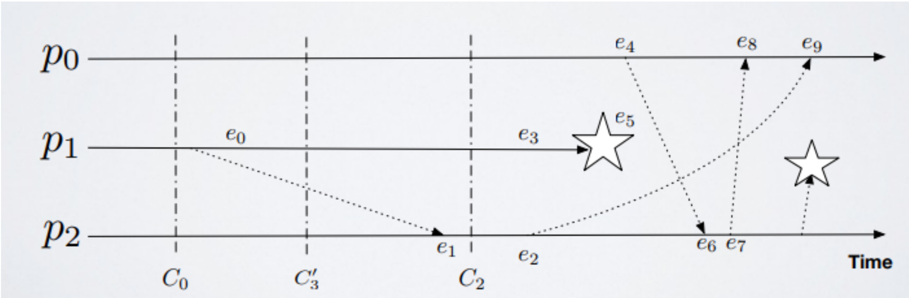
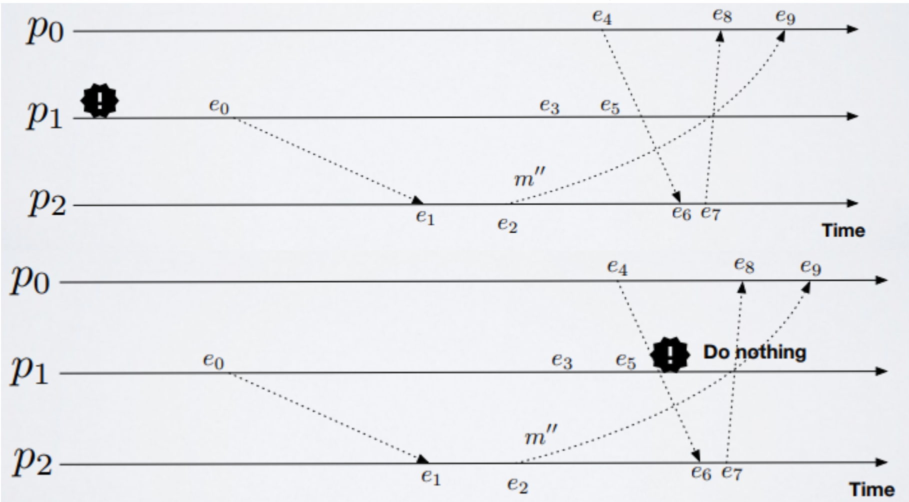
2. FAILURES, ABSTRACTION AND LINK-ALGORITHM

We can have two models for failures

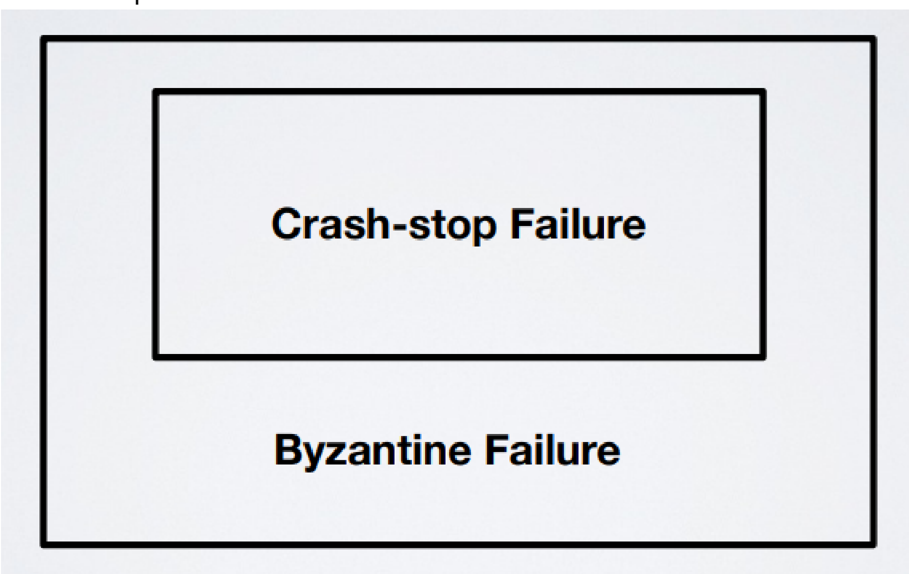
- **Crash-Stop** $Crash(p_j)$: after this event process p_j does not execute any local computation step $Exec(j)$.

  Star means that the process end in that point.
  If a process ends it can't be rebooted, it's died forever.
- **Byzantine Failure** $Byz(p_j)$: after this event process p_j behaves in an arbitrary way ($Exec(j)$ does not follow anymore the automaton of p_j).

  $Byz(p_j) \in Crash(p_j)$: if a byzantine is provided by the automaton then it works even if the crash-stop occurs.

A **process** is **correct** if deos not experience failure. We indicate with f the max number of failures and cannot be more than $n-1$.

ABSTRACTION

Abstraction formalize a problem with a description.
$\rightarrow$ implement the abstraction with a distributed protocol.

Formalize a link (link: comunication channel between 2 processes p and q). If we can implement a link for 2 processes, it can be extendend in an easy way.

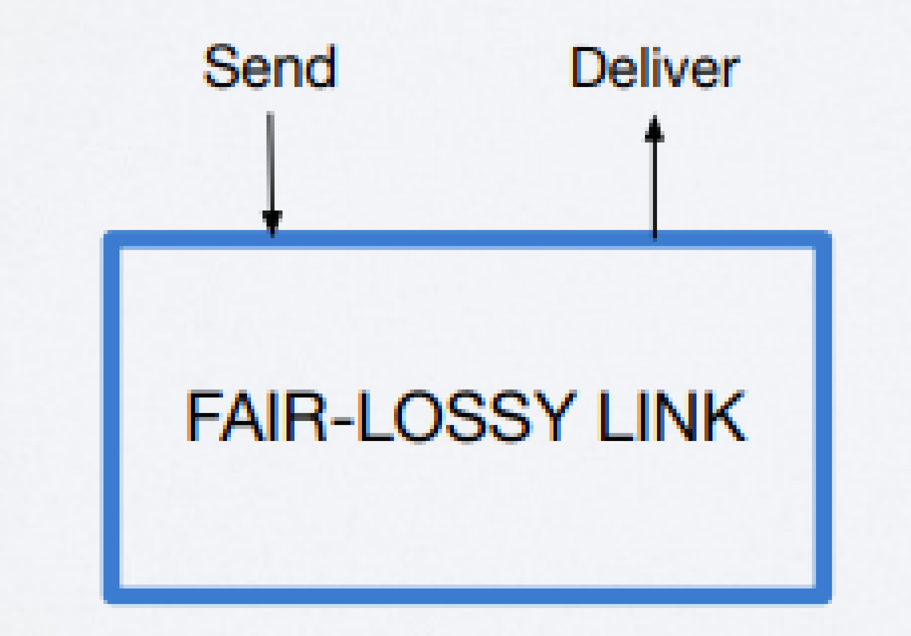
A **link loses messages with a certain probability** pr, the channel can duplicate a message a finite number of times, and it does not crate messages from thin air (if you recive a message, it must be sent by a process not by the link).

link is an object that has 2 events:

- Request (input): $< Send \mid q, m >$ that sends a message m to process q.
- Indication (output): $< Deliver, p, m >$ delivers message m from the process p.

Proprieties:

- **FL1** (fair-loss): if a correct process p sends infinitely often m to a process q, then q deòivers m an infinite number of times. (example, the process faile every even number).
- **FL2** (finite-duplication): if a correct process p sends m a finite number of times of times to q , then q cannot deliver m an infinite number of times.
- **FL3** (no creation): if some process q delivers a mesage m with sender p, then m was sent by p to q.



There are two classes of proprieties: **safety** or **liveness**.

- **Safety**: if violated a time t, it can never be satisfied after t (if a safety propriety is violated in execution E, there is a prefix E' of E such that any extension of E' also violates property example: die). FL3 is a safety propriety.

- **Liveness** cannot be violated in finite executions (any finite executione E thath does not satisfy a liveness property there is an extensione of E that satisfy it)

When you put a bound on a liveness proprety it becomes safety.
If we bound **FL2** with at most 7 duplications (if a correct process p sends m a finite number of times to a, then q cannot deliver m more than 7 times). In that case **FL2 becomes safety**.

Other Properties:

- **Mutual Exclusion**: if a process p is granted a resource r at time t, then no other process q is granted r at t.
- **No-deadlock**: if r is not already granted, eventually someone get a grant on resource r.
- **No-starvation**: if a process p request a grant on resource r, it will eventually get it.

Baddly written propierty (example: if process p sends a message m to q, then q will eventually deliver it and this deliver is unique) $\rightarrow$ it is mixing two aspects:

- a livenesse (q will eventually deliver it)
- a safety: the deliver is unique

It should be decomposed in two propierties:

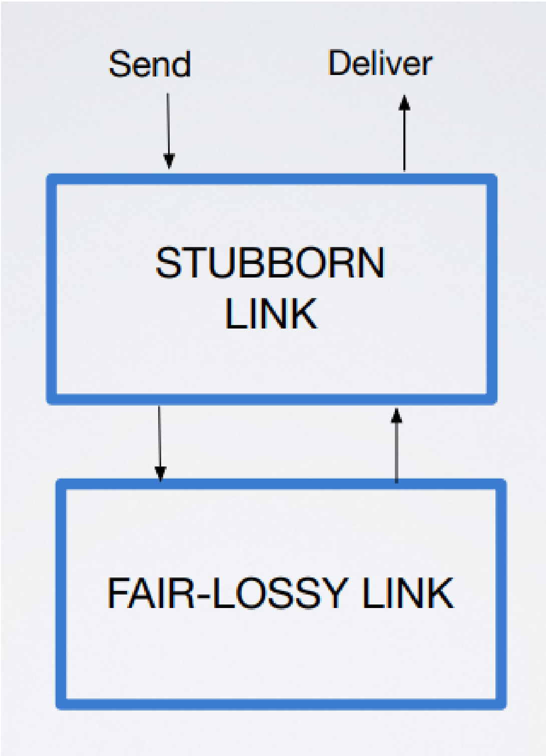
- if p sends a message m to q, then q eventually delivers it.
- if p sends a message m to q then m is delivered at most once.

ALGORITHM - COMMUNICATION LINK

Step 1 - Fair Lossy Link $\rightarrow$ Stubborn Link

Taken **FL1**, **FL2** and **FL3** to get a better algorithm we have to change:

- **FL1**: p sends infinitely often m.
- **FL2**: q cannot deliver m ad infinite number of times.
  that becomes:
- **SL1** (Stubbom-delivery): if a correct process p sends m to q then q delivers m an infinite number of times.
  and we also have:
- **FL3** (No Creation): if some process q delivers a message m with sender p, then m was sent by p to q.



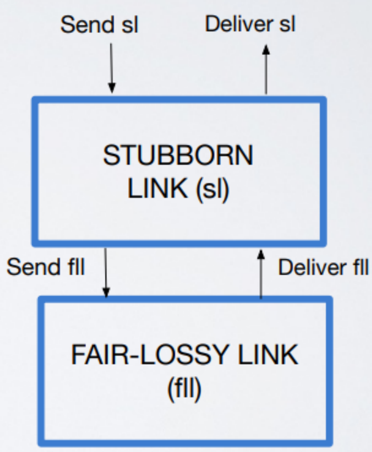
In the pseudocode we can find **handlers** (a function that react when an event happens), they are all atomics. When an hendler starts, the handler take a lock and in the process where the handler is executed, can't happen anything else. So we can't have a round robin execution (zig-zag).

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

upon event ⟨ Timeout ⟩ do
    forall (q, m) ∈ 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 ⟩;
```



But that isn't a correct algorithm because:

1. **FL3** (proof by contradiction): suppose process q executing our algorithm receives message m that was not sent by p.
    1. If q delivers a message, then it delivers here:

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

    Implies that **FLL**, is delivering a message that was not sent by p. This implies that FLL is not fair-lossy. This contradicts our hypothesis: FLL is fair-lossy.
2. **SL1** (proof by contradiction - suppose q delivers m a finite number of times):

1. p sends m on FLL an infinite number of times:

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* $\cup \{(q, m)\}$;

2. If q stubborn delivers m a finite number of times, then FLL delivered m a finite number of times:

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

Step 2 - Stubborn Link $\rightarrow$ Perfect P2P Link

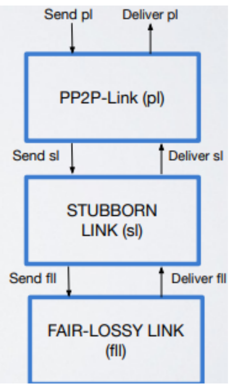
We can improve the Stubborn changing SL1 in:

- **PL1** (Reliable Delivery): if a correct process p sends m to q, then q eventually delivers m.
- **PL2** (No Duplication): a message is delivered at most once.
  and we also have:
- **FL3** (No Creation): if some process q delivers a message m with sender p, then m was sent by p to q.

upon event ⟨ *pl, Init* ⟩ **do**
  delivered := $\emptyset$;

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* $\cup \{m\}$;
    trigger ⟨ *pl, Deliver* | p, m ⟩;



Also this one isn't a good algorithm:

1. **FL3** (proof by contradiction): suppose process q executing our algorithm receives message m that was not sent by p.

    1. if q delivers a message then it delivers here:
       ```
       upon event ⟨ sl, Deliver | p, m ⟩ do
           if m ∉ delivered then
               delivered := delivered ∪ {m};
               trigger ⟨ pl, Deliver | p, m ⟩;
       ```
       and this implies that SL delivered a message that was not created. Violates the hypothesis that SL is a stubborn.
2. **PL2**: the pp2p-delivery of a message is guarded by an if $m \in delivered$:
   ```
   upon event ⟨ sl, Deliver | p, m ⟩ do
       if m ∉ delivered then
           delivered := delivered ∪ {m};
           trigger ⟨ pl, Deliver | p, m ⟩;
   ```
   suppose m is delivered twice, at time t and t' (with $t < t'$). We obtain that at time t the delivery handler is executed. Since the handler is atomic we have that $delivered := delivered \cup m$ is executed before t'. Therefore, at t', m is in delivered, this contradict the fact that trigger is executed at (or after) time t'.
3. **PL1**: suppose, p sends m and q does not deliver it. There could be two reasons for q to not deliver m:
    1. m is in delivered when the delivery handler is executed:
       ```
       upon event ⟨ sl, Deliver | p, m ⟩ do
           if m ∉ delivered then
               delivered := delivered ∪ {m};
               trigger ⟨ pl, Deliver | p, m ⟩;
       ```
       if m is in delivered then, q eventually will execute trigger . This contradicts the fact that q does not deliver m.
    2. The delivery handler is never triggered with $< p, m >$:
       ```
       upon event ⟨ sl, Deliver | p, m ⟩ do
           if m ∉ delivered then
               delivered := delivered ∪ {m};
               trigger ⟨ pl, Deliver | p, m ⟩;
       ```
       this means that SL is not stubborn. Violating our hypothesis.