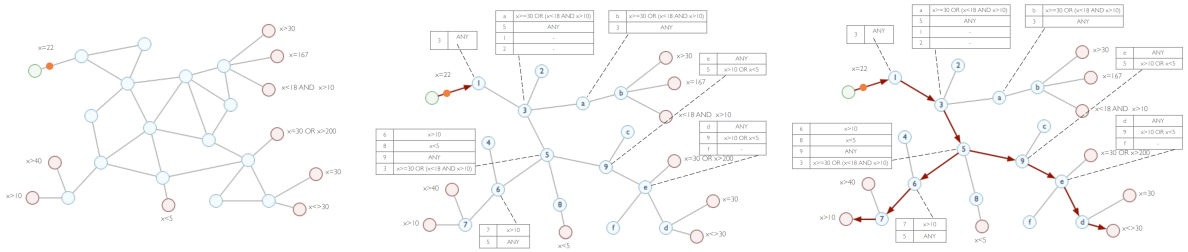
20.3.2 Subscription Flooding

In which each each *subscription* is copied on every *broker*, in order to have in each *broker* a **complete subscription table**, that we will use to *locally match events* and directly notify interested *subscribers*. This approach suffers from a *large memory overhead*, but *event diffusion* is optimal. The problem is that we cannot use this in *applications* where *subscriptions* change frequently.



20.3.3 Filter-based routing

In which *subscriptions* are **partially diffused** in the *system* and used to build **routing tables**, that are then exploited during *event diffusion* to dynamically build a **multi-cast tree** that connects the *publisher* to all the interested *subscribers*. Each *broker* will have the *subscriptions* of the **closest brokers**, so during the *event diffusion* we will go back finding through the *routing tables* all the interested *subscribers*.



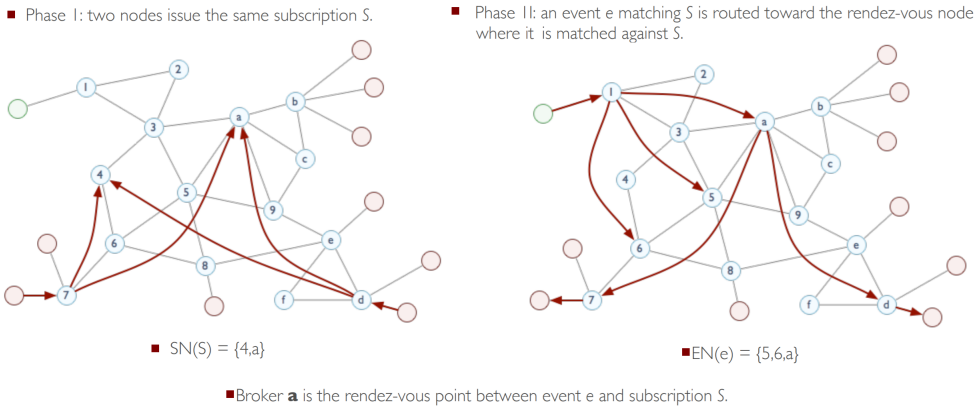
20.3.4 Rendez-Vous routing

The **Rendez-Vous routing** is based on two functions: SN and EN used to associate *subscriptions* and *events* to *brokers* in the *system*. Given a *subscription* s then the function $SN(s)$ returns a *set of nodes* which are responsible for storing s and forwarding **received events** *matching* s to all those *subscribers* that *subscribed* it. Given an *event* e the function $EN(e)$ returns a *set of nodes* which must receive e to match it against the *subscriptions* they store. The goal of this routing is to

obtain the intersection (not empty) between the output of the functions SN and EN. It is divided in two phases:

- **Phase 1**:
  - Given a **subscription** s issued by one or more *subscriber*, the *brokers* that are directly linked to these *subscribers* will ask to the *network* with the function $SN(s)$, in order to get the *set of nodes* that will store the *subscription* s;
- **Phase 2**:
  - When an *event* e that matches s is asked by a *publisher*, all the *nodes* linked to that *publisher* will ask for the *function* $EN(e)$, that will return the *set of nodes* that needs to receive that *message* in order to respect all the *subscriptions* that they contains.

At the end we will check the *nodes* in the **intersection** between the two *sets of nodes* obtained in the two phases, and this *node* will be the **rendez-vous point** between *event* e and subscription s.



20.4 Publish/Subscribe Architecture

In the **publish/subscribe architecture** we have that each *node* has a *service interface* consisting of two **operations**: *send_message*(m) and *set_predicate*(p). A **predicate** is a *disjunction* of *conjunctions* of *constraints* of *individual attributes*. A **content-based network** can be seen as a *dynamically-configurable broadcast network*, where each *message* is treated as a **broadcast message** whose *broadcast tree* is dynamically pruned using *content-based addresses*.

We will have a fusion between **Content-Based routing scheme** and **Broadcast** cause if a *node* need to send an *event* in the *network* he will *broadcast* it to all its *neighbor nodes* but if a *node* is receiving a *message* he will *rebroadcast* it only if all the *conditions* of the *subscriptions* inside him are respected. So the *broadcast* will take care of the **flooding** of the *message* in the *network*, instead the *content based* will create the **sub-tree** used to send the *events* to the correct *subscribers*. So we have two layers, a **Content-based layer** and a **Broadcast Layer**.

The **broadcast layer** is composed by a **broadcast function** available at each *router*, which given

a *source node* s and an *input interface* i will return the set of *output interfaces*. So this *function* defines a **broadcast tree** routed at each *source node* and it satisfy the *all-pairs path symmetry property*.

The **content-based layer** maintains *forwarding state* in the form of a *content-based forwarding table*. The *table*, for each *node*, associates a *content-based address* to each *interface*. So after the *broadcast layer* has initialized a *tree path symmetric* for each *source node*, then the *content-based* will be used in order to use the *forwarding mechanism* to route the *events* from the *source* to the correct *subscriber*.

The *content-based forwarding table* is used by a **forwarding function** F_c that given a *message* m, selects the subset of *interfaces* associated with *predicates* matching m. The result of F_c is then combined with the *broadcast function* B, computed for the *original source* of m. So the *message* is forwarded along the set of *interfaces* returned by: $(B(source(m), incoming_i f(m)) \cup l_0) \cap F_c(m)$, this means we will get the *intersection* between the *nodes* of the *tree* returned by the *broadcast function* and the set of *nodes* returned by the *forwarding function*. The **forwarding tables** maintenance is based on two *mechanism*: **push** that is based on *receiver advertisements*, and **pull** that is based on *sender requests* and *update replies*.

The **Receiver advertisements** are issued by *nodes* periodically and/or when the *node* changes its **local content-based address** p_0. When a *node* **changes** its **advertisements table**, will send to the *nodes* of the *broadcast tree* a *message* of type RA like (n, P) where n is the *identifier* of the *interface* from which comes the *change request* and P is the new *predicate* of the *interface*. When a *node* receives an RA message will do two things:

- First will check if in its own *table* there already exists a *predicate* p_i that contains P of the *interface* n, and in this case the *request* RA is **dropped**;
- Instead if p_i doesn't cover P then the *router* will add the *predicate* to the old predicate and will *rebroadcast* the *message* RA to all the *nodes* near him in the *tree path* of the *broadcast function*.

With the **RA protocol** we can have some *inflation* problems, in which a *node* drops the RA *request* since the *predicate* it's already in its own *table*, but in this way he will not *rebroadcast* the *request*, so these *nodes* cannot update their table.

In the **Sender Requests and Update Replies**, a *router* uses *sender requests* SRs to **pull** *content-based addresses* from all *receivers* in order to update its *routing table*, and the result of an SR come back to the *issuer* through *update replies* URs. The **SR/UR protocol** is designed to complement the RA *protocol*, specifically, it is intended to balance the effect of the *address inflation* caused by RAs, and also to compensate for possible *losses* in the *propagation* of RAs. In the RA *protocol* all the *tables* were updated through a *mono-directional communication* now we have a *two-way communication*. We start from a *node* n that want to *update* the *table* with a message SR, so will ask to all the *nodes* in the *network* their table through the *broadcast function*, so this *request* is *rebroadcasted* to all the *nodes* until it reaches the *leafs* of the *tree* created by the *broadcast function*. These *leafs* will replays with a *message* UR and they will send back their *tables*, so all the *nodes* will *update* their *table* and they will *rebroadcast* the UR *message* until they arrive to the *original node* n.