MODELS, ABSTRACTIONS AND BASIC CONCEPTS

DISTRIBUTED SYSTEMS

Master of Science in Cyber Security

A.Y. 2023/2024

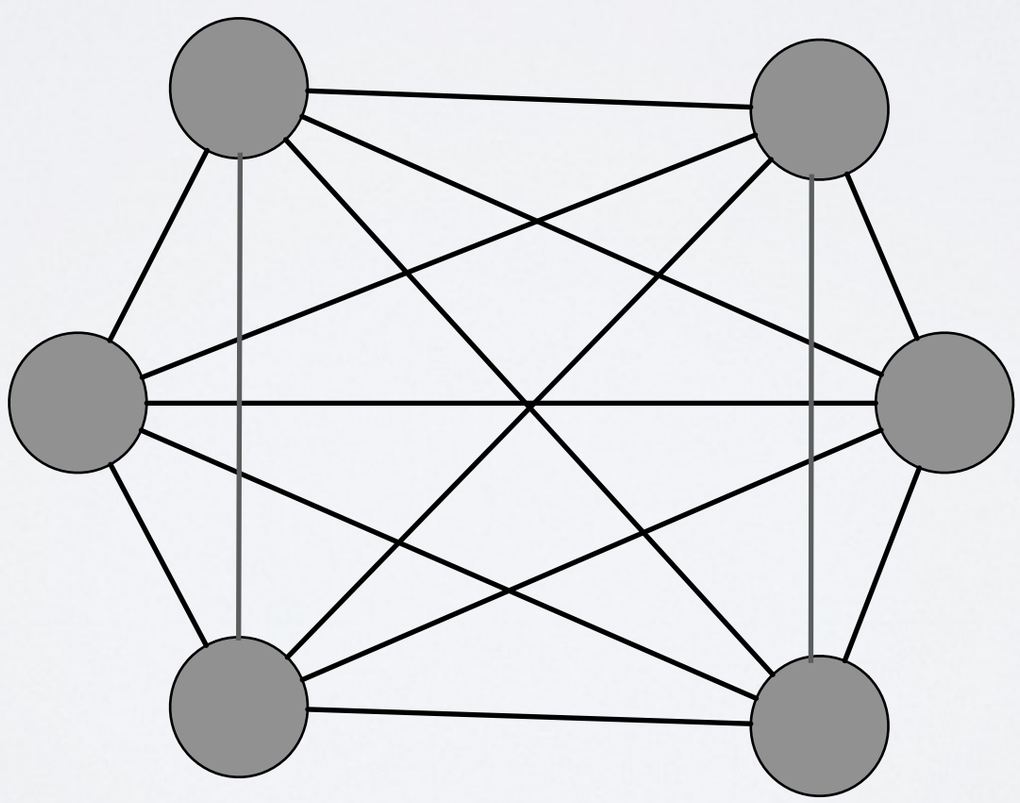
SAPIENZA
UNIVERSITÀ DI ROMA

CIS SAPIENZA
CYBER INTELLIGENCE AND INFORMATION SECURITY

MODEL:
SYSTEM AND PROCESSES

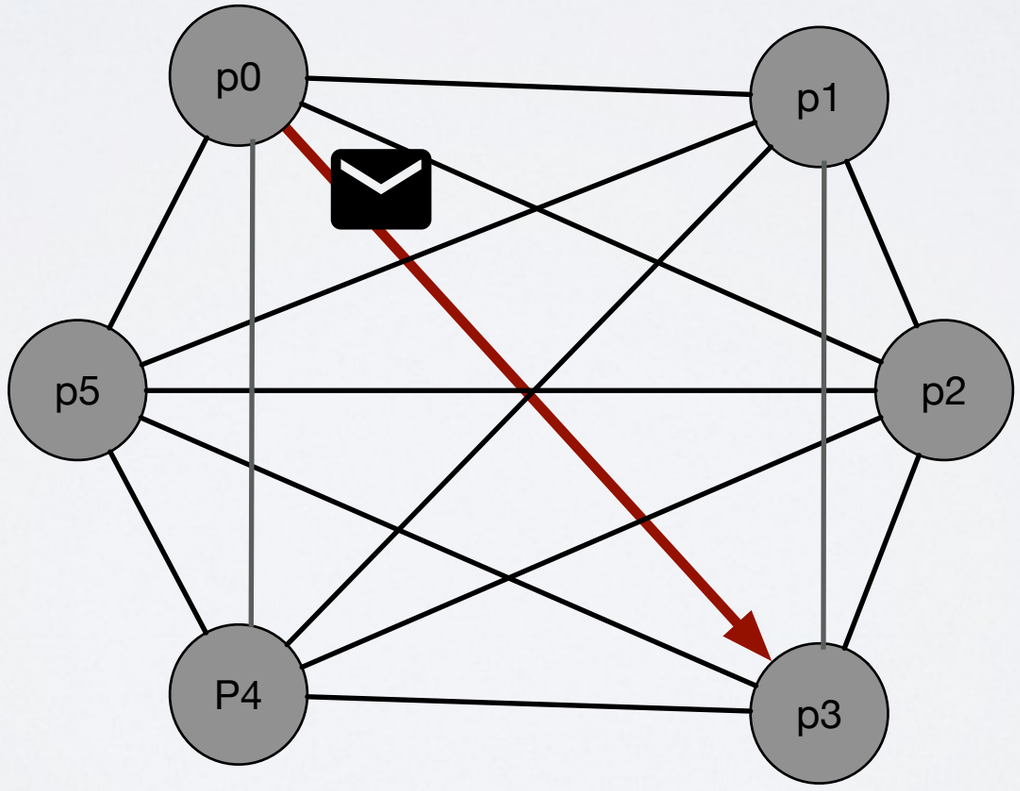
SYSTEM

- We have n processes in $\Pi : \{p_0, \ldots, p_{n-1}\}$ with distinct identities.
- Processes communicate using a communication graph $G : (\Pi, E)$ (usually, G is complete).
- The communication happens by exchanging messages on communication links (edges of G).



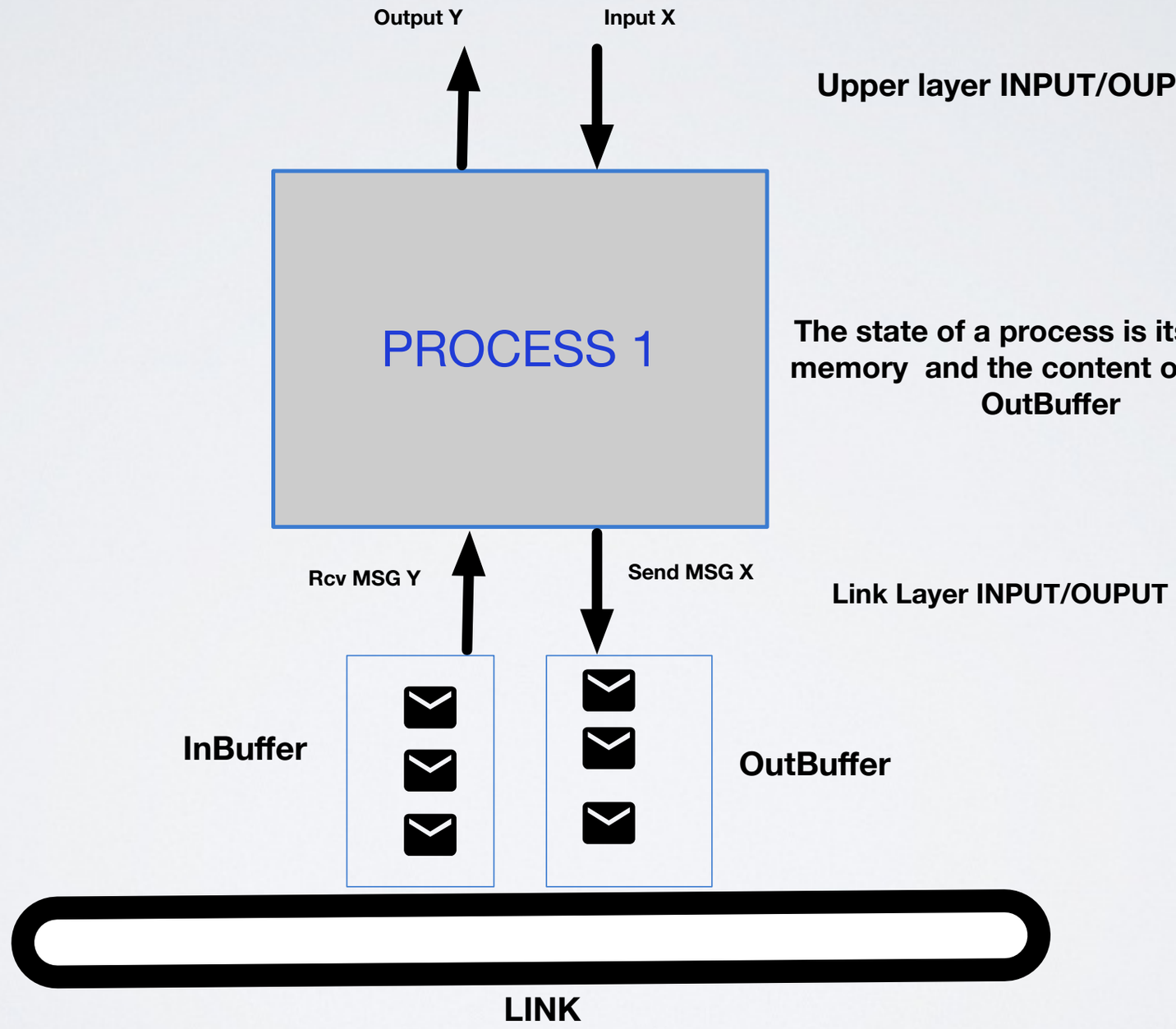
SYSTEM

- We have n processes in $\Pi : \{p_0, \ldots, p_{n-1}\}$ with distinct identities.
- Processes communicate using a communication graph $G : (\Pi, E)$ (usually, G is complete).
- The communication happens by exchanging messages on communication links (edges of G).



PROCESS

- A process is a (possibly infinite) State Machine (I/O Automaton).



PROCESS

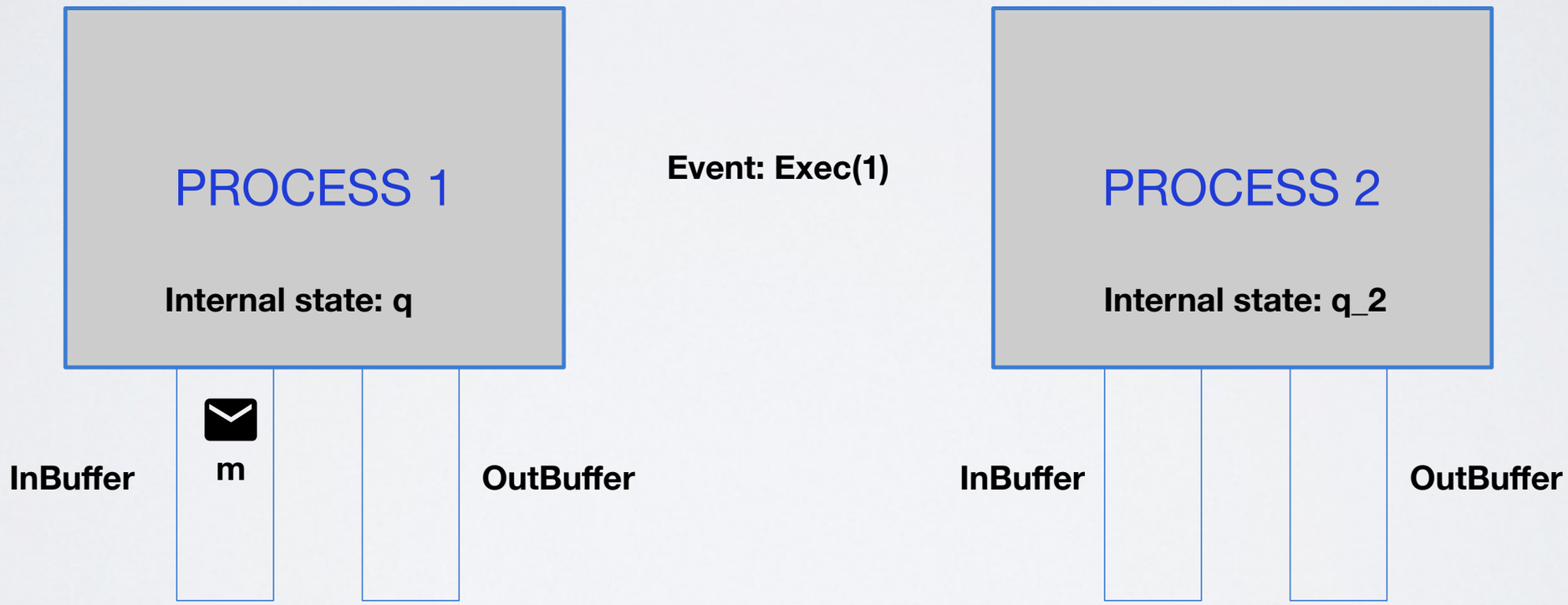
- A process is an infinite state I/O Automaton:
  - Internal states - set Q
  - Initial states - set Qi \subset Q
  - Messages - set of all possible messages M in the form <sender,receiver,payload>
  - InBuf_j - multiset of delivered messages
  - OutBuf_j - multiset of ``inflight'' messages (messages sent but not delivered).

$$P_j(q \in Q \cup Q_{in}, InBuf_j) = (q' \in Q, Send\_msg \subset M)$$
$$OutBuf_j = OutBuf_j \cup Send\_msg$$
$$InBuf_j = \emptyset$$

MODEL:
ASYNCHRONOUS EXECUTIONS

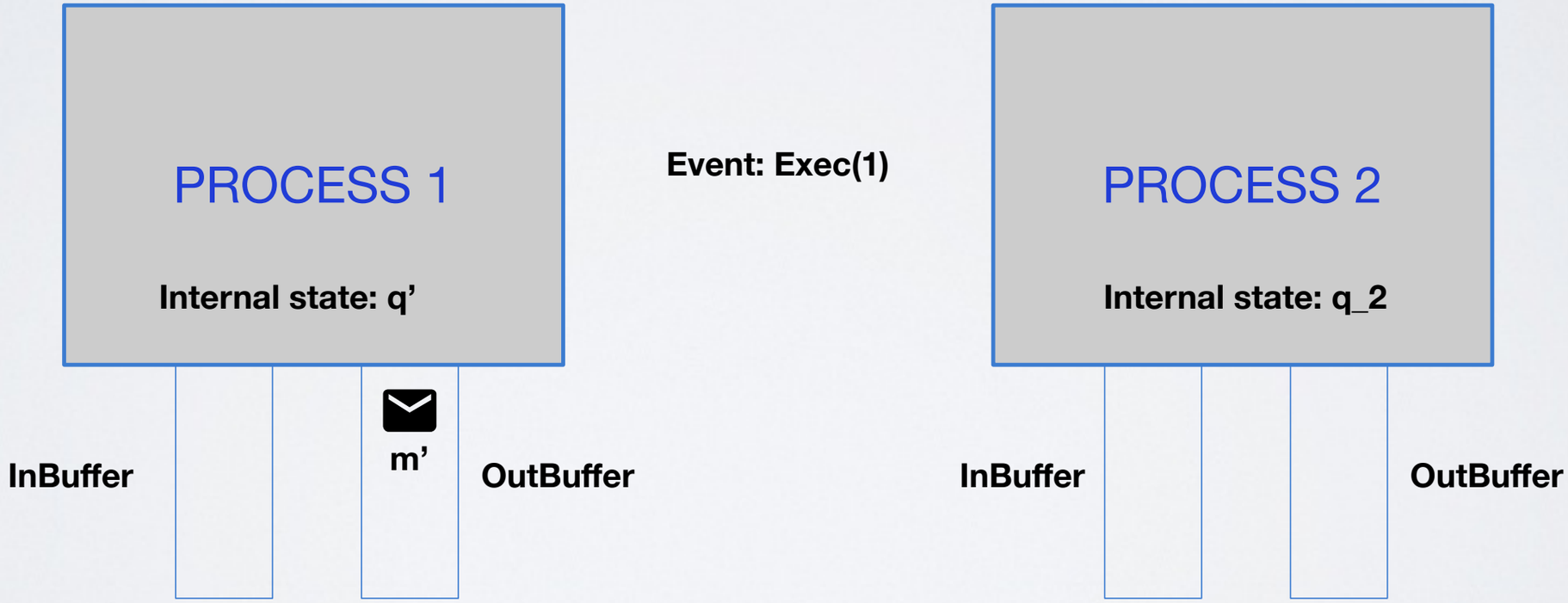
EXECUTION

- There is an adversary that schedule a set of events (scheduler):
  - Delivery of a msg - Del(m,i,j): move message m from OutBuff_i to InBuff_j
  - Execution of a local step - Exec(i): process i executes one step of its state machine.



EXECUTION

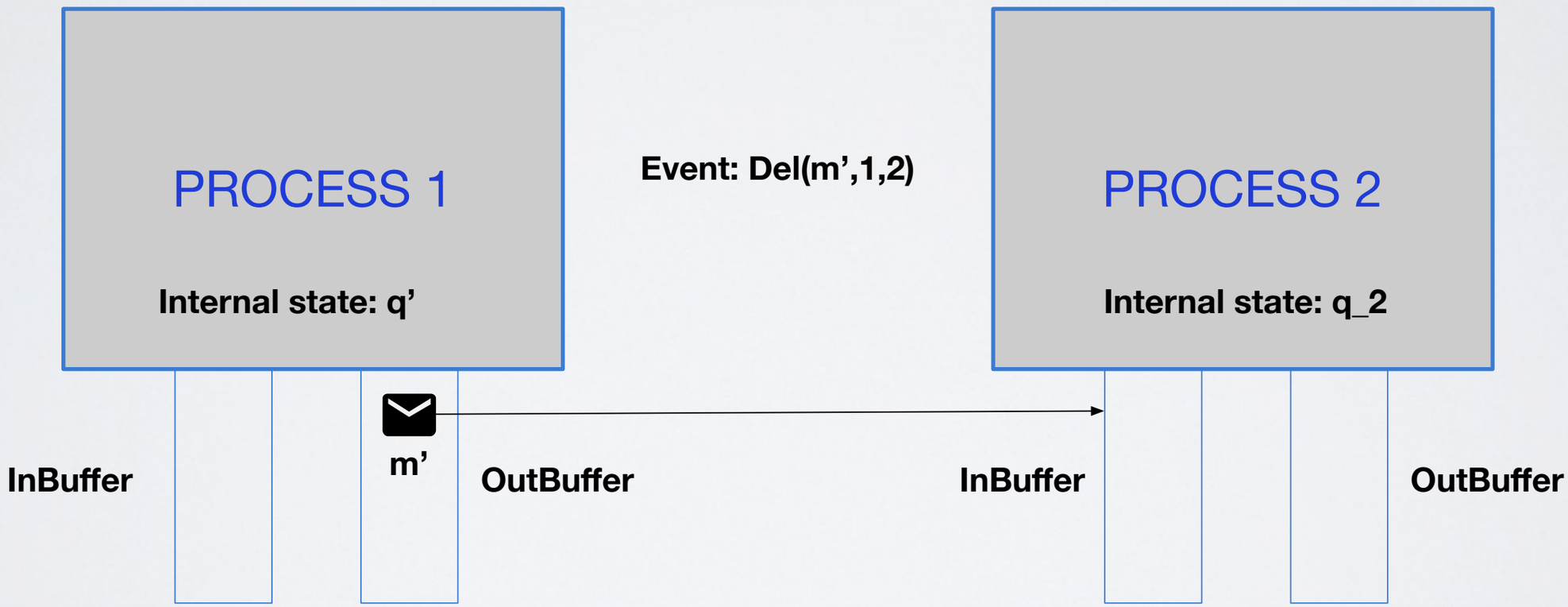
- There is an adversary that schedule a set of events (scheduler):
  - Delivery of a msg - Del(m,i,j): move message m from OutBuff_i to InBuff_j
  - Execution of a local step - Exec(i): process i executes one step of its state machine.



$$P_1(q, \{m\}) = (q', \{< 1, 2, m' >\})$$
$$OuBuf_1 = OutBuf_1 \cup \{< 1, 2, m' >\}$$

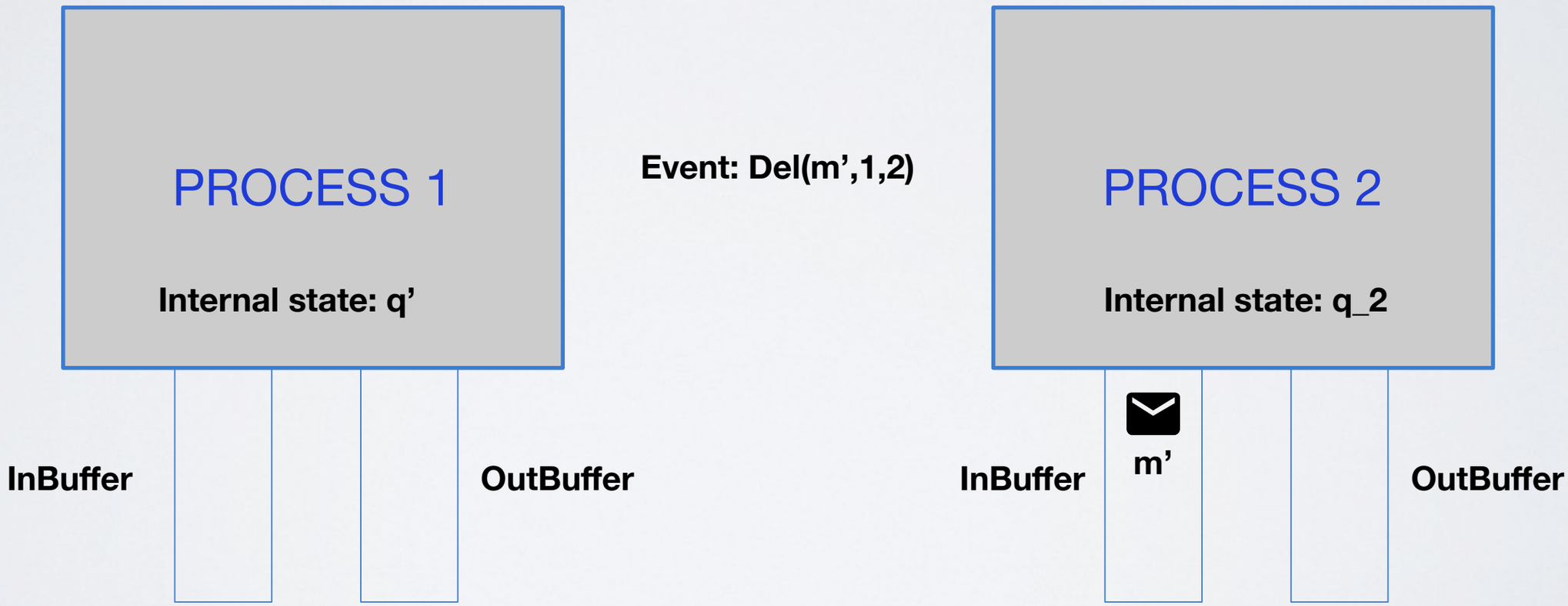
EXECUTION

- There is an adversary that schedule a set of events (scheduler):
  - Delivery of a msg - Del(m,i,j): move message m from OutBuff_i to InBuff_j
  - Execution of a local step - Exec(i): process i executes one step of its state machine.



EXECUTION

- There is an adversary that schedule a set of events (scheduler):
  - Delivery of a msg - Del(m,i,j): move message m from OutBuff_i to InBuff_j
  - Execution of a local step - Exec(i): process i executes one step of its state machine.

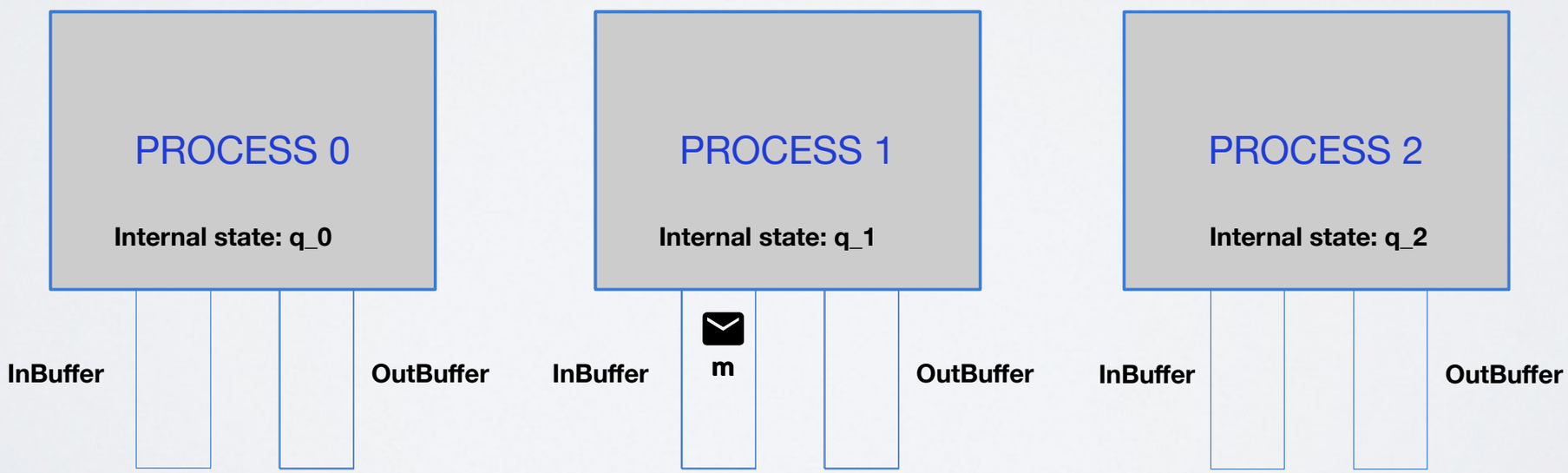


$$InBuf_2 = InBuf_2 \cup \{< 1, 2, m' >\}$$

EXECUTION

- A configuration C_t is a vector of n components. Component j indicates the state of process j (that is C_t[j] (q_i,InBuff_j,OutBuff_j)).

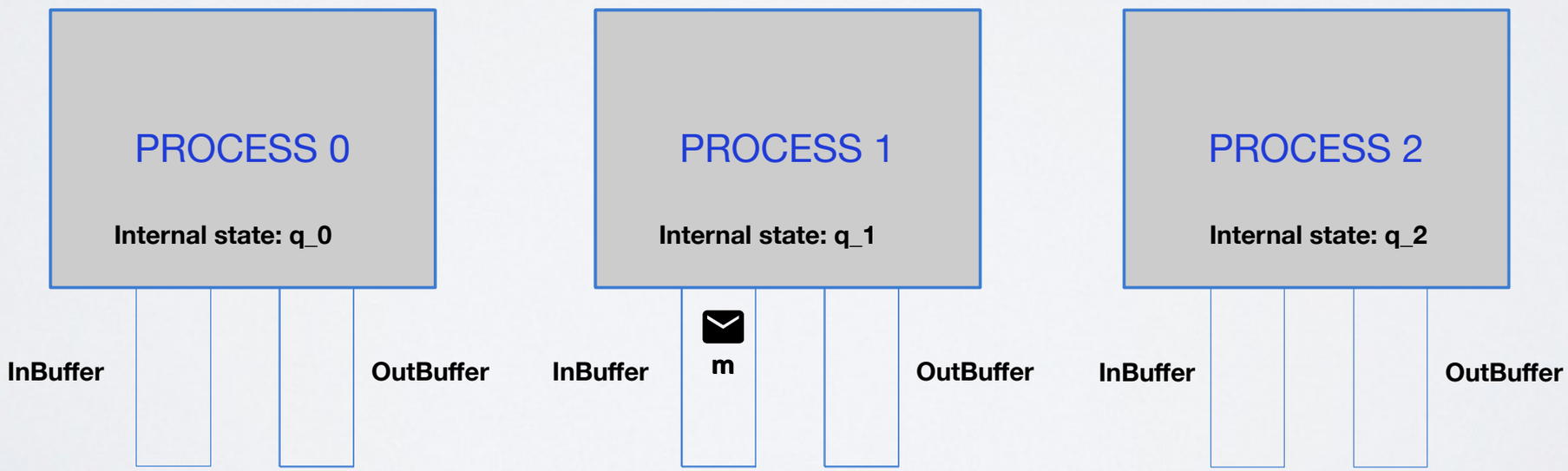
$$C_0 =< (q_0, \{\}, \{\}), (q_1, \{m\}, \{\}), (q_2, \{\}, \{\}) >$$



EXECUTION

- An event e is enabled in a configuration C if it can happen:
  - Del(m',0,2) is not enabled in C_0 because OutBuf_0 does not contain a message m'.
  - Exec(1) is enabled in C_0, as Exec(0) and Exec(2).

$$C_0 =< (q_0, \{\}, \{\}), (q_1, \{m\}, \{\}), (q_2, \{\}, \{\}) >$$

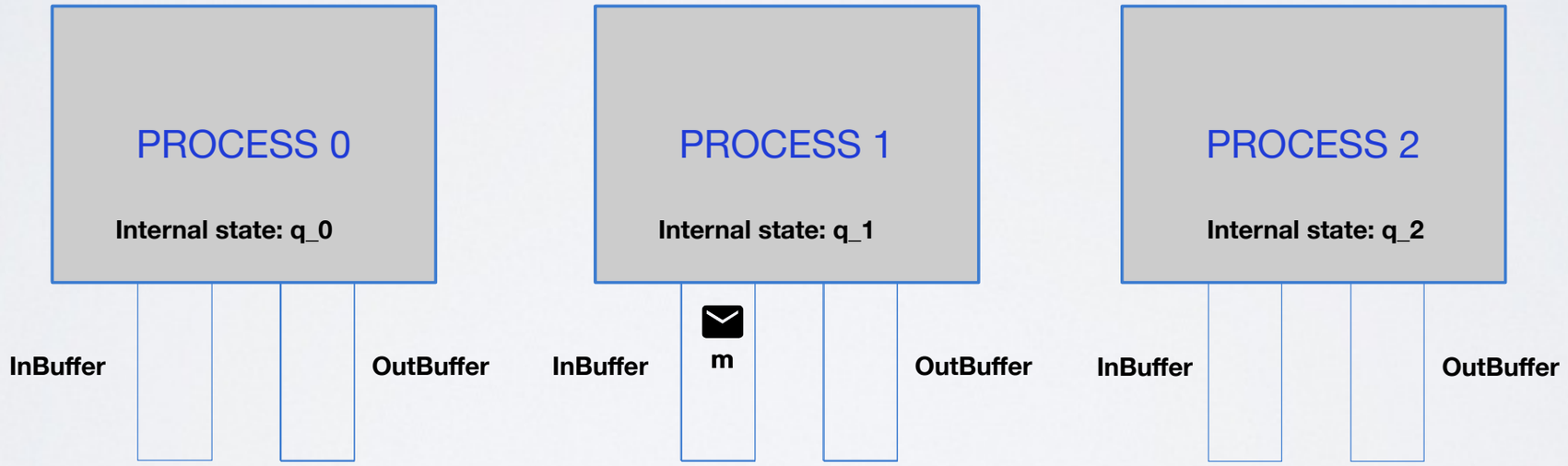


EXECUTION

- An execution is infinite sequence that alternates configurations and events: (C_0,e_0,C_1,e_1,C_2,e_2,….) such that each event e_{t} is enabled in configuration C_t and C_t is obtained by applying e_{t-1} to C_{t-1}.

$$\mathcal{E} : (C_0, e_0 = Exec(1), C_1, e_1 = Del(1, 2, m'), C_2, e_2 = Exec(2), C_3, e_3, C_4, \ldots)$$

C_0

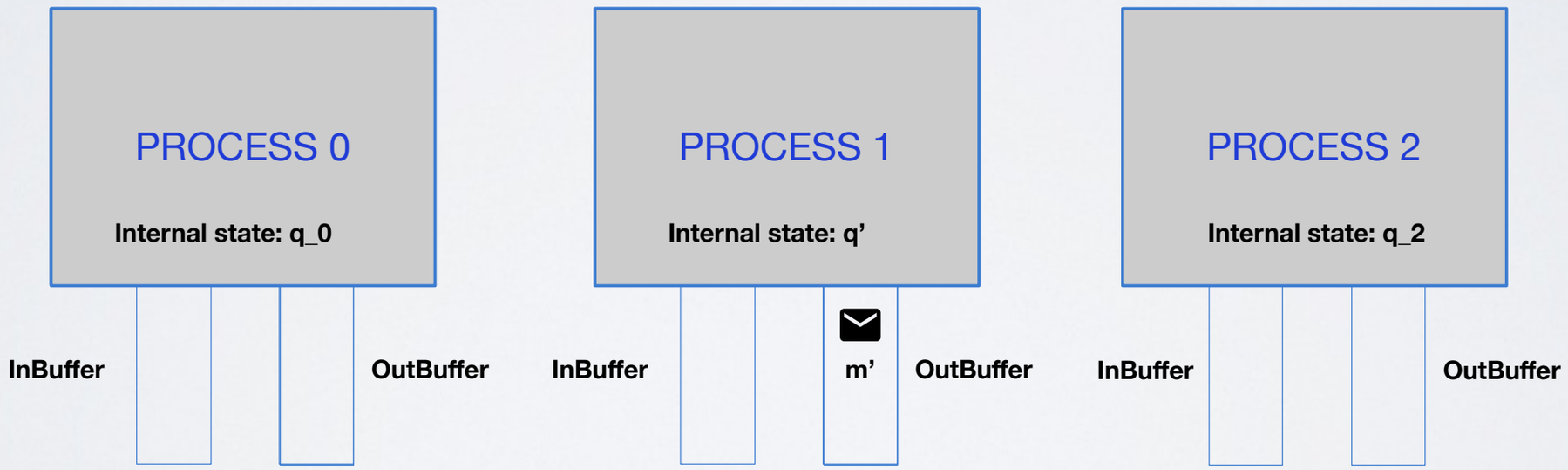


EXECUTION

- An execution is infinite sequence that alternates configurations and events: (C_0,e_0,C_1,e_1,C_2,e_2,….) such that each event e_{t} is enabled in configuration C_t and C_t is obtained by applying e_{t-1} to C_{t-1}.

$$\mathcal{E} : (C_0, e_0 = Exec(1), C_1, e_1 = Del(1, 2, m'), C_2, e_2 = Exec(2), C_3, e_3, C_4, \ldots)$$

C_1

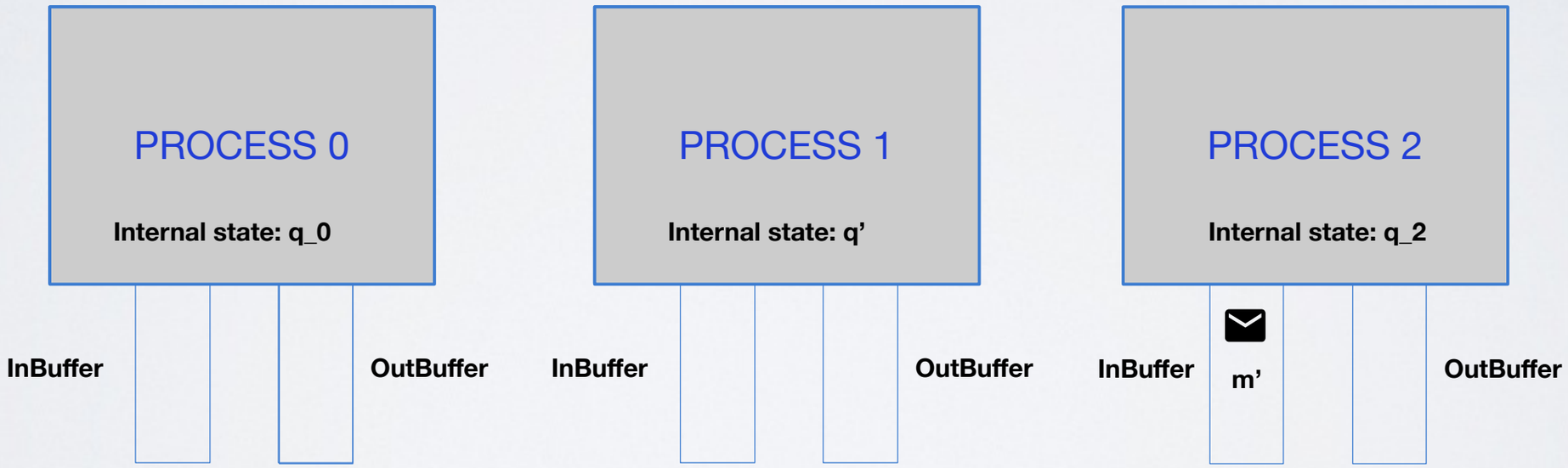


EXECUTION

- An execution is infinite sequence that alternates configurations and events: (C_0,e_0,C_1,e_1,C_2,e_2,….) such that each event e_{t} is enabled in configuration C_t and C_t is obtained by applying e_{t-1} to C_{t-1}.

$$\mathcal{E} : (C_0, e_0 = Exec(1), C_1, e_1 = Del(1, 2, m'), C_2, e_2 = Exec(2), C_3, e_3, C_4, \ldots)$$

C_2

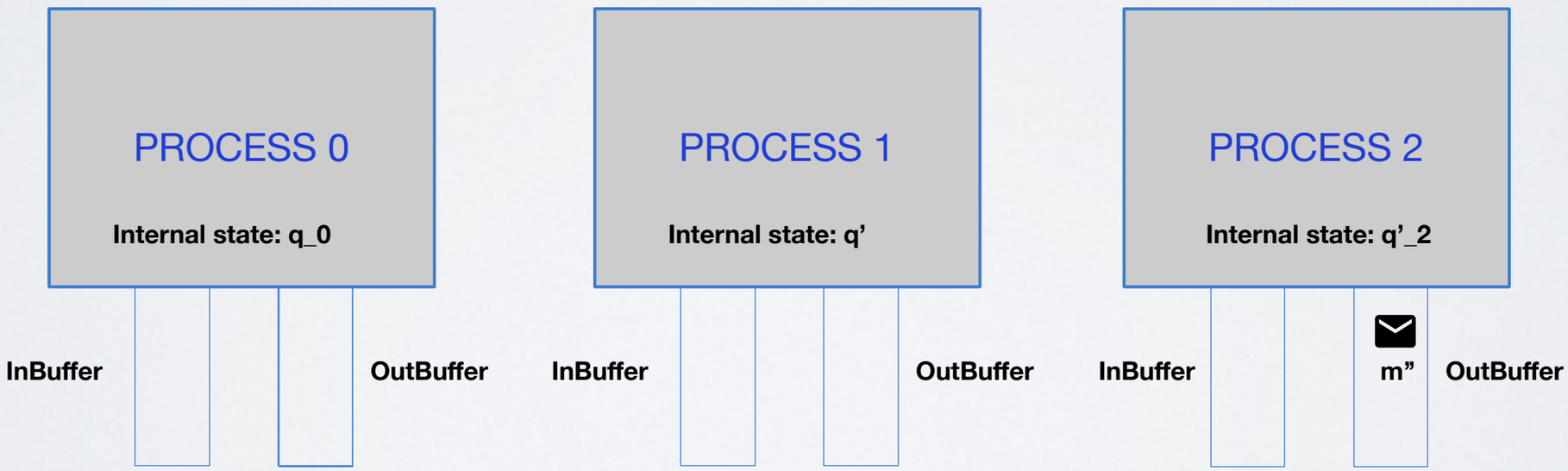


EXECUTION

- An execution is infinite sequence that alternates configurations and events: (C_0,e_0,C_1,e_1,C_2,e_2,….) such that each event e_{t} is enabled in configuration C_t and C_t is obtained by applying e_{t-1} to C_{t-1}.

$$\mathcal{E} : (C_0, e_0 = Exec(1), C_1, e_1 = Del(1, 2, m'), C_2, e_2 = Exec(2), C_3, e_3, C_4, \ldots)$$

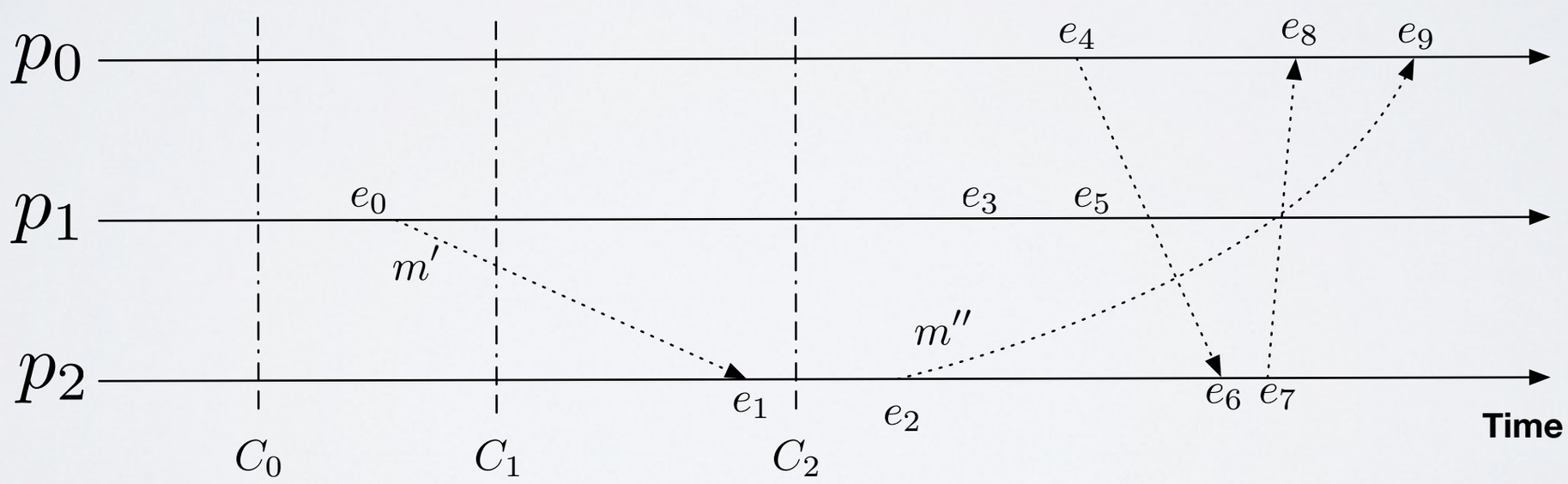
C_3



EXECUTION

- An execution is infinite sequence that alternates configurations and events: (C_0,e_0,C_1,e_1,C_2,e_2,….) such that each event e_{t} is enabled in configuration C_t and C_t is obtained by applying e_{t-1} to C_{t-1}.

$$\mathcal{E} : (C_0, e_0 = Exec(1), C_1, e_1 = Del(1, 2, m'), C_2, e_2 = Exec(2), C_3, e_3, C_4, \ldots)$$



FAIR EXECUTION

- An execution E is fair if each process p_i executes an infinite number of local computations (Exec(i) events are not finite) and each message m is eventually delivered (is not possible to stall forever a message: there must exists a Del(m,x,y)).

- We will always consider fair executions.

- Unfair executions brake any possible non-trivial algorithm (**why?**).

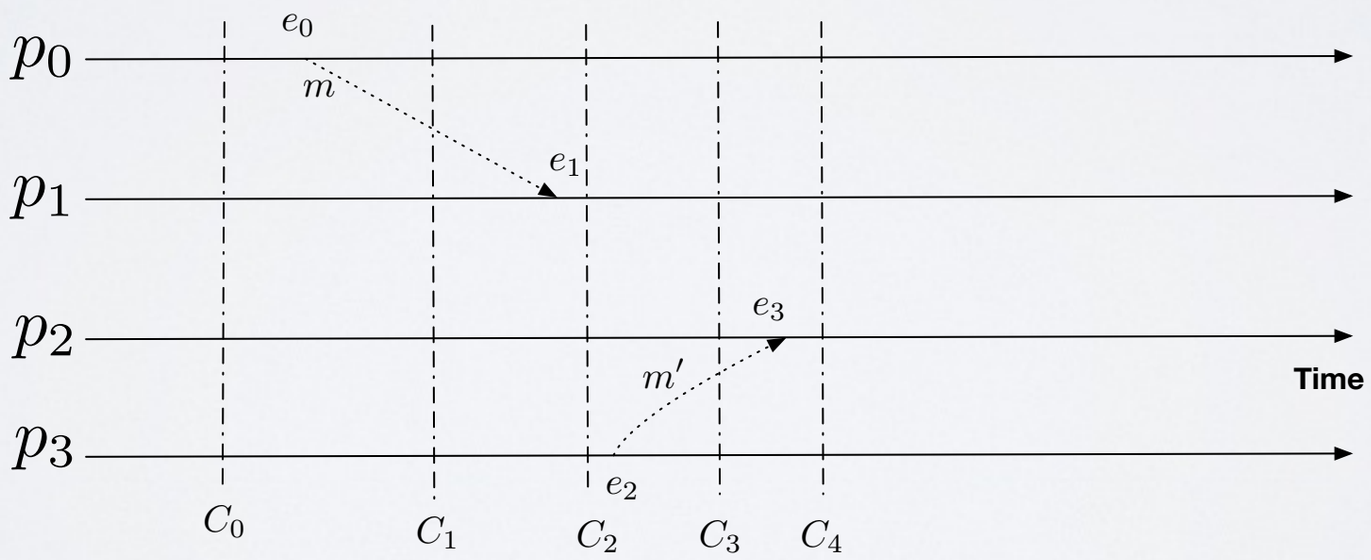
LOCAL EXECUTION (OR LOCAL VIEW)

- Given an execution E and a process p_j, we define the local execution (E|p_j) of p_j as the subset of events in E that ``impact" p_j.

$$\mathcal{E} = (C_0, e_0 = Exec(0), C_1, e_1 = Del(0,1,m), C_2, e_2 = Exec(3), C_3, e_3 = Del(3,2,m'), ...)$$

$$\mathcal{E}|p_1 = (Del(0,1,m), ...)$$

$$\mathcal{E}|p_2 = (Del(3,2,m'), ...)$$

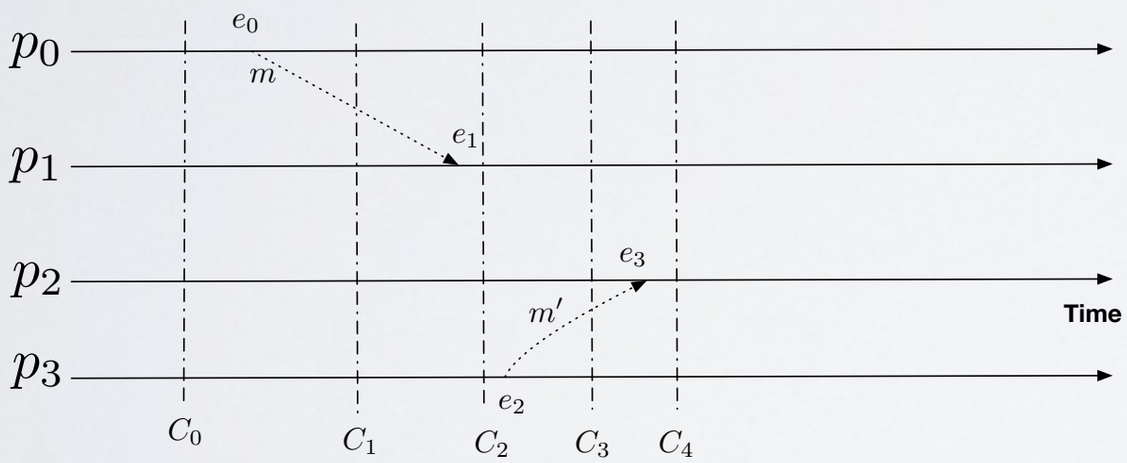


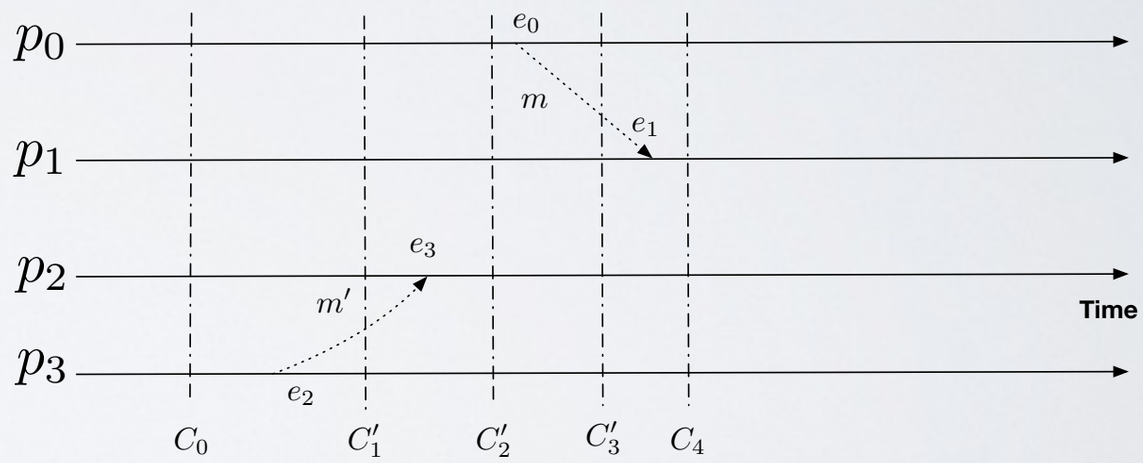
LOCAL EXECUTION

- Different executions could give the same local exec.:

$$\mathcal{E} = (C_0, e_0 = Exec(0), C_1, e_1 = Del(0,1,m), C_2, e_2 = Exec(3), C_3, e_3 = Del(3,2,m'), C_4, ...)$$
$$\mathcal{E}' = (C_0, e_2 = Exec(3), C_1', e_3 = Del(3,2,m'), C_2', e_0 = Exec(0), C_3', e_1 = Del(0,1,m), C_4, ...)$$

$$\mathcal{E}|p_1 = \mathcal{E}'|p_1 \text{ and } \mathcal{E}|p_2 = \mathcal{E}'|p_2$$



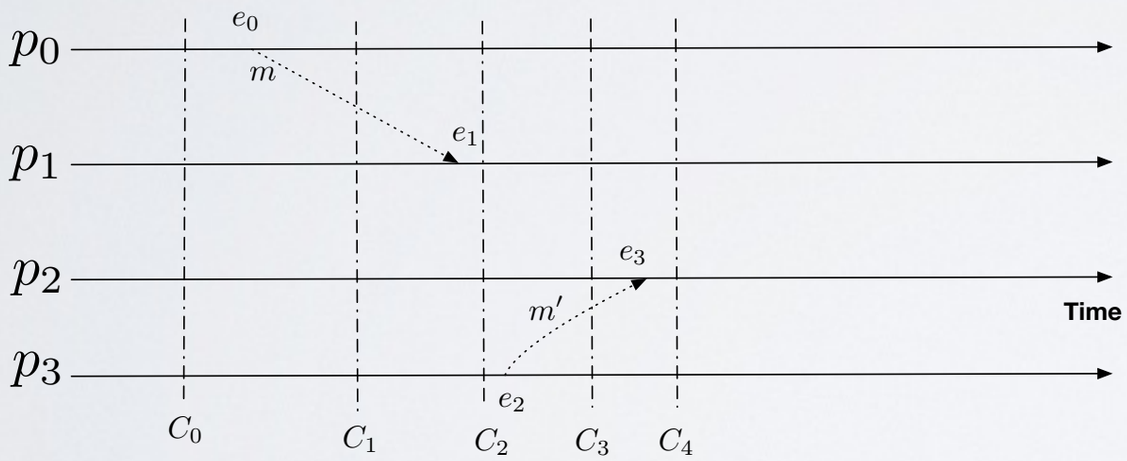


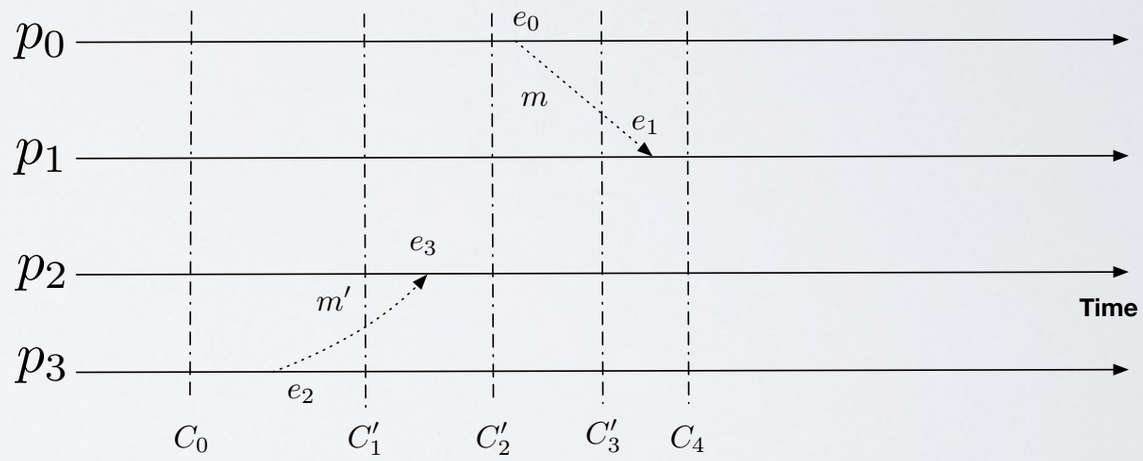
LOCAL EXECUTION

- Different executions could give the same local exec.:

$$\mathcal{E} = (C_0, e_0 = Exec(0), C_1, e_1 = Del(0,1,m), C_2, e_2 = Exec(3), C_3, e_3 = Del(3,2,m'), C_4, ...)$$
$$\mathcal{E}' = (C_0, e_2 = Exec(3), C_1', e_3 = Del(3,2,m'), C_2', e_0 = Exec(0), C_3', e_1 = Del(0,1,m), C_4, ...)$$

$$\mathcal{E}|p_1 = \mathcal{E}'|p_1 \text{ and } \mathcal{E}|p_2 = \mathcal{E}'|p_2$$





- We say that p_1 (and p_2) cannot distinguish E from E'.

INDISTINGUISHABILITY

$$\mathcal{E} = (C_0, e_0 = Exec(0), C_1, e_1 = Del(0,1,m), C_2, e_2 = Exec(3), C_3, e_3 = Del(3,2,m'), C_4, ...)$$
$$\mathcal{E}' = (C_0, e_2 = Exec(3), C_1', e_3 = Del(3,2,m'), C_2', e_0 = Exec(0), C_3', e_1 = Del(0,1,m), C_4, ...)$$

- It is not hard to see that: $\forall p_j \in \Pi,\ \mathcal{E}|p_j = \mathcal{E}'|p_j$
- In this case we say that E and E' are indistinguishable.

INDISTINGUISHABILITY

$$\mathcal{E} = (C_0, e_0 = Exec(0), C_1, e_1 = Del(0,1,m), C_2, e_2 = Exec(3), C_3, e_3 = Del(3,2,m'), C_4, ...)$$
$$\mathcal{E}' = (C_0, e_2 = Exec(3), C_1', e_3 = Del(3,2,m'), C_2', e_0 = Exec(0), C_3', e_1 = Del(0,1,m), C_4, ...)$$

- It is not hard to see that: $\forall p_j \in \Pi,\ \mathcal{E}|p_j = \mathcal{E}'|p_j$
- In this case we say that E and E' are indistinguishable.

- **Theorem**: In the asynch. model there is no distributed algorithm capable of reconstructing the system execution.

SYNCHRONOUS VS ASYNCHRONOUS

SYNCHRONOUS

- In this course we will see three definitions of synchrony:

- Asynchronous system
- Eventually-synchronous (in module M2)
- Synchronous system

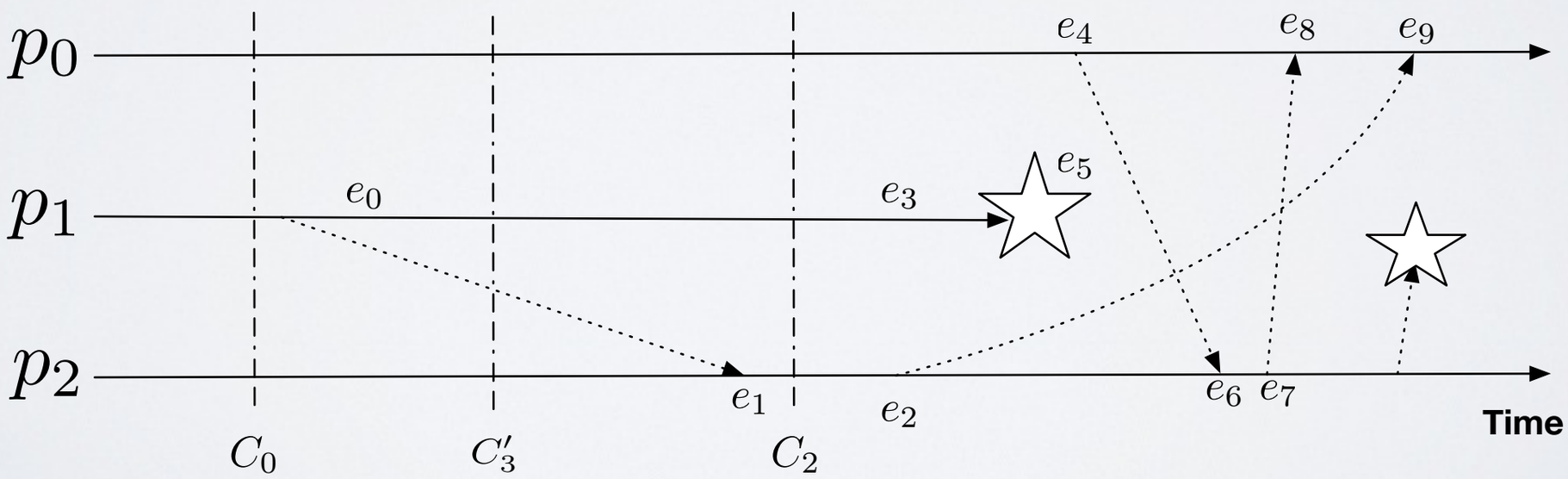
- A System is synchronous if there is fixed bound on the delay of messages.

MODEL: FAILURES

MODELS FOR PROCESS FAILURES

- Two main models:
  - Crash-Stop Failure
  - Byzantine Failure

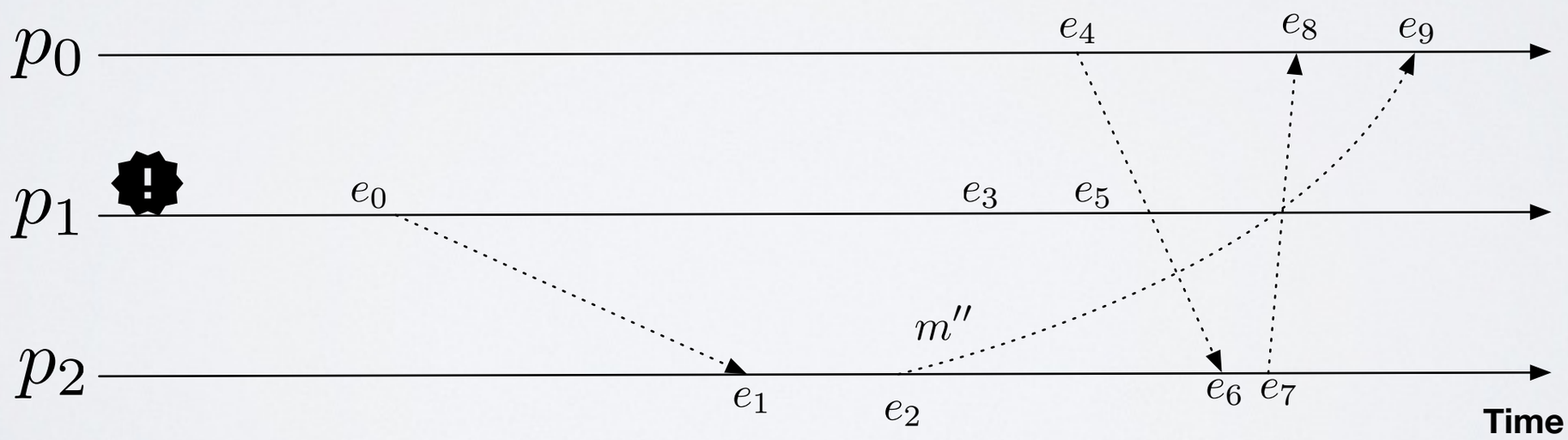
- Crash(p_j): after this event process p_j does not execute any local computation step (no Exec(j)).



MODELS FOR PROCESS FAILURES

- Two main models:
  - Crash-Stop Failure
  - Byzantine Failure

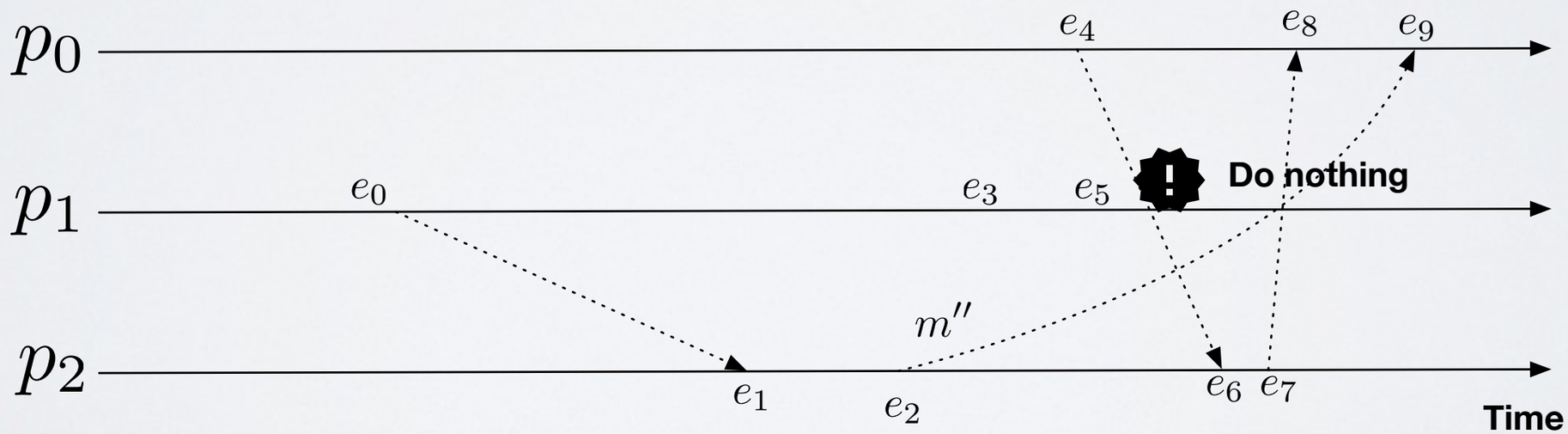
- Byz(p_j): after this event process p_j behaves in an arbitrary way (when we have Exec(j) we do not follow anymore the automaton of p_j).



MODELS FOR PROCESS FAILURES

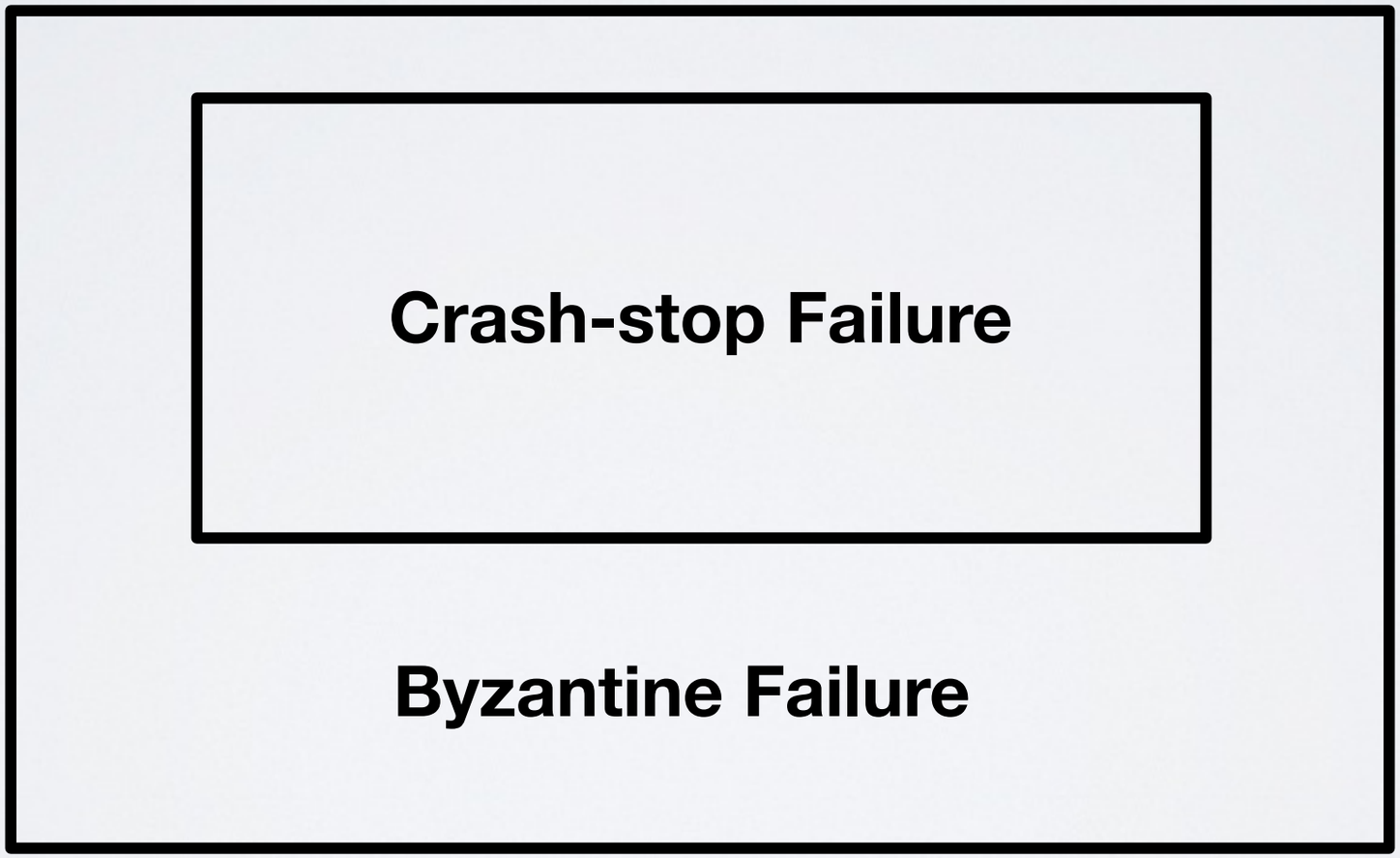
- Two main models:
  - Crash-Stop Failure
  - Byzantine Failure

- Byz(p_j): after this event process p_j behaves in an arbitrary way (when we have Exec(j) we do not follow anymore the automaton of p_j).



MODELS FOR PROCESS FAILURES

- Two main models:
  - Crash-Stop Failure
  - Byzantine Failure



MODELS FOR PROCESS FAILURES

- Two main models:
  - Crash-Stop Failure
  - Byzantine Failure
- We say that a process is **correct** if it **does not experience a failure** (but we do not know who is correct).
- We indicate with **f** the **maximum number** of processes that can experience a failure event in an execution.
  - Usually we bound f. Example: f cannot be more than n-1.
  - We never do assumptions on when a failure event happens.

HOMEWORKS

- Model the loss of a message as an event.

ABSTRACTIONS, PSEUDO-CODE CONVENTIONS AND OUR FIRST ALGORITHM

ABSTRACTION AND PROPERTIES

THE ROAD TO BUILD A DISTRIBUTED ABSTRACTION

Step 1: define a system model

Step 2: formalize a problem/(object) with an abstraction

understand how to design an abstract formal object that captures the problem that you want to solve:

- Specify unambiguously the properties of your abstraction

Step 3: implements the abstraction with a distributed protocol

Step 4: prove that the protocol implements the abstraction

ABSTRACTING A REAL COMMUNICATION LINK

- Informal description: We have **a link** that loses messages with a certain probability pr, the channel can duplicate a message a finite number of times, and it does not create messages from thin air.

- A link exposes two events:
  - Requests: <Send | q,m> send a message m to process q.
  - Indication: <Deliver, p,m> delivers message m from process p.

ABSTRACTING A REAL COMMUNICATION LINK

- Informal description: We have a link that **loses messages with a certain probability pr**, the channel can duplicate a message a finite number of times, and it does not create messages from thin air.

- A link exposes two events:
  - Requests: <Send | q,m> send a message m to process q.
  - Indication: <Deliver, p,m> delivers message m from process p.
- Properties:
  - FL1: (Fair-loss) If a correct process p sends infinitely often m to a process q, then q delivers m an infinite number of times.

ABSTRACTING A REAL COMMUNICATION LINK

- Informal description: We have a link that loses messages with a certain probability pr, **the channel can duplicate a message a finite number of times**, and it does not create messages from thin air.

- A link exposes two events:
  - Requests: <Send | q,m> send a message m to process q.
  - Indication: <Deliver, p,m> delivers message m from process p.
- Properties:
  - FL1: (Fair-loss) If a correct process p sends infinitely often m to a process q, then q delivers m an infinite number of times.
  - FL2: (Finite duplication) If a correct process p sends m a finite number of times to q, then q cannot deliver m an infinite number of times.

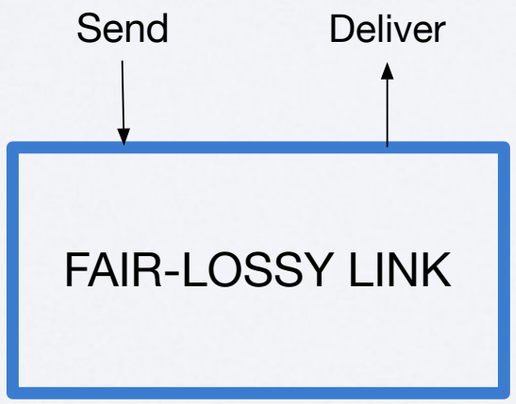
ABSTRACTING A REAL COMMUNICATION LINK

- Informal description: We have a link that loses messages with a certain probability pr, the channel can duplicate a message a finite number of times, and **it does not create messages from thin air.**

- A link exposes two events:
  - Requests: <Send | q,m> send a message m to process q.
  - Indication: <Deliver, p,m> delivers message m from process p.
- Properties:
  - FL1: (Fair-loss) If a correct process p sends infinitely often m to a process q, then q delivers m an infinite number of times.
  - FL2: (Finite duplication) If a correct process p sends m a finite number of times to q, then q cannot deliver m an infinite number of times.
  - FL3: (No creation) If some process q delivers a message m with sender p, then m was sent by p to q.

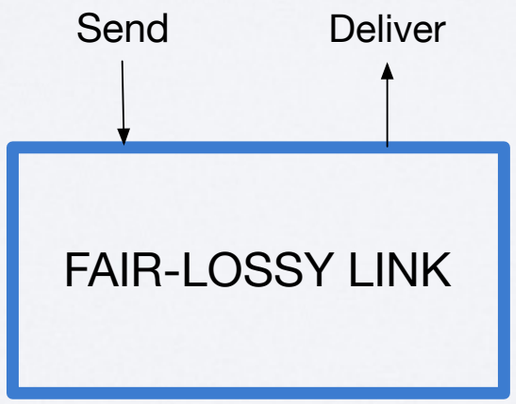
FAIR-LOSSY LINK

- A link exposes two events:
  - Requests: <Send | q,m> send a message m to process q.
  - Indication: <Deliver, p,m> delivers message m from process p.
- Properties:
  - FL1: (Fair-loss) If a correct process p sends infinitely often m to a process q, then q delivers m an infinite number of times.
  - FL2: (Finite duplication) If a correct process p sends m a finite number of times to q, then q cannot deliver m an infinite number of times.
  - FL3: (No creation) If some process q delivers a message m with sender p, then m was sent by p to q.



FAIR-LOSSY LINK

- A link exposes two events:
  - Requests: <Send | q,m> send a message m to process q.
  - Indication: <Deliver, p,m> delivers message m from process p.
- Properties:
  - FL1: (Fair-loss) If a correct process p sends infinitely often m to a process q, then q delivers m an infinite number of times.
  - FL2: (Finite duplication) If a correct process p sends m a finite number of times to q, then q cannot deliver m an infinite number of times.
  - FL3: (No creation) If some process q delivers a message m with sender p, then m was sent by p to q.



LET US FOCUS ON PROPERTIES

- Properties:
  - FL1: (Fair-loss) If a correct process p sends infinitely often m to a process q, then q delivers m an infinite number of times.
  - FL2: (Finite duplication) If a correct process p sends m a finite number of times to q, then q cannot deliver m an infinite number of times.
  - FL3: (No creation) If some process q delivers a message m with sender p, then m was sent by p to q.

- All properties fall in two classes: ***Safety*** or ***Liveness***.
  - A **Safety property** is property that **if violated at a time t, it can never be satisfied after t.**
    - Formally, if a safety property is violated in execution E, there is a prefix E' of E such that any extension of E' also violates the property.

LET US FOCUS ON PROPERTIES

- Properties:
  - FL1: (Fair-loss) If a correct process p sends infinitely often m to a process q, then q delivers m an infinite number of times.
  - FL2: (Finite duplication) If a correct process p sends m a finite number of times to q, then q cannot deliver m an infinite number of times.
  - FL3: (No creation) If some process q delivers a message m with sender p, then m was sent by p to q.

- All properties fall in two classes: ***Safety*** or ***Liveness***.
  - A Safety property is property that if violated at a time t, it can never be satisfied after t.
    - Formally, if a safety property is violated in execution E, there is a prefix E' of E such that any extension of E' also violates the property.
    - Informally, **a safety property states that something bad cannot happen**.
    - Among FL1, FL2 and FL3 there is at least one safety property. Which one?

LET US FOCUS ON PROPERTIES

- Properties:
  - FL1: (Fair-loss) If a correct process p sends infinitely often m to a process q, then q delivers m an infinite number of times.
  - FL2: (Finite duplication) If a correct process p sends m a finite number of times to q, then q cannot deliver m an infinite number of times.
  - **FL3: (No creation) If some process q delivers a message m with sender p, then m was sent by p to q.**

- ***Safety:***
  - Once a process q delivers a message m that was not sent by anyone FL3 is violated forever.
    - E=(e_1,e_2,e_3,e_4,e_5,……)
    - E'=(e_1,e_2,e_3,e_4)

LET US FOCUS ON PROPERTIES

- Properties:
  - FL1: (Fair-loss) If a correct process p sends infinitely often m to a process q, then q delivers m an infinite number of times.
  - FL2: (Finite duplication) If a correct process p sends m a finite number of times to q, then q cannot deliver m an infinite number of times.
  - FL3: (No creation) If some process q delivers a message m with sender p, then m was sent by p to q.

- All properties fall in two classes: ***Safety*** or ***Liveness***.
  - A Liveness property is property that cannot be violated in finite executions:
    - Formally, given any finite execution E that does not satisfy a liveness property there is an extension of E that satisfy it.
    - Informally, a liveness property specify that something good will happen.
    - Among FL1, FL2 and FL3 there is at least one liveness property. Which one?

LET US FOCUS ON PROPERTIES

- Properties:
  - **FL1: (Fair-loss) If a correct process p sends infinitely often m to a process q, then q delivers m an infinite number of times.**
  - FL2: (Finite duplication) If a correct process p sends m a finite number of times to q, then q cannot deliver m an infinite number of times.
  - FL3: (No creation) If some process q delivers a message m with sender p, then m was sent by p to q.
- ***Liveness***.
  - If p sent a message m, and q has not delivered it at time t, there is always hope that it can deliver it at time t+\Delta.
    - E=(e_1,e_2,e_3,e_4,e_5)
    - E'=(e_1,e_2,e_3,e_4,e_5, del(p,q,m))

LET US FOCUS ON PROPERTIES

- Properties:
  - FL1: (Fair-loss) If a correct process p sends infinitely often m to a process q, then q delivers m an infinite number of times.
  - **FL2: (Finite duplication) If a correct process p sends m a finite number of times to q, then q cannot deliver m an infinite number of times.**
  - FL3: (No creation) If some process q delivers a message m with sender p, then m was sent by p to q.
- ***?***

LET US FOCUS ON PROPERTIES

- Properties:
  - FL1: (Fair-loss) If a correct process p sends infinitely often m to a process q, then q delivers m an infinite number of times.
  - **FL2: (Finite duplication) If a correct process p sends m a finite number of times to q, then q cannot deliver m an infinite number of times.**
  - FL3: (No creation) If some process q delivers a message m with sender p, then m was sent by p to q.
- ***Liveness.***
  - It cannot be violated in finite execution.
    - E=(del(p,q,m),del(p,q,m),del(p,q,m),…,del(p,q,m))
    - E'=(del(p,q,m),del(p,q,m),del(p,q,m),…,del(p,q,m),e_1,e_2,e_3,….)
      - Such that no e_j is del(p,q,m).

LET US FOCUS ON PROPERTIES

- Property:
  - **FL2: (At-most7 duplication)** If a correct process p sends m a finite number of times to q, then q cannot deliver m **more than 7 times**.
- ***Liveness?***

LET US FOCUS ON PROPERTIES

- Property:
  - **FL2: (At-most7 duplication)** If a correct process p sends m a finite number of times to q, then q cannot deliver m **more than 7 times**.
- ***Liveness?***
  - E=(del(p,q,m),del(p,q,m),del(p,q,m),del(p,q,m),del(p,q,m),del(p,q,m),del(p,q,m),del(p,q,m))
    - E cannot be extended, it always violates At-most7 duplication.
  - At-most7 duplication is a **safety** property.

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.

BADLY WRITTEN PROPERTIES

- If process p sends a message m to q, then q will eventually deliver it and this deliver is unique.
  - It is mixing two aspects:
    - A liveness: q will eventually deliver it
    - A safety: the deliver is unique

BADLY WRITTEN PROPERTIES

- If process p sends a message m to q, then q will eventually deliver it and this deliver is unique.
  - It is mixing two aspects:
    - A liveness: q will eventually deliver it
    - A safety: the deliver is unique
- It should be decomposed in two properties:
  - 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.

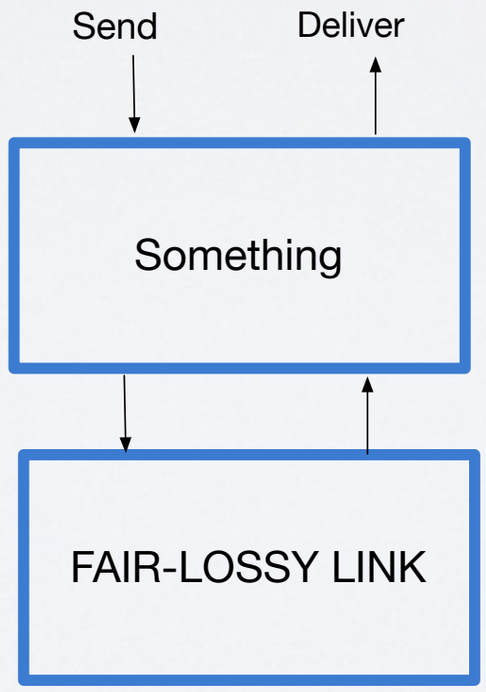
HOMEWORKS

- Safety or liveness?
  - Fairness of an execution.
  - (At most-f failures) An execution E can contain at most f failures.
  - (Eventual recovery) If a process p experience a crash(p), then it eventually experiences also a recovery(p) (this event starts again process p).
  - (Time-bounded recovery) If a process p experiences a recovery(p), then this recovery happens at most k events after its failure.
- Write three new safety properties and three new liveness properties.
- Does Time-bounded recovery implies Eventual recovery?
- How would you improve the writing of this property:
  - If a process p outputs a set of integer O_p, then any other process q outputs also outputs a set O_q. For any two processes z and t we have either O_z is a subset of O_t or that O_t is a subset of O_z.

ALGORITHMS: COMMUNICATION LINKS

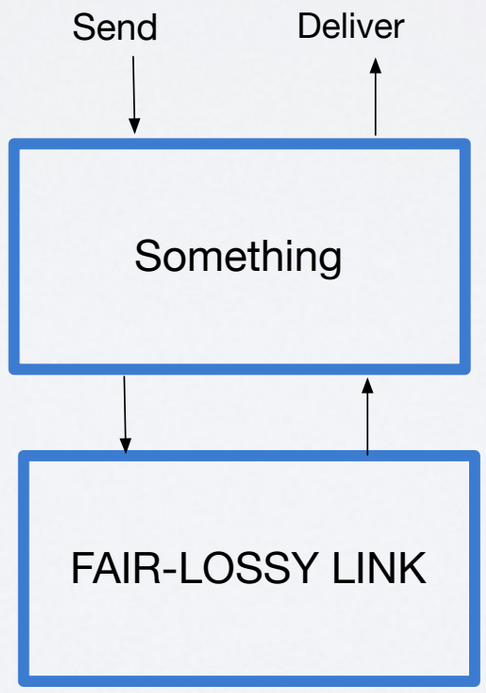
LET US GO BACK ON OUR FL-LINK

- Properties:
  - FL1: (Fair-loss) If a correct process p sends infinitely often m to a process q, then q delivers m an infinite number of times.
  - FL2: (Finite duplication) If a correct process p sends m a finite number of times to q, then q cannot deliver m an infinite number of times.
  - FL3: (No creation) If some process q delivers a message m with sender p, then m was sent by p to q.



LET US GO BACK ON OUR FL-LINK

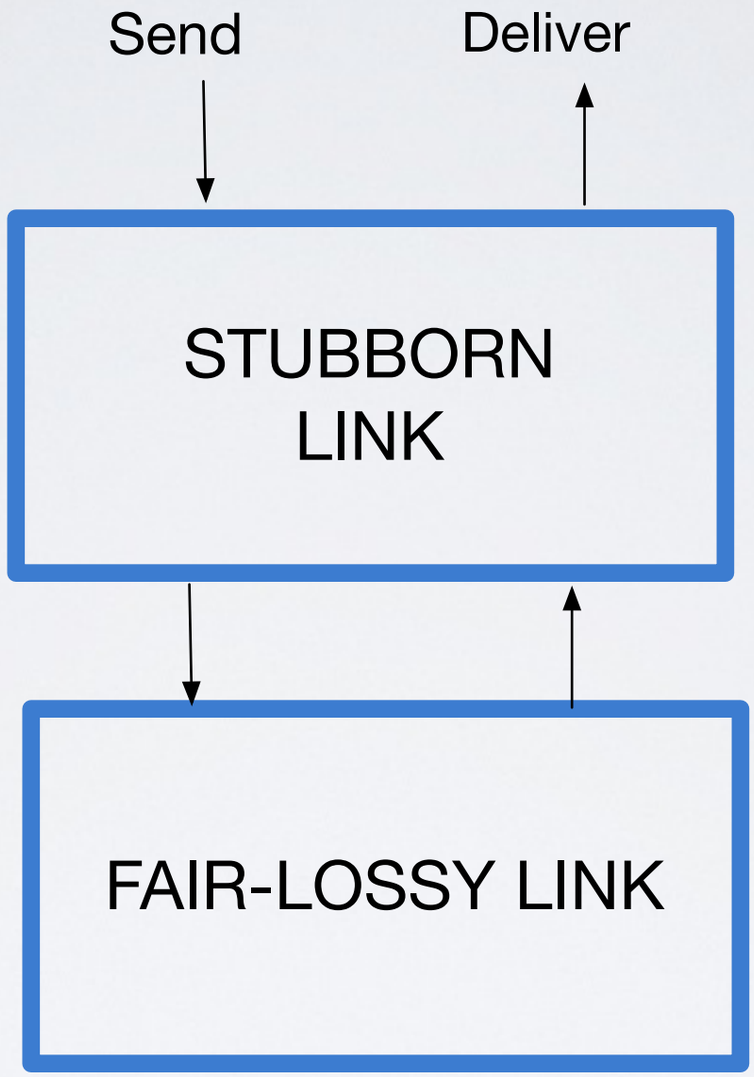
- Properties:
  - FL1: (Fair-loss) If a correct process **p sends infinitely often m** to a process q, then q delivers m an infinite number of times.
  - FL2: (Finite duplication) If a correct process p sends m a finite number of times to q, **then q cannot deliver m an infinite number of times**.
  - FL3: (No creation) If some process q delivers a message m with sender p, then m was sent by p to q.



LET US GO BACK ON OUR FL-LINK

- Let us fix:
  - FL1: (Fair-loss) If a correct process **p sends infinitely often m** to a process q, then q delivers m an infinite number of times.
  - FL2: (Finite duplication) If a correct process p sends m a finite number of times to q, **then q cannot deliver m an infinite number of times**.
- In:
  - SL1: (Stubborn-delivery) If a correct process p sends m to q, then q delivers m an infinite number of times.
- While we keep:
  - FL3: (No creation) If some process q delivers a message m with sender p, then m was sent by p to q.

OUR FIRST ALGORITHM



COMPOSITION MODEL

- The protocols we will consider in this course are presented in pseudo-code
- The pseudo code reflects a reactive computing model where
  - components of the same process communicate by exchanging events
  - the algorithm is described as a set of event handlers
  - handlers react to incoming events and possibly trigger new events.
  - Handlers are atomic. They can only be interrupted in case of a crash.

STUBBORN

- Events:
  - Requests: <Send | q,m> send a message m to process q.
  - Indication: <Deliver, p,m> delivers message m from process p.
- Properties:
  - SL1: (Stubborn-delivery) If a correct process p sends m to q, then q delivers m an infinite number of times.
  - FL3: (No creation) If some process q delivers a message m with sender p, then m was sent by p to q.

STUBBORN

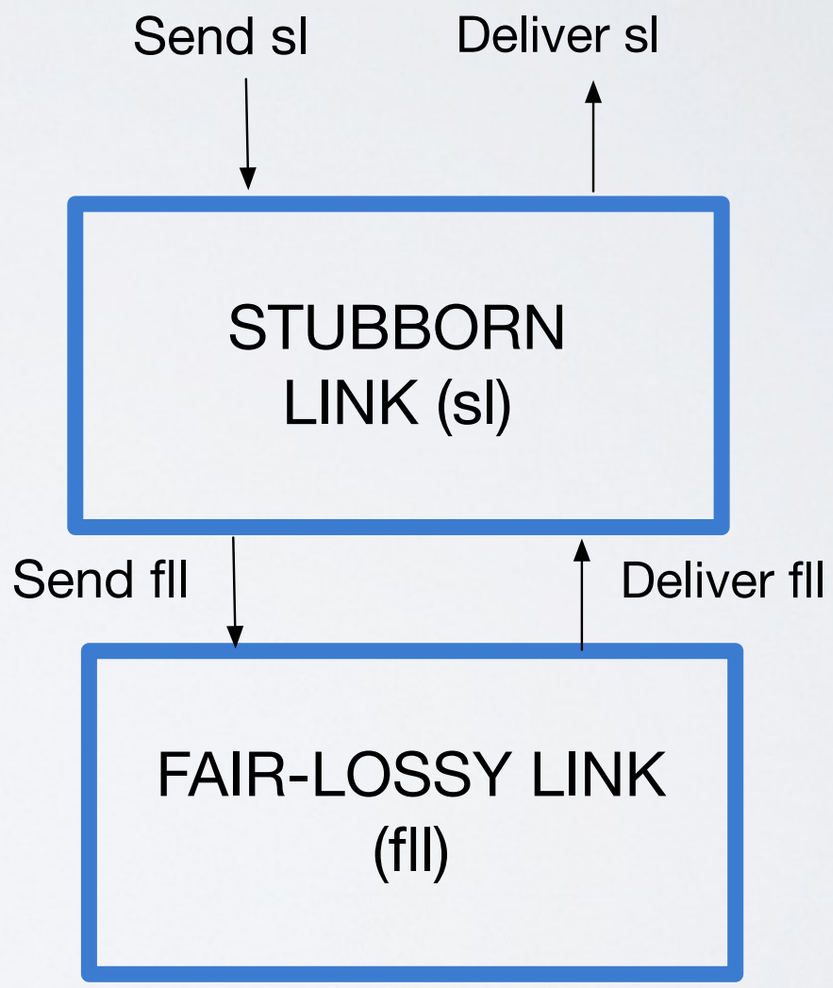
- Properties:
  - SL1: (Stubborn-delivery) If a correct process p sends m to q, then q delivers m an infinite number of times.
  - FL3: (No creation) If some process q delivers a message m with sender p, then m was sent by p to q.

upon event $\langle\ sl, Init\ \rangle$ **do**
  $sent := \emptyset$;
  $starttimer(\Delta)$;

upon event $\langle\ Timeout\ \rangle$ **do**
  forall $(q, m) \in sent$ **do**
    trigger $\langle\ fll, Send \mid q, m\ \rangle$;
  $starttimer(\Delta)$;

upon event $\langle\ sl, Send \mid q, m\ \rangle$ **do**
  trigger $\langle\ fll, Send \mid q, m\ \rangle$;
  $sent := sent \cup \{(q, m)\}$;

upon event $\langle\ fll, Deliver \mid p, m\ \rangle$ **do**
  trigger $\langle\ sl, Deliver \mid p, m\ \rangle$;



STUBBORN FORMAL PROOF

- FL3: (No creation) If some process q delivers a message m with sender p, then m was sent by p to q.

- Proof. By contradiction.
- Suppose process q executing our algorithm receives message m that was not sent by p.
- Fact 1: If q delivers a message, then it delivers here:

upon event $\langle$ *fll*, *Deliver* | p, m $\rangle$ **do**
trigger $\langle$ *sl*, *Deliver* | p, m $\rangle$;

STUBBORN FORMAL PROOF

- FL3: (No creation) If some process q delivers a message m with sender p, then m was sent by p to q.

- Proof. By contradiction.
- Suppose process q executing our algorithm receives message m that was not sent by p.
- Fact 1: If q delivers a message, then it delivers here:

upon event $\langle$ *fll*, *Deliver* | *p*, *m* $\rangle$ **do**
trigger $\langle$ *sl*, *Deliver* | *p*, *m* $\rangle$;

- Fact 1 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.

STUBBORN FORMAL PROOF

- SL1: (Stubborn-delivery) If a correct process p sends m to q, then q delivers m an infinite number of times.
- Proof. By contradiction. Suppose q delivers m a finite number of times.
- Fact 1: p sends m on fll an infinite number of times:

```
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)};
```

- Fact 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 ⟩;
```

IMPROVING STUBBORN

- Properties:
  - SL1: (Stubborn-delivery) If a correct process p sends m to q, then q **delivers m an infinite number of times**.
  - FL3: (No creation) If some process q delivers a message m with sender p, then m was sent by p to q.
- 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.
  - FL3: (No creation) If some process q delivers a message m with sender p, then m was sent by p to q.

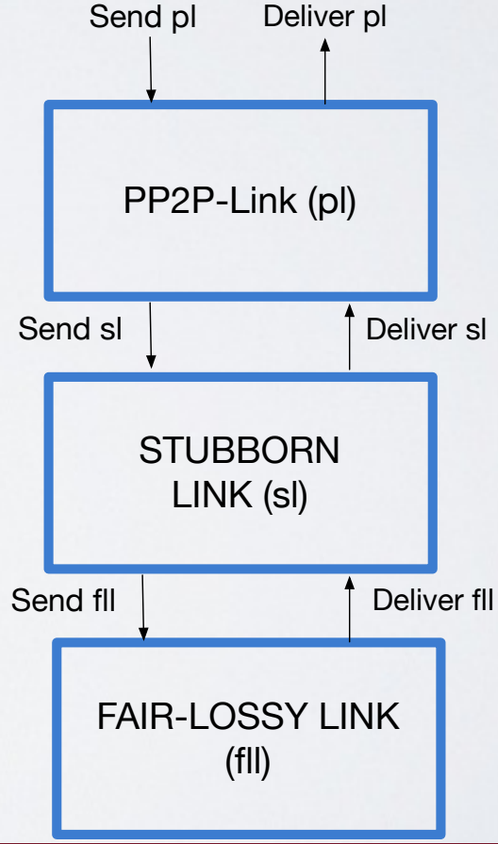
PERFECT P2P LINK

- Properties:
  - 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.
  - FL3: (No creation) If some process q delivers a message m with sender p, then m was sent by p to q.

upon event $\langle\ pl, Init\ \rangle$ **do**
    $delivered := \emptyset;$

upon event $\langle\ pl, Send \mid q, m\ \rangle$ **do**
    trigger $\langle\ sl, Send \mid q, m\ \rangle;$

upon event $\langle\ sl, Deliver \mid p, m\ \rangle$ **do**
    if $m \notin delivered$ **then**
        $delivered := delivered \cup \{m\};$
        trigger $\langle\ pl, Deliver \mid p, m\ \rangle;$



PERFECT P2P LINK - FORMAL PROOF

- FL3: (No creation) If some process q delivers a message m with sender p, then m was sent by p to q.

- Proof. By contradiction.
- Suppose process q executing our algorithm receives message m that was not sent by p.
- Fact 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 ⟩;
```

- This implies that sl delivered a message that was not created. Violates the hypothesis that sl is a stubborn.

PERFECT P2P LINK - FORMAL PROOF

- PL2: (No duplication) A message is delivered at most once
- Fact 1: The pp2p-delivery of a message is ``guarded'' by an if m \in delivered:

upon event $\langle\ sl, Deliver \mid p, m\ \rangle$ **do**
  if $m \notin delivered$ **then**
    $delivered := delivered \cup \{m\}$;
    trigger $\langle\ pl, Deliver \mid p, m\ \rangle$;

- Suppose m is delivered twice, at time t and t' (with t< t').
- 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 <pl, deliver | P, m> is executed at (or after) time t'.

PERFECT P2P LINK - FORMAL PROOF

- PL1: (Reliable delivery) If a correct process p sends m to q, then q eventually delivers m.

- Suppose, p sends m and q does not deliver it. There could be two reasons for q to not deliver m:
- Reason 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** <pl, Deliver| p, m>. This contradicts the fact that q does not deliver m.

PERFECT P2P LINK - FORMAL PROOF

- PL1: (Reliable delivery) If a correct process p sends m to q, then q eventually delivers m.

- Suppose, p sends m and q does not deliver it. There could be two reasons for q to not deliver m:
- Reason 2: the delivery handler is never triggered with <p,m>

upon event $\langle\ sl, Deliver \mid p, m\ \rangle$ **do**
  if $m \notin delivered$ **then**
    $delivered := delivered \cup \{m\}$;
    trigger $\langle\ pl, Deliver \mid p, m\ \rangle$;

- This means that sl is not stubborn. Violating our hypothesis.

HOMEWORKS 1

- Show that our stubborn algorithm does not work if we change the first property in:
  - SL1: (Stubborn-delivery) If a process p sends m to q, then q delivers m an infinite number of times. {Pay attention something is missing}.
- Show that there is no algorithm that satisfies the property above using a fair-lossy link.
- *Suppose you can use only a fixed amount k of memory (see it as having the possibility to store at most k different messages), could you implement a Perfect p2p link on a fair-lossy?

HOMEWORKS 1

- Show that our stubborn algorithm does not work if we change the first property in:
  - SL1: (Stubborn-delivery) If a process p sends m to q, then q delivers m an infinite number of times. {Pay attention something is missing}.
- The property does not specify the correctness of p. This means that even if p crashes, then q has to deliver m (at least once).
- If p crashes, then it cannot send an infinite number of messages on fll.
- If p sends a finite number of messages on fll, then it could be that no message is delivered to q.

HOMEWORKS 1

- Show that our stubborn algorithm does not work if we change the first property in:
  - SL1: (Stubborn-delivery) If a process p sends m to q, then q delivers m an infinite number of times. {Pay attention something is missing}.
- Show that there is no algorithm that satisfies the property above using a fair-lossy link.
- The same argument shows that no algorithm could satisfy the property above on a fair-lossy link.

REASONING TIME…

- Recall the following property of a perfect p2p link:
  - PL1: (Reliable delivery) If a correct process p sends m to q, then q eventually delivers m.

- Assume a system with crash failure, is there something wrong?

REASONING TIME…

- PL1: (Reliable delivery) If a correct process p sends m to q, then q eventually delivers m.

- Suppose, p sends m and q does not deliver it. There could be two reasons for q to not deliver m:
- Reason 1: m is in delivered when the delivery handler is executed:

upon event $\langle\ sl, Deliver \mid p, m\ \rangle$ **do**
  if $m \notin delivered$ **then**
    $delivered := delivered \cup \{m\}$;
    trigger $\langle\ pl, Deliver \mid p, m\ \rangle$;

- If m is in delivered then, q eventually will execute **trigger** <pl, Deliver| p, m>. This contradicts the fact that q does not deliver m.

REASONING TIME…

- Recall the following property of a perfect p2p link:
  - PL1: (Reliable delivery) If a correct process p sends m **to q and q is correct**, then q eventually delivers m.

HOMEWORKS 2

- One of your friends, Caio, is really excited about its new discovery. Caio claims to have implemented a stubborn over a fair-lossy without the need of any internal timer or infinite loop. It shows you this algorithm:

upon event INIT
  $set = \emptyset$

upon event SL, SEND($< m, q >$)
  $set = set \cup \{< m, q >\}$
  for all $< x, y > \in set$ **do**
    invoke fl Send($< x, y >$)

upon event FL, DELIVER($< m, q >$)
  invoke sl Deliver($< m, q >$)

What is wrong with his algorithm? In which executions does it work? Could you find an argument that convinces Caio on the impossibility of implementing a stubborn (and thus a perfect link) over a fair-lossy without using a timer or an infinite loop?

(note that an infinite loop is actually a timer)

HOMEWORKS 2

What is wrong with his algorithm? In which executions does it work? Could you find an argument that convinces Caio on the impossibility of implementing a stubborn (and thus a perfect link) over a fair-lossy without using a timer or an infinite loop?

(note that an infinite loop is actually a timer)

upon event INIT
  $set = \emptyset$

upon event SL, SEND($< m, q >$)
  $set = set \cup \{< m, q >\}$
  for all $< x, y > \in set$ **do**
    invoke fl Send($< x, y >$)

upon event FL, DELIVER($< m, q >$)
  invoke sl Deliver($< m, q >$)

HOMEWORKS 2

What is wrong with his algorithm? In which executions does it work? Could you find an argument that convinces Caio on the impossibility of implementing a stubborn (and thus a perfect link) over a fair-lossy without using a timer or an infinite loop?

(note that an infinite loop is actually a timer)

upon event INIT
$set = \emptyset$

upon event SL, SEND($< m, q >$)
$set = set \cup \{< m, q >\}$
for all $< x, y > \in set$ **do**
invoke fl Send($< x, y >$)

upon event FL, DELIVER($< m, q >$)
invoke sl Deliver($< m, q >$)

The algorithm does not implement PL1: (Reliable delivery). Suppose, p stubborn sends message m, and then it stops sending messages. Then the fair-lossy send is triggered only once. This implies that q could not deliver m.

HOMEWORKS 2

What is wrong with his algorithm? **In which executions does it work?** Could you find an argument that convinces Caio on the impossibility of implementing a stubborn (and thus a perfect link) over a fair-lossy without using a timer or an infinite loop?

(note that an infinite loop is actually a timer)

upon event INIT
$set = \emptyset$

upon event SL, SEND($< m, q >$)
$set = set \cup \{< m, q >\}$
for all $< x, y > \in set$ **do**
invoke fl Send($< x, y >$)

upon event FL, DELIVER($< m, q >$)
invoke sl Deliver($< m, q >$)

HOMEWORKS 2

What is wrong with his algorithm? **In which executions does it work?** Could you find an argument that convinces Caio on the impossibility of implementing a stubborn (and thus a perfect link) over a fair-lossy without using a timer or an infinite loop?

(note that an infinite loop is actually a timer)

upon event INIT
$set = \emptyset$

upon event SL, SEND($< m, q >$)
$set = set \cup \{< m, q >\}$
for all $< x, y > \in set$ **do**
invoke fl Send($< x, y >$)

upon event FL, DELIVER($< m, q >$)
invoke sl Deliver($< m, q >$)

The algorithm works only if all messages in set are fll sent an infinite number of times. Therefore, it only works in executions in which p stubborn sends an infinite number of messages.

HOMEWORKS 2

What is wrong with his algorithm? In which executions does it work? **Could you find an argument** that convinces Caio **on the impossibility of implementing a stubborn** (and thus a perfect link) **over a fair-lossy without using a timer or an infinite loop?**

(note that an infinite loop is actually a timer)

HOMEWORKS 2

What is wrong with his algorithm? In which executions does it work? **Could you find an argument** that convinces Caio **on the impossibility of implementing a stubborn** (and thus a perfect link) **over a fair-lossy without using a timer or an infinite loop?**

(note that an infinite loop is actually a timer)

Proof: Suppose you can, then in your algorithm you must have an infinite number of internal events in which you send message m over the fair-lossy link.

If you do not have an infinite loop (or a timer) and p does not receive an infinite number of input events, then it means that after a certain number of local execution steps, p will never change its internal state neither execute any actions.

Therefore, there exists at least an execution in which p executes a finite number of actions, in particular p sends m on the fair-lossy a finite number of times. This implies that is impossible for the algorithm to ensure the delivery of m by q

HOMEWORKS BIS

From the question of one of you colleagues:

- We have seen that the fairness property requires a process to be scheduled forever, so if a process crashes the execution is not fair. The reason is that a failed process will never be scheduled again for execution.

How this is **consistent** with the definition of fairness?

- The reason is that the definition of Fairness has been given for a model without failures! In case of processes failures, we have to modify the definition of fairness. That becomes:

HOMEWORKS BIS

From the question of one of you colleagues:

- We have seen that the fairness property requires a process to be scheduled forever, so if a process crashes the execution is not fair. The reason is that a failed process will never be scheduled again for execution.

How this is **consistent** with the definition of fairness?

- An execution E is fair if each **correct** process p_i executes an infinite number of local computations (Exec(i) events are not finite) and each message m sent **by a correct process or to a correct process** is eventually delivered.

HOMEWORKS 3

- Main book: page 67 exercises 2.2, 2.3
- You built a COUNTING SERVER that communicates with you using our Perfect-p2p algorithm over a stubborn link. The server stores a variable v (initially equal to 0) and it exposes three operations:
  - INC: that increases of 1 the value of v.
  - DEC: that decreases of 1 the value of v.
  - GET: that invokes <pl, Send | v, client> (client is your name as a process).

The server executes an operation when it delivers the corresponding message. As example, the increase happens when the server sees the event <pl, Deliver | INC,client>.

You act as a client and you invoke exactly three operations in the following order: <pl, Send| INC, server>,<pl, Send| INC, server>,<pl, Send| GET, server>.

You are really puzzled when, after a while, on your client you see <pl, Deliver, 1, server>. Startled, you asked again <pl, Send| GET, server>. Unfortunately, you are still waiting for an answer.

Questions: What is happening? Is there any way to fix it?

Note that, there are two possible explanations, try to find both. (Hint: you can see one of the explanation by answering to the question: How much will you wait for an answer?)