Modelli Concorrenti
e
Algoritmi Distribuiti

Introduzione algoritmi distribuiti

Modelli sincroni, asincroni e asincroni con memoria condivisa

Alcuni lucidi sono stati liberamenti tratti da:

- Algoritmi Avanzati a.a. 2010/2011. prof. Rossella Petreschi
- Distributed Algorithms. Fall, 2011. prof. Nancy Lynch

Algoritmi Distribuiti

- ***Definizione*** : algoritmi progettati per lavorare in sistemi distribuiti
- ***si occupano di***
  - comunicazione
  - gestione dati e risorse
  - sincronizzazione
  - consenso
- ***presentano***
  - attività concorrente
  - indeterminatezza nella temporizzazione, ordine di avvenimento eventi,
  - **possibilità di guasti e/o recuperi di processori e/o canali di comunicazione**
- ***quindi sono***
  - difficili da progettare
  - difficili da analizzare (correttezza e complessità).

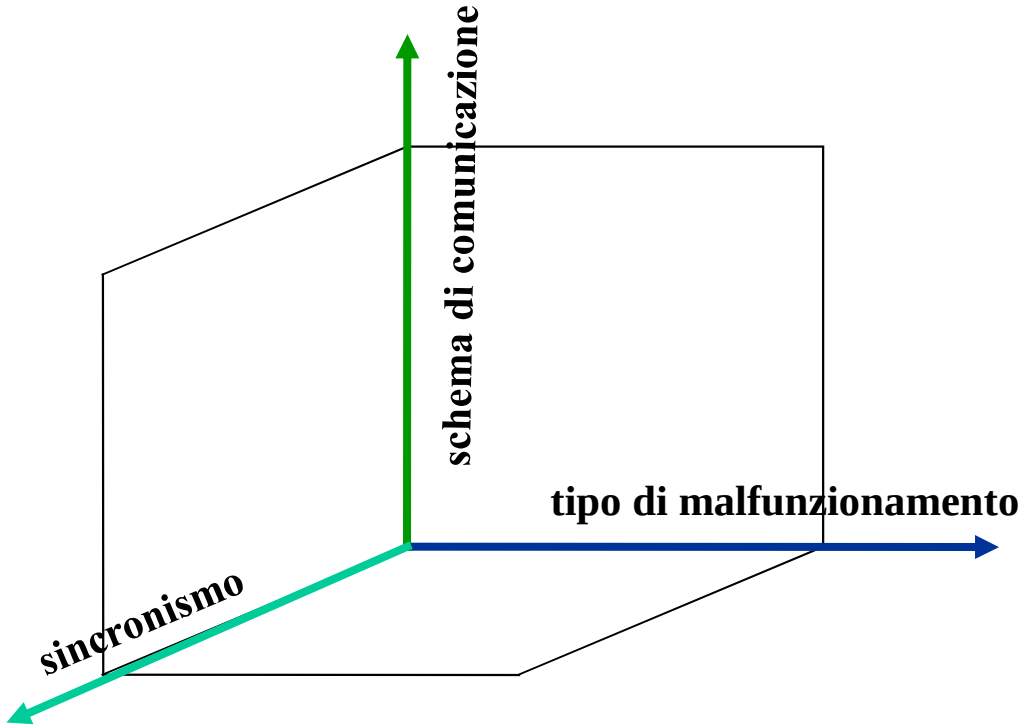
Modelli di sistemi distribuiti

Come si possono caratterizare i *sistemi distribuiti*?

- modalità di comunicazione (memoria comune o messaggi)
- sincronia/asincronia di esecuzione e interazioni
- comportamento in caso di malfunzionamenti hardware o software

Modelli di sistemi distribuiti

Occorre considerare tre dimensioni:



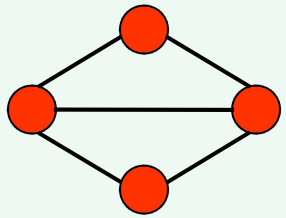
Modelli di sistemi distribuiti
Schemi di comunicazione

Schemi di comunicazione

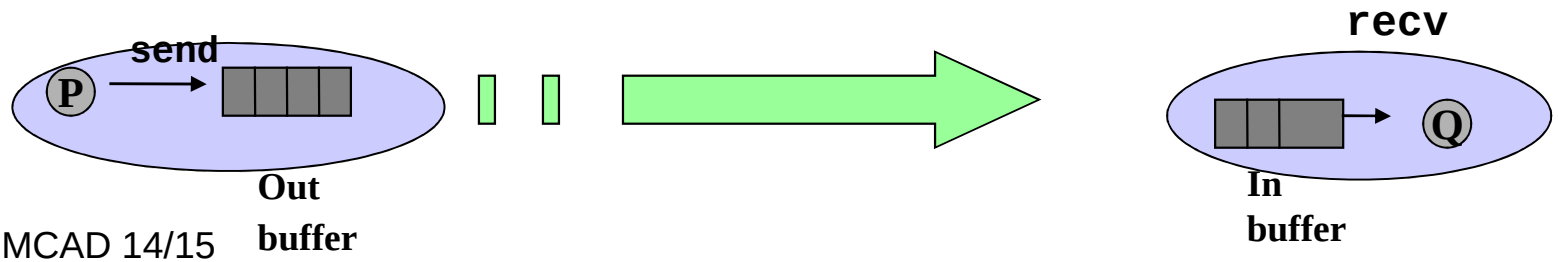
- **rete a scambio di messaggi**
  - connessioni punto-a-punto
  - topologie specifiche (anello, stella, broadcast,..)
- **memoria condivisa**

Modelli di sistemi distribuiti: reti punto-a-punto

- **Reti punto-a-punto** si possono modellare come ***grafi***, i cui *nodi* rappresentano i *processi* e gli *archi* le *linee di comunicazione* (eventualmente bidirezionali)



- Si suppone che i processi comunichino tramite primitive tipo `send/recv`
  - `send` non bloccante
  - spesso si ipotizza un grafo completo
  - linee virtuali non necessariamente fisiche



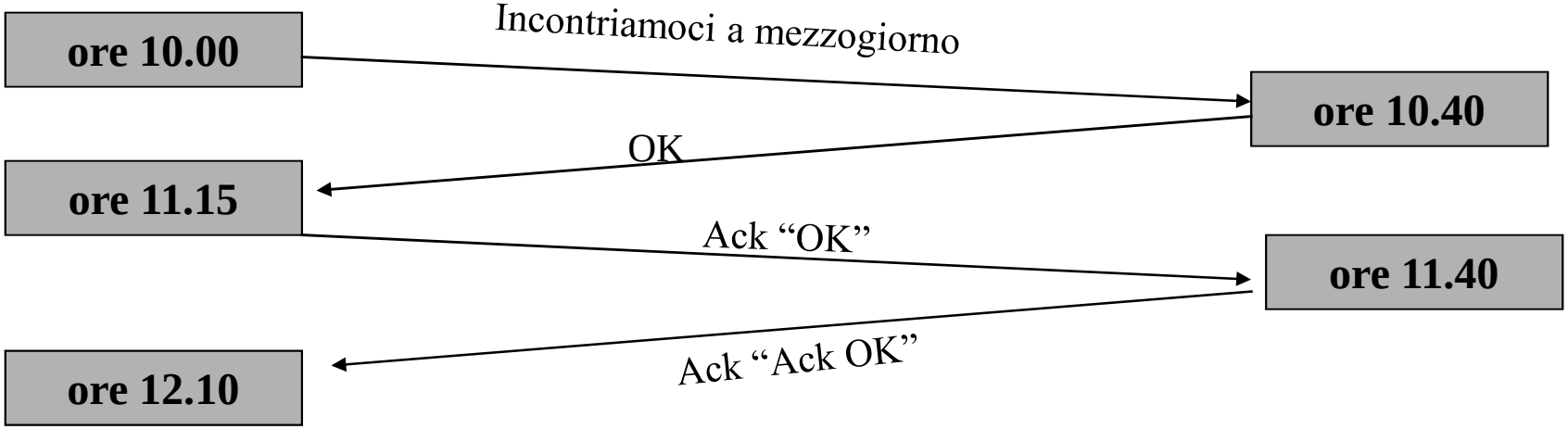
Modelli di sistemi distribuiti: reti punto-a-punto

Caratteristiche delle *reti punto-a-punto failure free*

- *Per quanto riguarda i processi:*
  - Se un processo non ha ancora raggiunto il suo stato finale, "*prima o poi*" eseguira' certamente un altro **step** *(liveness)*
- *Per quanto riguarda la comunicazione:*
  - P riceve un messaggio *m* da Q al piu' una volta e solo se Q glielo ha precedentemente inviato *(safety)*
  - Se P manda un messaggio *m* a Q, Q "*prima o poi*" lo riceverà certamente (*liveness*)

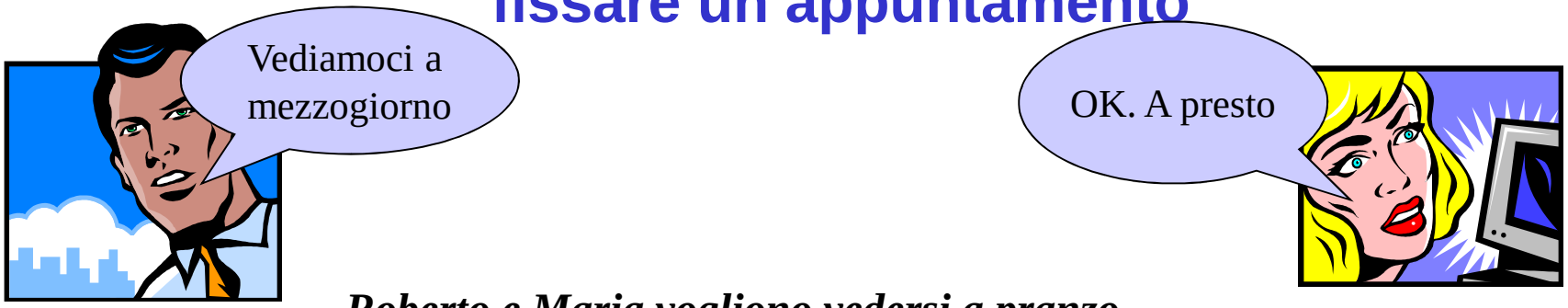
Modelli di sistemi distribuiti
Sincrono/Asincrono
fissare un appuntamento

Roberto e Maria vogliono vedersi a pranzo
e comunicano via e-mail

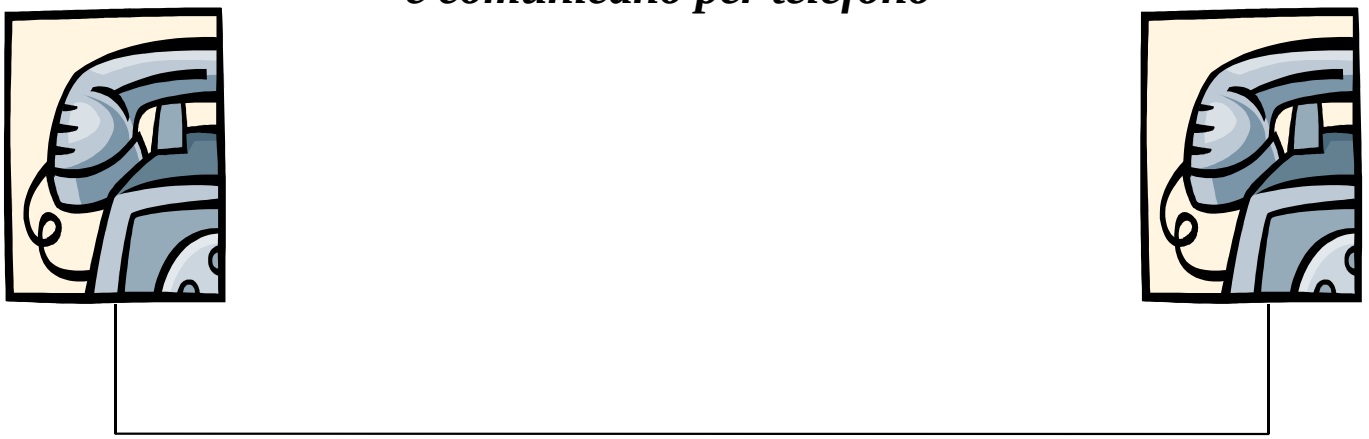


caratteristiche del sistema:
non esiste memoria comune
trasmissione non sicura (non si sa se il messaggio e' pervenuto correttamente..)
ritardo di trasmissione non noto e non limitato

Modelli di sistemi distribuiti - Sincrono/Asincrono
fissare un appuntamento



Roberto e Maria vogliono vedersi a pranzo e comunicano per telefono



caratteristiche del sistema:
non esiste memoria comune
ognuno dei due sente le parole dell'altro con un ritardo noto e limitato
se la comunicazione cade entrambi ne vengono a conoscenza

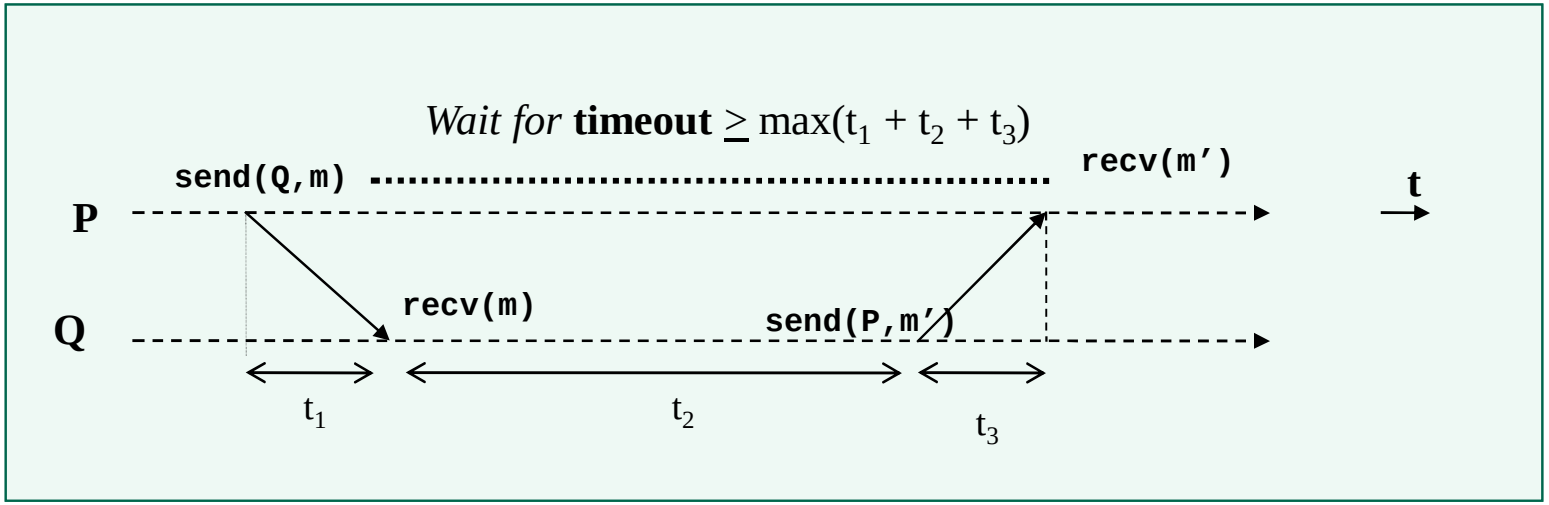
Modelli di sistemi distribuiti Sincrono/Asincrono

Modello Sincrono

- E' noto un ***limite di tempo massimo*** per l'esecuzione di uno **step locale** da parte di un processo
- E' noto un ***limite di tempo massimo*** di **ritardo di trasmissione**
- Si può supporre che i processi abbiano ***orologi fisici*** perfettamente ***sincronizzati*** (o approssimativamente sincronizzati, cioè all'interno di un intervallo di tempo ε)

Modelli di sistemi distribuiti
Sincrono/Asincrono

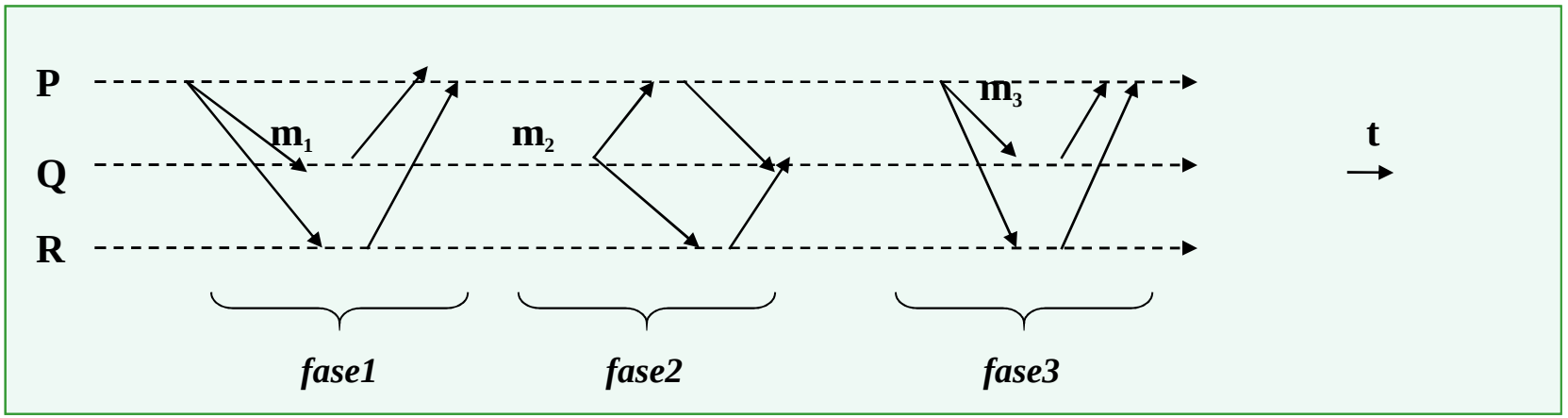
Modello sincrono- Conseguenze



Si possono usare **timeout** per scoprire malfunzionamenti di comunicazione

Modelli di sistemi distribuiti
Sincrono/Asincrono

Modello sincrono: conseguenze



Si puo' organizzare la computazione in **fasi**:

- invio di messaggi a un insieme di processi P
- ricezione di messaggi della fase da parte dei processi di P
- cambio di stato

Modelli di sistemi distribuiti
Sincrono/Asincrono

Modello Asincrono

- anche se il tempo d'esecuzione di uno step locale e' *finito*, non è noto *nessun limite di tempo massimo* per la sua esecuzione
- non è noto *nessun limite di tempo massimo* per la trasmissione del messaggio
- non si può supporre l'esistenza di *orologi fisici sincronizzati*

Il modello asincrono è il più generale: un algoritmo progettato per sistemi asincroni funziona anche in sistemi sincroni

Modelli di sistemi distribuiti Sincrono/Asincrono

Sfortunatamente:

alcuni problemi computazionali non hanno soluzione in *sistemi asincroni*

- se si desidera una soluzione *fault tolerant*
- se si vuole usare un *algoritmo deterministico*

quindi

per alcuni problemi di deve ricorrere a

- sistemi sincroni
- algoritmi probabilistici
- …

Modelli di sistemi distribuiti
Tipi di malfunzionamenti

Malfunzionamento di processi : CRASH

...un processo si blocca prima di raggiungere lo stato finale....

Processo fallito	***non soddisfa la proprieta' di liveness dei processi***
Processo corretto	***soddisfa la liveness dei processi***

Malfunzionamento di comunicazione: PERDITA DI MESSAGGI

..un messaggio inviato non viene ricevuto...

Linea fallita	***non soddisfa la liveness della comunicazione***
Linea corretta	***soddisfa la liveness della comunicazione***

Modelli di sistemi distribuiti

Per descrivere un algoritmo distribuito, occorre specificare:

- ***grafo di comunicazione*** (completo, anello, ecc…)
- possibilità di ***fallimento di processi***
- possibilità e tipo di ***fallimenti di comunicazione***
- ipotesi sul ***numero massimo*** di processi falliti
  e/o di comunicazioni fallite
- tipo di ***sincronismo***

E' cruciale essere chiari e precisi su questi aspetti poiché essi hanno una ricaduta essenziale su:

- progetto e funzionamento degli algoritmi distribuiti
- correttezza algoritmo
- analisi complessità computazionale

Ricerca sugli Algoritmi Distribuiti

Area attiva da oltre 30 anni.

Si occupa di problemi *astratti* che provengono da problemi *pratici*, sorti nel campo della programmazione in rete ed in ambito multiprocessore.

- ***Teoria Statica***
  - Assume un numero fisso di "nodi"
  - L'insieme dei processi partecipanti, la configurazione della rete è in generale nota

    [testo Lynch, Attiya, Welch]
- ***Teoria Dinamica***
  - Client/server, peer-to-peer, wireless, mobile ad hoc
  - Participanti possono inserirsi, lasciare la rete, spostarsi..
- ***Teoria per architetture parallele particolari***
  - Processori multicore
  - Memoria transazionale

Alcune note sul testo di N. Lynch

- ***Approccio teorico, matematico***

  - Definizione *formale* sistema distribuito:
    - Modello sincrono
    - Modello asincrono con memoria condivisa
    - Modello asincrono

  - Definizione del problema astratto.
  - Descrizione algoritmo
  - Traduzione algoritmo nel formalismo del modello.
  - Analisi di complessità.
  - Risultati di correttezza.

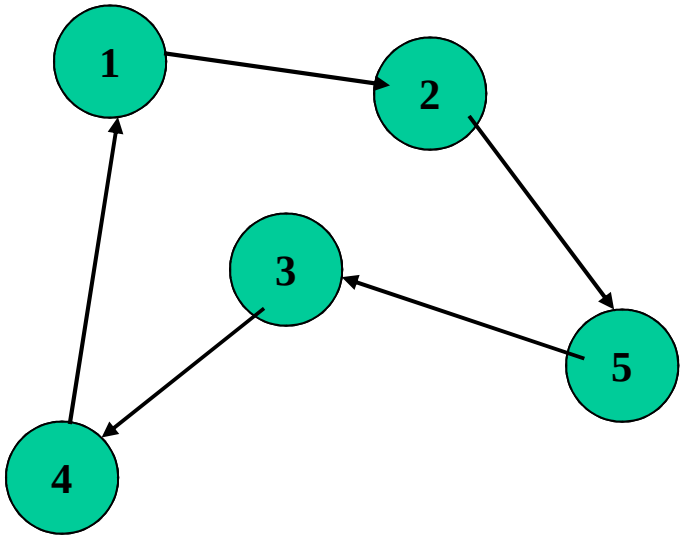
Modelli di sistemi distribuiti
Modello sincrono

Grafo orientato $G = (V,E)$

I nodi $v \in V$ rappresentano i processi, gli archi $e \in E$ rappresentano le linee di comunicazione

distanza(i,j): lunghezza del cammino minimo tra i e j in G

diametro (i,j): distanza massima tra tutte le coppie di vertici (i,J) in G



Modelli di sistemi distribuiti
Modello sincrono

Processi

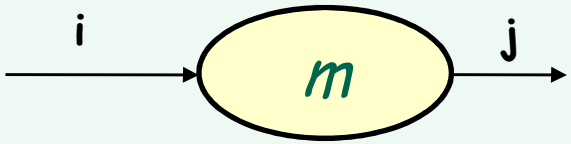
Ogni processo è modellato da una quaterna:

(*start*, *states*, *msgs*, *trans*)

states	insieme di stati
start	insieme degli stati iniziali (sottoinsieme non vuoto di *states*)
msgs	funzione di generazione dei messaggi (ad ogni stato associa un insieme di messaggi in uscita)
trans	funzione di transizione (ad ogni coppia <stato, messaggi ricevuti> associa uno stato)

Modelli di sistemi distribuiti
Modello sincrono

Linee di comunicazione



Ogni arco rappresenta un *canale*, cioè un posto per un unico messaggio $m \in M \cup \{null\}$ (assenza di messaggio).

Modelli di sistemi distribuiti
Modello sincrono

Esecuzione

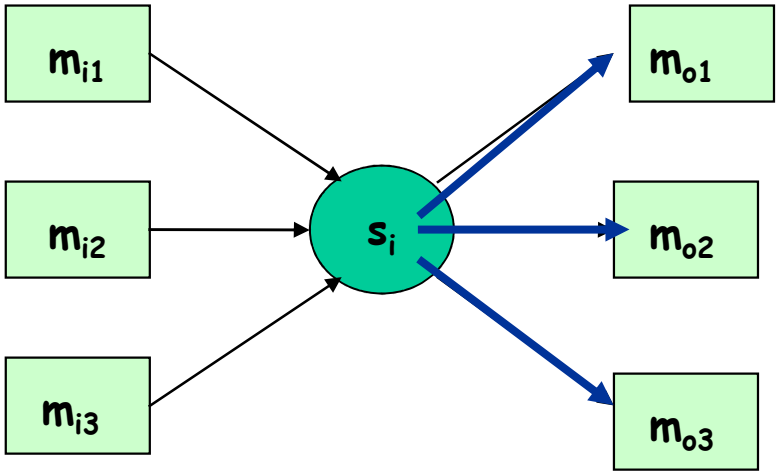
Nel modello sincrono, l'esecuzione procede in ***fasi.*** Inizialmente tutti i processi sono in uno stato iniziale e tutti i canali sono vuoti. Ogni fase è costituita da due ***step.***

Step1. Ogni processo applica la propria funzione di generazione dei messaggi (***msg***) e invia i messaggi generati nei propri canali di uscita.

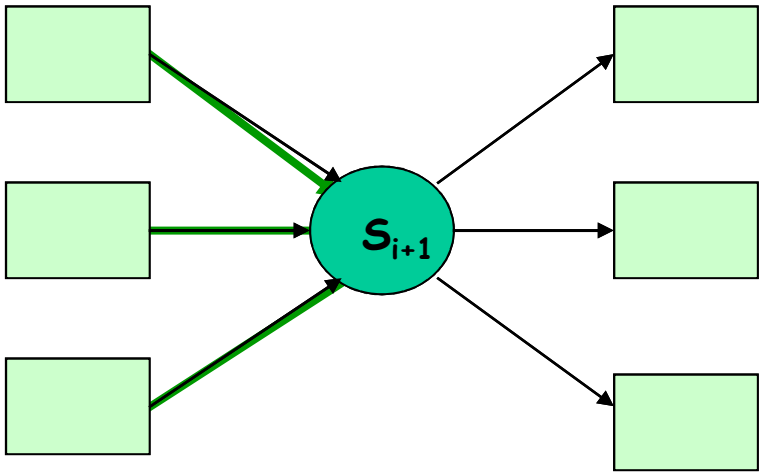
Step2. Si applica la funzione di transizione (***trans***) alla coppia <stato, messaggi in ingresso> per ottenere il nuovo stato; si rimuovono i messaggi dai canali.

Modelli di sistemi distribuiti
Modello sincrono

STEP1



STEP2



L'assenza di messaggi sarà implementata nei sistemi reali usando un timeout (nel modello con un messaggio *null*)

Modelli di sistemi distribuiti
Modello sincrono

- **Malfunzionamento processi**
  - **Crash**. Il processo si blocca prima o dopo lo step1 oppure in un qualsiasi punto dello step2
  - **Fallimento bizantino**. Si generano messaggi e stati arbitrari

    (non secondo le funzioni *trans* e *msgs*)

- **Malfunzionamento comunicazioni**
  - **Canali**.Perdita di messaggi

Modelli di sistemi distribuiti
Modello sincrono

Modello formale di esecuzione sincrona

Una esecuzione sincrona di un sistema distribuito e' definita da una sequenza infinita:

$$C_0, M_1, N_1, C_1, M_2, N_2, C_2, \ldots,$$

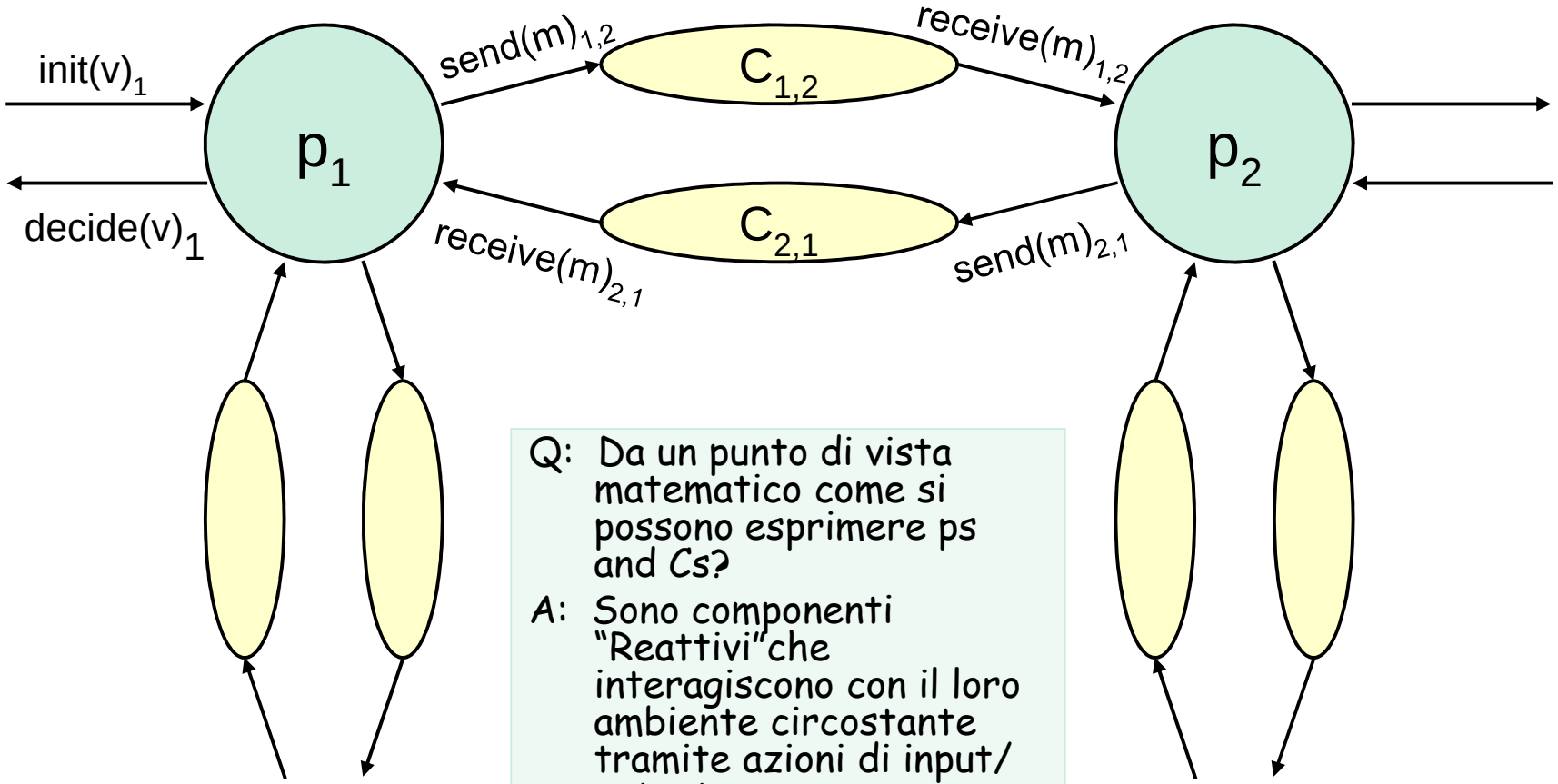
C_r	assegnamento di stati alla fase *r*, cioe' dopo r fasi
M_r	messaggi inviati alla fase *r*
N_r	messaggi ricevuti alla fase r
$M_r \neq N_r$	significa che ci sono state perdite di messaggi

Reti asincrone
Processi e canali

- Nessuna assunzione sul tempo: non ci sono fasi!

- Due tipi di modelli asincroni:
  - **Reti asincrone**:
    - Processi comunicano tramite canali
  - **Sistemi asincroni con memoria condivisa**:
    - Processi comunicano tramite oggetti condivisi(variabili)

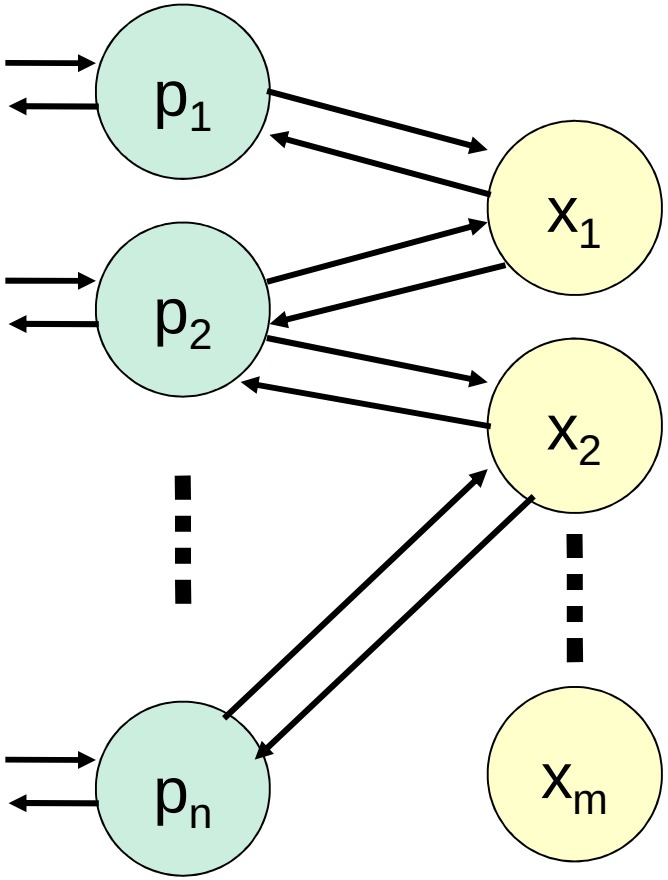
Reti asincrone
Processi e Canali



Q: Da un punto di vista matematico come si possono esprimere ps and Cs?

A: Sono componenti "Reattivi" che interagiscono con il loro ambiente circostante tramite azioni di input/ output.

Reti asincrone con Memoria Condivisa
Processi e Oggetti



Anche in questo caso i processi e gli oggetti sono componenti "reattivi".

Così cerchiamo un unico modello generale per descrivere entrambi.

Reti asincrone
Processi e canali

Modelliamo processi, canali e oggetti come **automi**

Eseguono **azioni e cambiano stato**.

- Sono reattivi
  - Interagiscono con l'ambiente circostante tramite azioni di input/output.
  - Le interazioni non sono descrivibili solo come funzioni da input all'output, ma sono più flessibili.

Esecuzione

- Definita da una sequenza di azioni
- Semantica Interleaving

Comportamento esterno (tracce):

- Osserviamo azioni esterne
- Le azioni interne e gli stati sono nascosti
- Problemi specificano quali siano le tracce permesse

Modelli asincroni: Automi di Input/Output

- Gli automi forniscono un **modello matematico generale** per componenti reattivi.
- Adatto per descrivere i sistemi in modo **modulare**:
  - Rende possibile la descrizione di componenti individuali e dice come **comporli** per ottenere sistemi più grandi .
  - Rende possibile la descrizione di sistemi a vari livelli di astrazione:
    - Implementazioni dettagliate vs. descrizione di algoritmi a più alto livello.
    - Algoritmi ottimizzati vs. versioni più semplici, non ottimizzate.
- Supporta tecniche standard di **dimostrazioni:**
  - **Invarianti**
  - **Relazioni di simulazione**
  - **Ragionamenti composizionali** (la prova di proprietà di componenti individuali usata per dimostrare proprietà dell'intero sistema).

Modelli asincroni: Automi di Input/Output

- Sistemi basati su transizioni di stato
- Transizioni etichettate con il nome delle azioni
- Azioni classificate come **azioni di input**, **azioni di output** e **azioni interne**
- Le azioni di input e di output sono **esterne**.
- Le azioni di output e le azioni interne sono **controllate localmente**.

Modelli asincroni: Automi di Input/Output

- **signature** = (in, out, int)
  - insiemi disgiunti di azioni di input, output e interne
    azioni = in $\cup$ out $\cup$ int
  - ext = in $\cup$ out
  - local = out $\cup$ int
- **states**: non necessariamente un insieme finito
- **start** $\subseteq$ states
- **trans** $\subseteq$ states $\times$ acts $\times$ states
- **tasks** partizioni di azioni controllate localmente

Modelli asincroni: Automi di Input/Output

- Uno **step** di un automa è un elemento della transizione .
- Un azione π è detta **abilitata (enabled)** in uno stato s se esiste uno step:

  (s, π, s') per un qualche stato s'.
- Gli automi I/O devono essere **input-enabled**, cioè in ogni stato sono sempre abilitate tutte le azioni di input. Questa assunzione cattura l'idea che un automa non può controllare gli eventi di input. Eventuali restrizioni possono essere descritte definendo l'ambiente circostante come un automa.

- I tasks corrispondono all'esecuzione di "threads di controllo". (Sono usati per definere la fairness).

Modelli asincroni: Esecuzioni

- Un automa di I/O esegue nel seguente modo:
  - Parte da uno stato di start.
  - Ripetutamente applica uno step $\in$ **trans**, passando dallo stato corrente ad un altro stato
- Formalmente , un' ***esecuzione*** è una sequenza, finita o infinita di step:
  - $s_0\ \pi_1\ s_1\ \pi_2\ s_2\ \pi_3\ s_3\ \pi_4\ s_4\ \pi_5\ s_5$ ... (se finita deve terminare con uno stato)
  - s_0 is a start state
  - $(s_i, \pi_{i+1}, s_{i+1})$ è uno step $\in$ **trans**

Modelli asincroni
Frammenti di esecuzione

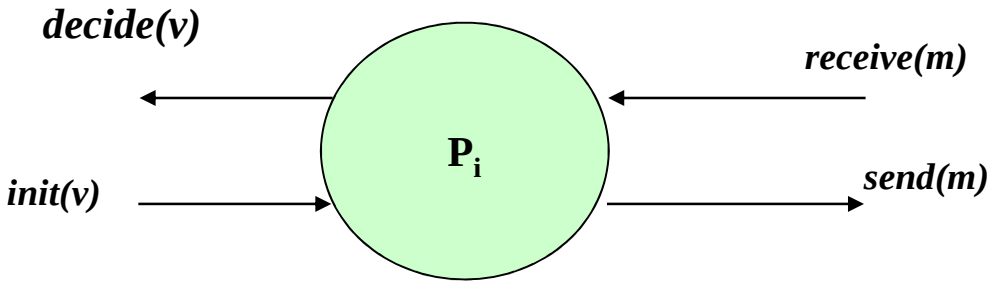
- Un automa di I/O esegue nel seguente modo:
  - Parte da uno stato di start.
  - Ripetutamente applica una step $\in$ *trans* dallo stato corrente ad un altro stato
- Formalmente , un' *esecuzione* è una sequenza, finita o infinita:
  - $s_0\ \pi_1\ s_1\ \pi_2\ s_2\ \pi_3\ s_3\ \pi_4\ s_4\ \pi_5\ s_5$ ... (se finita deve terminare con uno stato)
  - s_0 is a start state
  - $(s_i, \pi_{i+1}, s_{i+1})$ è uno step $\in$ *trans*

frammento di esecuzione

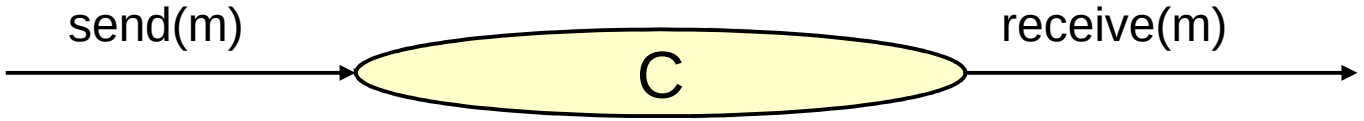
Modelli asincroni
Processi come automi di Input/Output

Un processo viene modellato come una automa I/O:

tra le azioni di output esiste anche l'azione: ***send (m)$_{i,j}$***
(*j* è un processo che riceve dal nodo *i* e *m* è un messaggio)
tra le azioni di input esiste anche l'azione ***receive (m)$_{j,i}$***
(*j* è un processo che invia al nodo *i* e *m* è un messaggio)



Modelli asincroni
Canali come automi di I/O



- Canale è un link orientato tra due processi , realizzato da una coda. Esiste un alfabeto fisso di messaggi M.
- **signature**
  - Azioni di input: send(m), m ∈ M
  - Azioni di output: receive(m), m ∈ M
  - Nessuna azione interna
- **states**
  - varie configurazioni della coda : inizialmente vuota
- **trans**
  - send(m)
    - Effetto : aggiunge m alla fine della coda
  - receive(m)
    - Precondizione : m è in testa alla coda
    - Effetto: rimuove la testa della coda
- **tasks**
  - Tutte le azioni receive

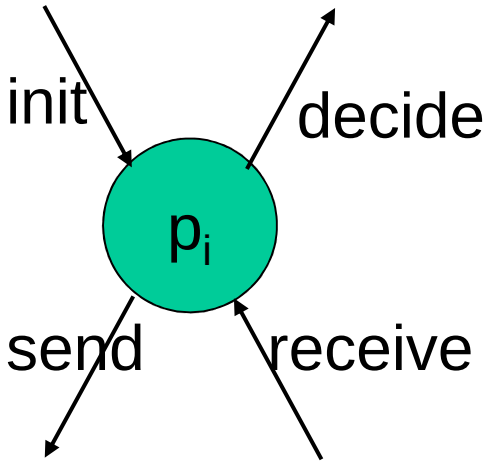
Modelli asincroni
Stati raggiungibili e Invarianti

- Uno stato è **raggiungibile** se è presente in una qualche esecuzione
- Un **invariante** è un predicato che è sempre vero in ogni stato raggiungibile.
- Gli invarianti sono lo strumento più importante per provare le **proprietà** degli algoritmi concorrenti e distribuiti.
-
- Solitamente sono provati per **induzione** sulla lunghezza delle esecuzioni.

Modelli asincroni
Tracce

- Permettono di focalizzare solo sul comportamento esterno dei sistemi.
- Utili per definire la correttezza
- La **traccia** di una esecuzione è la sottosequenza delle azioni esterne presenti nella esecuzione. Non vengono presi in considerazione nè gli stai nè le azioni interne.
- Viene denotata con **trace(α),** dove α è una esecuzione.
- Modella il comportamento osservabile.

Modelli asincroni
Automi di I/O: un esempio



- Consideriamo un protocollo di consenso.

- I valori di input arrivano dall'esterno
- I processi inviano e ricevono i valori e li memorizzano ciascuno in un vettore (un valore per ciascuno dei processi).
- Quando il vettore è riempito il processo prende una decisione sull'output in funzione della configurazione del vettore.
- Si possono verificare nuovi input , cambi di valori, send e output ripetutamente.

- Tasks per:
  - Inviare a ciascun vicino i valori .
  - Inviare decisioni finali.

Modelli di sistemi distribuiti
Modello asincrono

Esempio

Processo P_i (descritto da un I/O automa)

V e' un insieme di valori,

azioni di input:	$init(v)_i$	$v \in V$
	$receive(v)_{j,i}$	
azioni di output:	$decide(v)_i$	
	$send(v)_{i,j}$	
states	*val,* un vettore [1, . . .,n] di elementi di V	
transizioni		
$init(v)_i$ $v \in V$	*val[i]:= v*	
$receive(v)_{j,i}$	*val[j] := v*	
$send(v)_{i,j}$	*nessuna transizione, precond. val(i) := v*	
$decide(v)_i$	*nessuna transizione, precond.* *v = val[1]+ …+ val[n]*	

Modelli di sistemi distribuiti
Modello asincrono

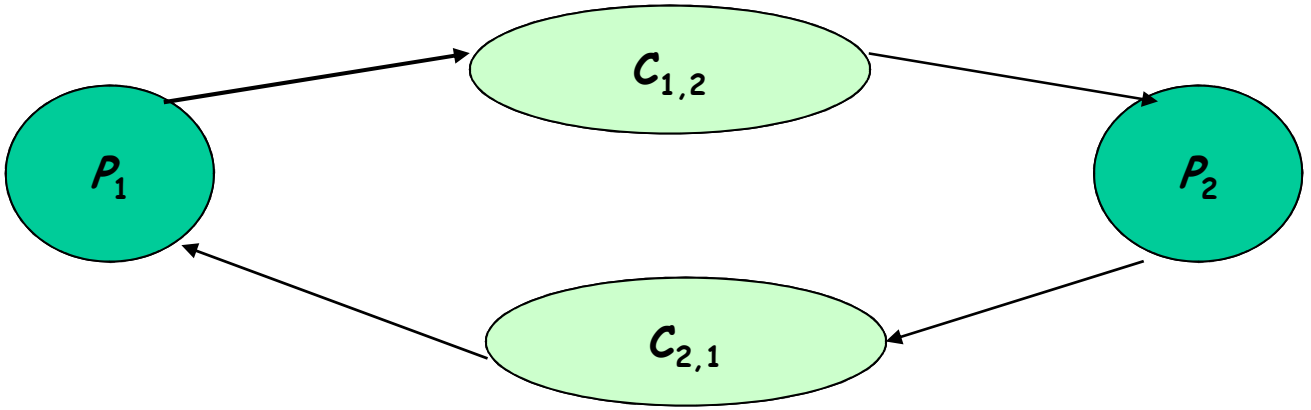
Esempio

Canale $C_{i,j}$
M e' l'alfabeto dei messaggi

azioni di input:	$send(m)_{i,j}$	$m \in M$
azioni di output:	$receive(m)_{j,i}$	
stati	*queue* una coda FIFO di elementi di M,inizialmente vuota	
transizioni		
$send(m)_{i,j}$	aggiunge *m* alla coda	
$receive(m)_{j,i}$	rimuove il primo elemento della coda	
task		
{receive(m)$_{j,j}$: m$\in$ M}		

Modelli di sistemi distribuiti
Modello asincrono

Consideriamo un sistema formato da due processi P_1 e P_2 e due canali $C_{1,2}$ e $C_{2,1}$.



Traccia:

$init(2)_1$, $init(1)_2$,$send(2)_{1,2}$, $receive(2)_{1,2}$,$send(1)_{2,1}$, $receive(1)_{2,1}init(4)_1$, $init(0)_2$

$decide(5)_1$, $decide(2)_2$

Modelli di sistemi distribuiti
Modello asincrono

Esecuzioni del canale

[λ], *send* $(1)_{i,j}$,[1], *receive* $(1)_{i,j}$,[λ], *send* $(2)_{i,j}$,[2], receive$(2)_{i,j}$,[λ]

[λ], *send* $(1)_{i,j}$,[1], *receive* $(1)_{i,j}$,[λ], *send* $(2)_{i,j}$,[2]

[λ], *send* $(1)_{i,j}$,[1], *send* $(1)_{i,j}$,[11], *send* $(1)_{i,j}$,[111], ….

La prima esecuzione è fair poiché nessuna *receive* è abilitata nello stato finale,

La seconda esecuzione non è fair perché è finita e nello stato finale una azione *receive* è abilitata

La terza esecuzione non è fair perché è infinita e in ogni punto della computazione l'azione *receive* è abilitata

Modelli asincroni
Fairness

- Un task T (insieme di azioni) corrisponde a un "thread di controllo".
- Usato per definire esecuzioni "fair": un task che è continuamente abilitato deve poter eseguire uno step.
- Formalmente , l' esecuzione α è **fair per il task T** se vale una delle seguenti condizioni:
  - α è finita e T non è abilitato nello stato finale di α.
  - α è infinita e contiene una quantità infinita di eventi in T.
  - α è infinita e contiene una quantità infinita di stati in cui T non è abilitato
- L' esecuzione di A è **fair** se è fair per tutti i tasks di A.
- Una traccia di A è **fair** se è la traccia di una esecuzione fair di A.

Esempio di uso degli invarianti

- Si supponga di avere un sistema costituito da due processi, P_1 e P_2, ciascuno con una sola variabile locale val.
- Inizialmente P_1.val = 1, P_2.val = 2.
- I due processi comunicano tramite i canali C_{12} ande C_{21}: **send(v)$_{1,2}$**, **receive(v)$_{1,2}$**, **send(v)$_{2,1}$**, **receive(v)$_{2,1}$**.

- Transizioni:
  - **send(v),** where v = val, in ogni istante.
  - Alla ricezione di **receive(v):** val := v + 1.

- **Invariante 1:** In ogni stato s, P_1.val è dispari e P_2.val è pari
- **Proof:** Per induzione sulla lunghezza dell' esecuzione
  - **Base**: Vero per ipotesi nello stato s_0
  - **Passo induttivo** : Si considera l' iesimo step (s_i, π_{i+1}, s_{i+1}). Supponendo l' invariante vero nello stato s_i, vogliamo dimostrarlo vero nello stato s_{i+1} . La prova è ovvia analizzando i vari casi dell'azione π_{i+1} .

Modelli di sistemi distribuiti
Malfunzionamenti

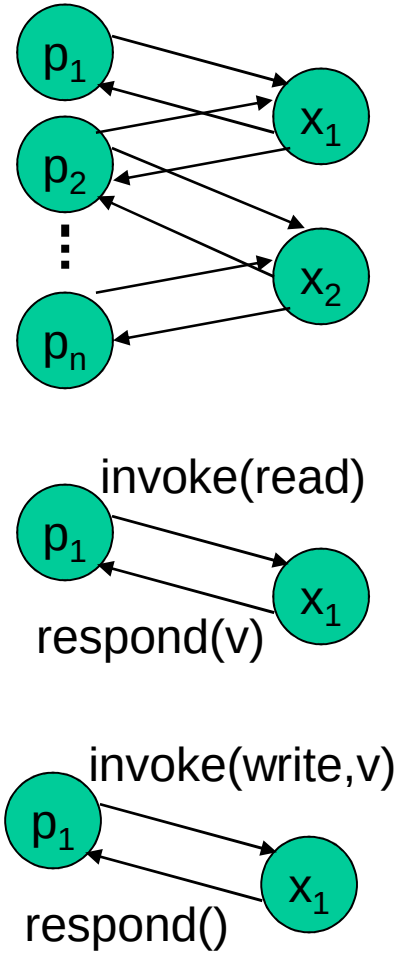
- **Malfunzionamento processi**
  - **Crash**. Il crash è modellato includendo tra le azioni di input di un processo dell'azione di input *stop*
  - **Fallimento bizantino**. Il fallimento bizantino è modellato permettendo di sostituire l'automa P con un qualsiasi automa con la stessa interfaccia esterna.
- **Malfunzionamento comunicazioni**
  - **Canali**.Perdita di messaggi

Sistemi asincroni con memoria condivisa: sistemi ASM

- La teoria dei sistemi ASM ha molto in comune con la teoria dei sistemi asincroni, si usano algoritmi simili e ci sono analoghi risultati di impossibilità, anche in caso di malfunzionamento.
- Storicamente la teoria dei sistemi ASM nasce prima, nell'ambito dei primi sistemi operativi multitasking
- Attualmente i modelli ASM si applicano ad ambienti multiprocessori con memoria condivisa.

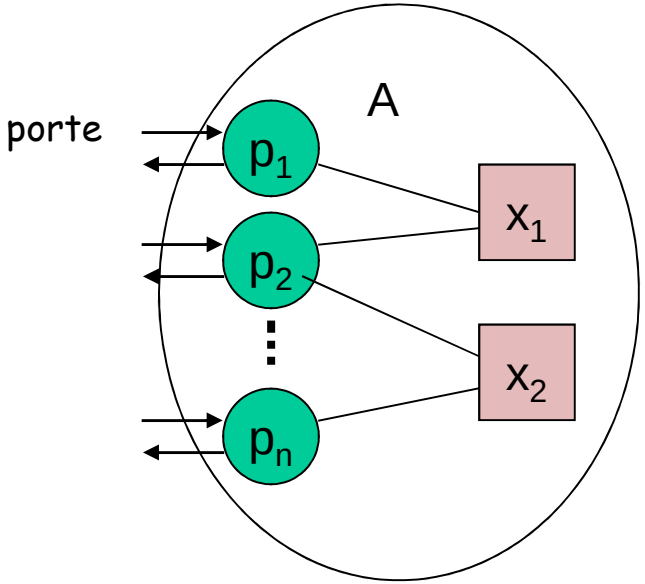
Un automa ASM, prima versione

- Processi e variabili (oggetti) modellati come **automi**
- Frecce:
  - Rappresentano richieste e risposte per operazioni sulle variabili; possono essere modellate come azioni di input/output.
- Il modello ha una **granularità fine** e può descrivere:
  - Il **ritardo** tra la richiesta e la risposta
  - L' "overlapping" tra le operazioni
- E' possibile usare un modello più semplice in cui:
  - Le variabili eseguono le richieste nell'ordine di richiesta una alla volta
  - In realtà ogni operazione collassa in un unico step



Un automa ASM, seconda versione

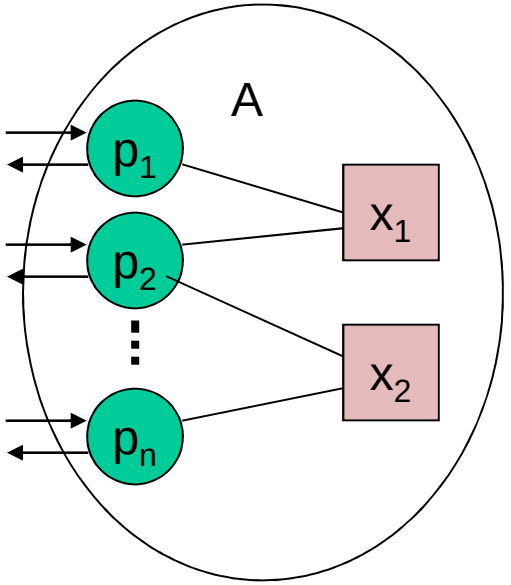
- Un **unico automa *A***.
- Azioni esterne alle "porte".
- Ogni processo P_i ha:
  - Un insieme di stati: **$states_i$**.
  - Un sottoinsieme di stati di partenza: **$start_i$**.
- Ogni variabile x ha:
  - Un insieme di valori che può assumere: **$values_x$** .
  - Un sottoinsieme di valori iniziali **$initial_x$**.



Un automa ASM, seconda versione

L'automa A:

- **Stati**: definiti dagli stati di ogni processo e dai valori delle variabili
- **Start**: Stati di partenza, valori iniziali delle variabili.
- **Azioni**: Ogni azione associata ai processi e anche alcune azioni associate a singole variabili
- **Azioni di input/output**: ai confini esterni.
- **Transizioni** : Corrispondono agli step dei processi locali e agli accessi alle variabili
- **Tasks**: uno o più per processo



Un automa ASM

Esempio

Sistema costituito da un insieme di processi P_i e da una variabile condivisa x con valori in V. Ogni processo ha una variabile input **e una variabile** output.

States di P_i :

Status $\in$ {idle,access,decide,done}, iniz. idle

Input $\in$ V $\cup$ {unknown}, iniz. unknown

Output $\in$ V $\cup$ {unknown}, iniz. unknown

Trans di P_i :

Init(v)$_i$

Effetto:

input := v

if status = idle **then** status := access

access$_i$

Precondizione:

status = access

Effetto:

if x = unknown **then** x := input

output := x

status := decide

decide(v)$_i$

Precondizione:

status = decide

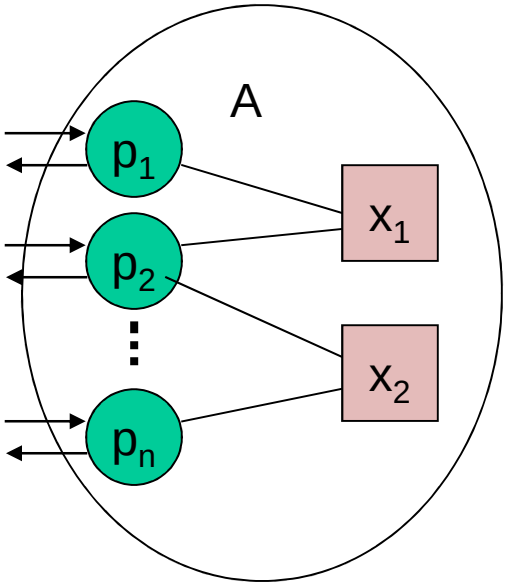
output = v

Effetto:

status := done

Un automa ASM

- **Esecuzione di *A*:**
  - Come definito dalle definizioni delle esecuzioni fair per automi I/O.
  - E' possibile definire **l'ambiente circostante come un automa** per imporre eventuali vincoli sul comportamento dell'ambiente.



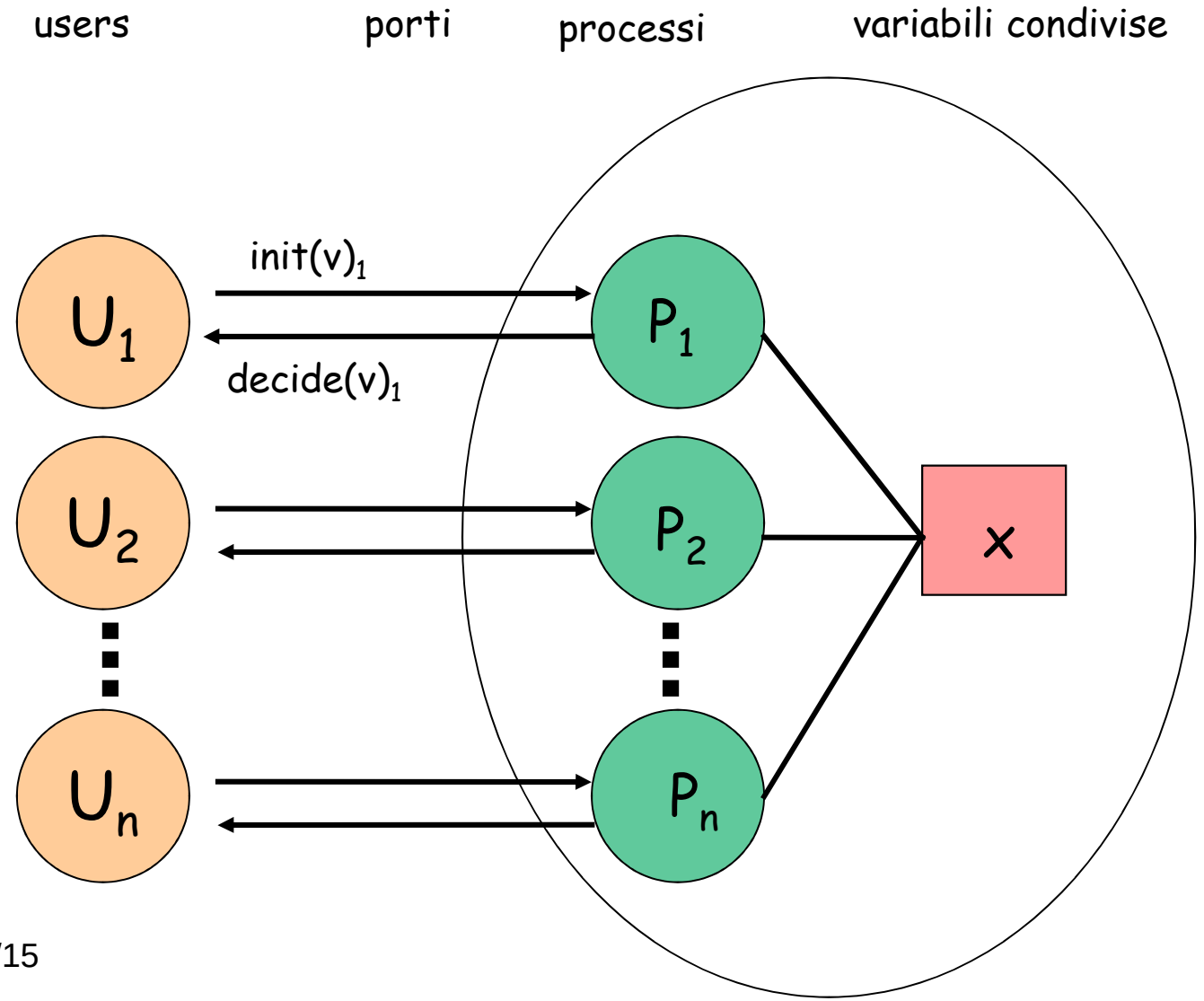
Ambiente esterno come automa

- Modellare l'ambiente esterno come un automa di I/O permette di descrivere eventuali ipotesi e vincoli sul comportamento dell'ambiente circostante.

- In alcuni casi l'ambiente circostante si può descrivere come una collezione di automi indipendenti, uno per porto.

- Nell'esempio precedente l'ambiente esterno può essere definito da un unico automa di I/O, composto di automi utente U_i, uno per ciascun processo P_i.

Ambiente esterno come automa



Ambiente esterno come automa

Signature:
Input: $decide(v)_i$, $v \in V$ interne: $dummy_i$
Output: $init(v)_i$, $v \in V$

States:
Status $\in$ {request,wait,done}, iniz. request
Decision $\in V \cup$ {unknown}, iniz. unknown
error, boolean, iniz. False

Trans:
$Init(v)_i$
Precondizione:
status = request or error = true
Effetto:
if error = false **then** status := wait

$dummy_i$
Precondizione:
error = true
Effetto:
nessuno

$decide(v)_i$

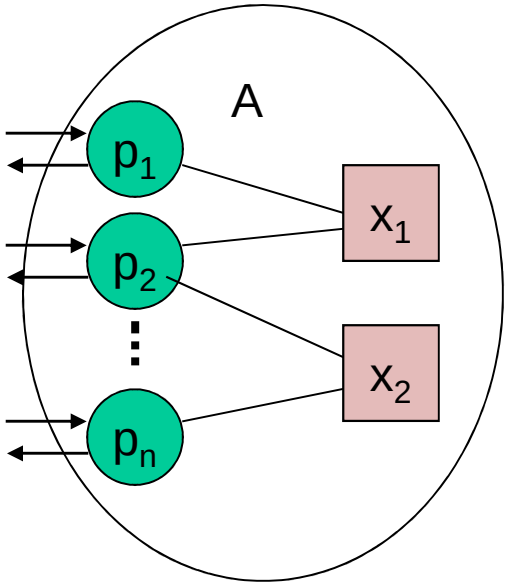
if error = false **then**
if status = wait **then**
decision := v
status := done
else error := true

Automi ASM

Si possono considerare diversi tipi di variabili:

- **read/write**: primitiva più elementare.
  - Le operazioni di read e di write sulle variabili sono separate.
- **read-modify-write**: primitiva più potente:
  - In modo atomico si legge il valore della variabile, si esegue una computazione locale e si scrive il nuovo valore .
- Compare-and-swap, fetch-and-add, queues, stacks, etc.

•Diversi risultati di computabilità e di complessità per ciascun tipo di variabile.



Variabili Read/Write

- Molto usate nei sistemi multiprocessori sono le variabili (registri) su cui si possono eseguire solo le due azioni atomiche read e write.

- Le **richieste** sono read(v) e write(v), $v \in V$

- Le **risposte** sono $v \in V$ e ack

- Possiamo descrivere tali variabili tramite una **funzione f** : f(read,v) = (v,v) e f(write(v),w) = (ack, v)

- Lo stato **access** è sostituito da due nuovi stati read e write

Variabili Read/Write

Trans:

Init(v)$_i$
  Effetto:
    input := v
    if status = idle **then** status := read

read$_i$
  Precondizione:
    status = read
Effetto :
    if x = unknown **then**
      output := input
      status := write
    else
      output := x
      status := decide

decide(v)$_i$
  Precondition:
    status = decide
    output = v
Effetto :
    status := done

write(v)$_i$
  Precondizione:
    status = write
    v = input
      Effetto :
    x := v
      status := decide

Variabili read-modify-write

Un processo P_i quando esegue una operazione **read-modify-write** su una variabile condivisa x, **atomicamente**:

1. legge x
2. esegue una qualche computazione sul valore di x, con l'effetto di modificare lo stato di P_i e determinare un nuovo valore di x
3. scrive il nuovo valore in x

Descriviamo la computazione mediante una funzione h: $V \rightarrow V$

Funzione f: f(h,v)= (v,h(v)) – restituisce il valore "vecchio" v e aggiorna con il valore "nuovo" h(v)