Distributed Algorithms

Seif Haridi

Distributed algorithms

- A distributed system is considered as a set of processors connected together by some sort of network. The system is either physical: computers connected by a network; or logical: a set of software processes connected through a message passing mechanism.

- Distributed systems differs from centralized systems in a number of essential aspects:
  - Lack of knowledge of the global state of the system. Collecting state information may be possible but may not be upto date.
  - Lack of global time frame. No total order of events.
  - Nondeterminism. For example the order of arrival of requests to a server.

Outline: Part One

- Communication protocols, techniques for analysis of distributed algorithms.
  - A model for development and verification of distributed algorithms: transition systems, proof methods for safety and liveness properties, and causality as a partial order on events in the system
  - Message transmission between two nodes; a protocol using timers; corretness proof of protocols.
  - Routing in computer networks: algorithms for computing the routing tables. Netchange algorithm. Routing algorithms, interval and prefix routing (compact routing: small amount of information are required in the network).
  - Avoiding store-and-forward packet deadlocks

Part two: Fundamental algorithms

- Algorithmic building blocks used in many distributed algorithms.
- **Wave algorithms**: general scheme to visit all nodes of a network.
- Used as component in many applications:
  - To synchronize nodes.
  - To disseminate information through a network.
  - To compute a function that depends on information stored in all nodes of the network
- Time complexity of distributed algorithms

Part two: Fundamental algorithms

- **Election**: selection of a single node in a network (to perform certain control functions).
  - Ring of N processors: message complexity is $\Theta(N \log N)$
  - General Network: election is obtained from a wave algorithm and a traversal algorithm.
- **Termination detection**: the recognition that a distributed computation has terminated.
- **Anonymous Networks**: computation power of a system where processors cannot be distinguished (identified). The study of probabilistic algorithms.
- **Snapshots**: how to compute a global picture of the system's state.

Part two: Fundamental algorithms

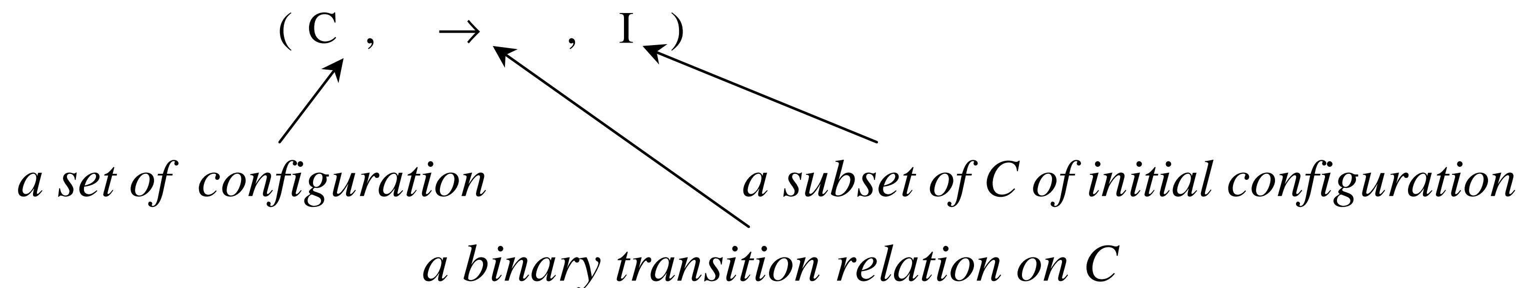
- Synchronous systems: availability of global time.
  - Asynchronous systems can simulate synchronous systems by trivial algorithms.
  - Computational complexity: the better the synchronism, the lower the complexity of “many problems”.

Fault Tolerance

- Deterministic asynchronous algorithms “in general” cannot cope with simple failures (single process failure).
- Deterministic synchronous algorithms tolerate nontrivial failures. Implementation of synchronism in unreliable networks.
- Self stabilization algorithms: regardless of the initial configuration the algorithm converges eventually on what is its intended behavior.
- Examples of self-stabilizing algorithms:
  - Computation of depth-first search trees.
  - Computation of routing tables.
  - Data transmission.

The model

- We model only systems which communicate by message passing.
- The natural model is that of a transition system.
- A transition system is a triple:



$\gamma \rightarrow \delta$: a move from configuration γ to δ

An execution of S is a maximal sequence:

$E = (\gamma_0, \gamma_1, \gamma_2, \cdots)$

$\gamma_0 \in I$, for all $i \geq 0$, $\gamma_i \rightarrow \gamma_{i+1}$

Transition Systems

<u>Terminal configuration γ</u>

is a configuration for which there is no δ such that

$\gamma \rightarrow \delta$

<u>Configuration δ reachable from γ</u>: $\gamma \Rightarrow \delta$

There exist a sequence: $\gamma = \gamma_0, \gamma_1, \gamma_2, \cdots, \gamma_k = \delta$,

$\gamma_i \rightarrow \gamma_{i+1}$, for all $0 \leq i < k$

<u>Configuration δ is reachable</u>

if it is reachable from an initial configuration

Systems with asynchronous message passing

- A system consists of a set of processes, and a communication subsystem.
- Each process is modeled as a transition systems, where a process-configuration is called a **state**, and a process-transition is called an **event.**
- Each process performs three types of events:
  - internal event
  - send event
  - receive event

Local algorithm

M : a set of possible messages

The local algorithm of a process:

---

is $\langle Z, I, \rhd^i, \rhd^s, \rhd^r \rangle$

Z: the set of all states

I: subset of Z, the set of initial state

$\rhd^i$: a subset of $Z \times Z$

$\rhd^s$: a subset of $Z \times M \times Z$

$\rhd^r$: a subset of $Z \times M \times Z$

$c \rhd d \equiv c \rhd^i d \vee \exists m \in M(\langle c, m, d \rangle \in \rhd^s \cup \rhd^r)$

An event is either an internal event or

a communication event

Distributed algorithms

- A distributed algorithm (DA) is a collection of local algorithms, one for each process in the system.
- The transition system of a distributed algorithm DA is
  - A set of configuration, each configuration consists of the state of each process, and a network component representing the messages in transit.
  - A transition is an event of one of the processes. If the event is a communication event it has conditions on the communication subsystem.
  - The communication subsystem is a multiset of messages.

Distributed algorithms

Transition system S induced under asynchronous communication by a distributed algorithm for processes

$p_1,\ldots,p_N$ is $\langle C, \rightarrow, I \rangle$

$\langle c_{p_1}, \cdots, c_{p_N}, M \rangle \in C$

$c_{p_i} \in Z_{p_i}$ is a state of process p_i

$\langle c_{p_1}, \cdots, c_{p_N}, \varnothing \rangle \in I, \forall p_i (1 \leq i \leq N) c_{p_i} \in I_{p_i}$

A move is made if one of the process components makes a move.

Each message in M has a unique destination

Transition relation

$$\frac{\langle c_{p_i}, M_1\rangle \rightarrow_{p_i} \langle d_{p_i}, M_2\rangle \ (\forall 1 \le i \le N)}{\langle c_{p_1}, \cdots, c_{p_i}, \cdots, c_{p_N}, M_1\rangle \rightarrow \langle c_{p_1}, \cdots, d_{p_i}, \cdots, c_{p_N}, M_2\rangle}$$

$\langle c_{p_i}, M_1\rangle \rightarrow_{p_i} \langle d_{p_i}, M_2\rangle$ is defined in terms of the local execution of p_i

$$\frac{c_{p_i} \rhd^{i} d_{p_i}}{\langle c_{p_i}, M_1\rangle \rightarrow_{p_i} \langle d_{p_i}, M_1\rangle} \quad \text{internal event}$$

$$\frac{\langle c_{p_i}, m, d_{p_i}\rangle \in \rhd^{s}}{\langle c_{p_i}, M_1\rangle \rightarrow_{p_i} \langle d_{p_i}, M_1 \cup m\rangle} \quad \text{send m}$$

$$\frac{\langle c_{p_i}, m, d_{p_i}\rangle \in \rhd^{r} \wedge m \in M_1}{\langle c_{p_i}, M_1\rangle \rightarrow_{p_i} \langle d_{p_i}, M_1 - m\rangle} \quad \text{receive m}$$

Synchronous message passing

Similar to the asynchronous system, but:

- Configuration consists only of a tuple of process states
- The is a "synchronizing transition"

$$\frac{\exists m \in M : \langle c_{p_i}, m, d_{p_i} \rangle \in \triangleright^{s}_{p_i} \wedge \langle c_{p_j}, m, d_{p_j} \rangle \in \triangleright^{r}_{p_j}}{\langle \cdots, c_{p_i}, \cdots, c_{p_j}, \cdots \rangle \rightarrow \langle \cdots, d_{p_i}, \cdots, d_{p_j}, \cdots \rangle}$$

Synchronous message passing is more restrictive that asynchronous: (possible executions of SMP is a subset of possible executions of AMP).

Fair executions

Weak fair execution:

An execution is weakly fair if no event is applicable in infinitely many consecutive configurations without ocurring in the execution.

Strong fair execution:

An execution is weakly fair if no event is applicable in infinitely many configurations without ocurring in the execution.

Fairness in not generally assumed.

Proving properties of transition systems

Safety and liveness properties:

theses are predicates on a configuration.

Safety requirements:

A safety property is a property that must hold for every execution on each reachable configuration in the execution.

Liveness requirements:

A liveness property is a property that must hold for every execution on some reachable configurations in the execution.

Invariants

$S = \langle C, \rightarrow, I \rangle$

$\{P\} \rightarrow \{Q\}$ means for all $\gamma \rightarrow \delta$, if $P(\gamma)$ then $Q(\delta)$

Definition: An assertion P is an **invariant** of S if

- for all $\gamma \in I$, $P(\gamma)$,
- $\{P\} \rightarrow \{P\}$

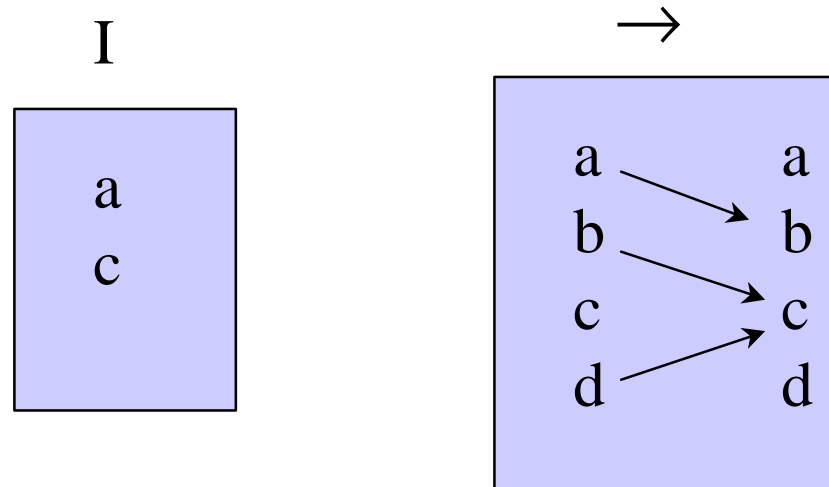
Theorem:

If P is an invariant of S, then P holds for each configuration of each execution of S.

The converse is not true:

if P holds for each configuration of each execution of S then P is an invariant of S

Invariants



Invariants are static properties, whereas executions are dynamic. The only possible execution is (a,b,c).
If P holds for a,b, and c it is still not an invariant.
Hence not every safety property can be proved by the previous theorem.

Liveness properties

"P is eventually true in each execution"

Definition: A partial order $\langle W, > \rangle$ is well founded

if there is no infinite decreasing sequence:

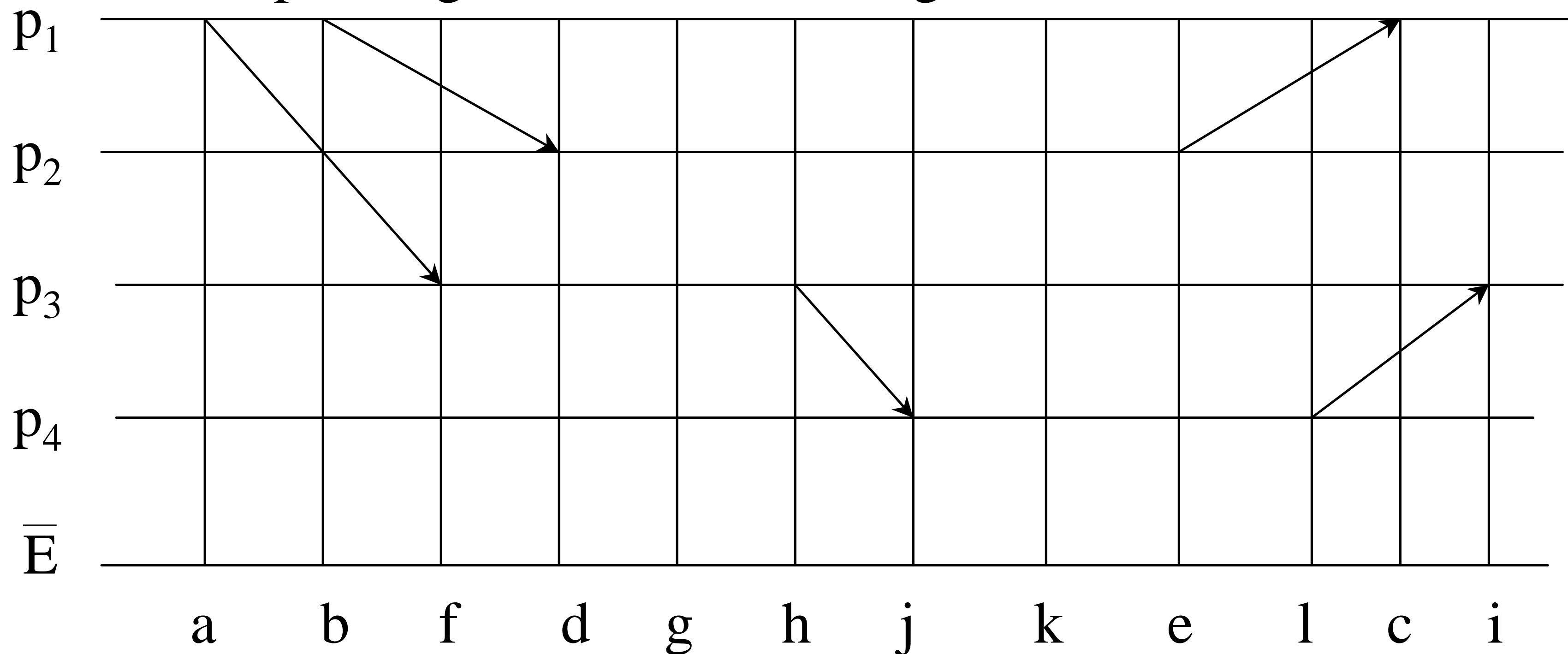
$w_1 > w_2 > w_3 > \cdots$

Show that there is a function f from C to a well-founded set W such that on each transition the value of f decreases or P becomes true.

Causal order of events

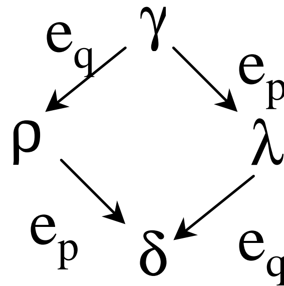
For an execution $E = \langle \gamma_0, \gamma_1, \cdots \rangle$ there is a corresponding sequence of events $\overline{E} = \langle e_0, e_1, \cdots \rangle$; where e_i is the event that causes the transition $\gamma_i \rightarrow \gamma_{i+1}$.

Time-space diagram: an arrow is drawn between a send and a corresponding receive of a message.



Independence and dependence of events

Theorem(asynchronous message passing):
Let γ be a configuration, e_p and e_q be events by two different processes p and q, both applicable in γ.
Then e_p is applicable in $e_q(\gamma)$, e_q is applicable in $e_p(\gamma)$, and $e_p(e_q(\gamma)) = e_q(e_p(\gamma))$.



- Remember: messages are distined to only one process.
- e_p and e_q cannot be a corresponding send-receive events

Causal order

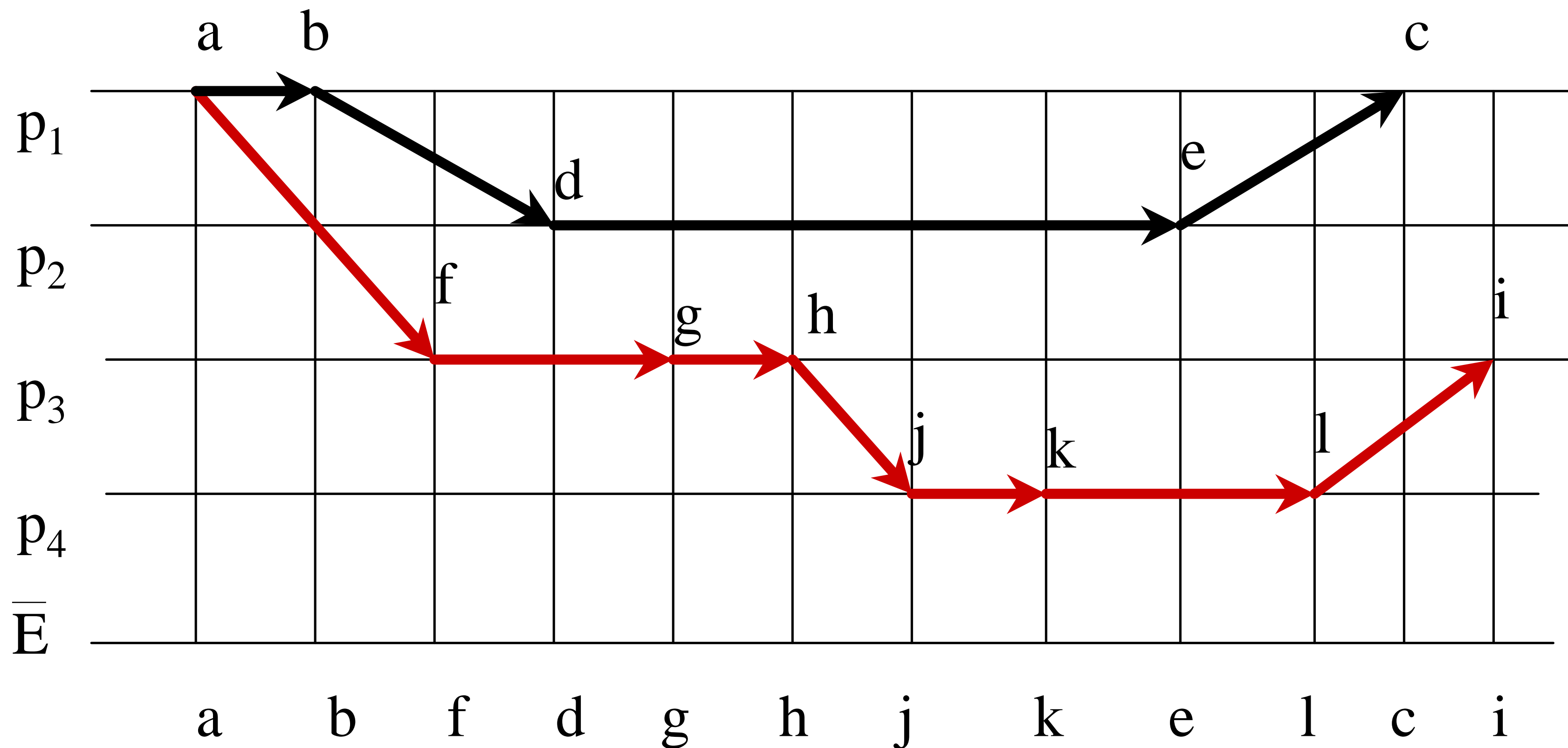
Let E be an execution. The relation $\prec$ called the causal order on the events of E is the smallest relation satisfying:
(1) If e and f are different events of the same process and e is before f, then $e \prec f$.
(2) If s is a send event and r the corresponding receive event then $s \prec r$.
(3) $\prec$ is transitive.

$a \preceq b$ is $a \prec b \vee a = b$ ($\preceq$ is a partial order).

Events a and b that neither $a \preceq b$ nor $b \preceq a$ are called concurrent ($a||b$).

Causality chain

The causality chain between events a and l is a,f,g,h,j,k,l,i

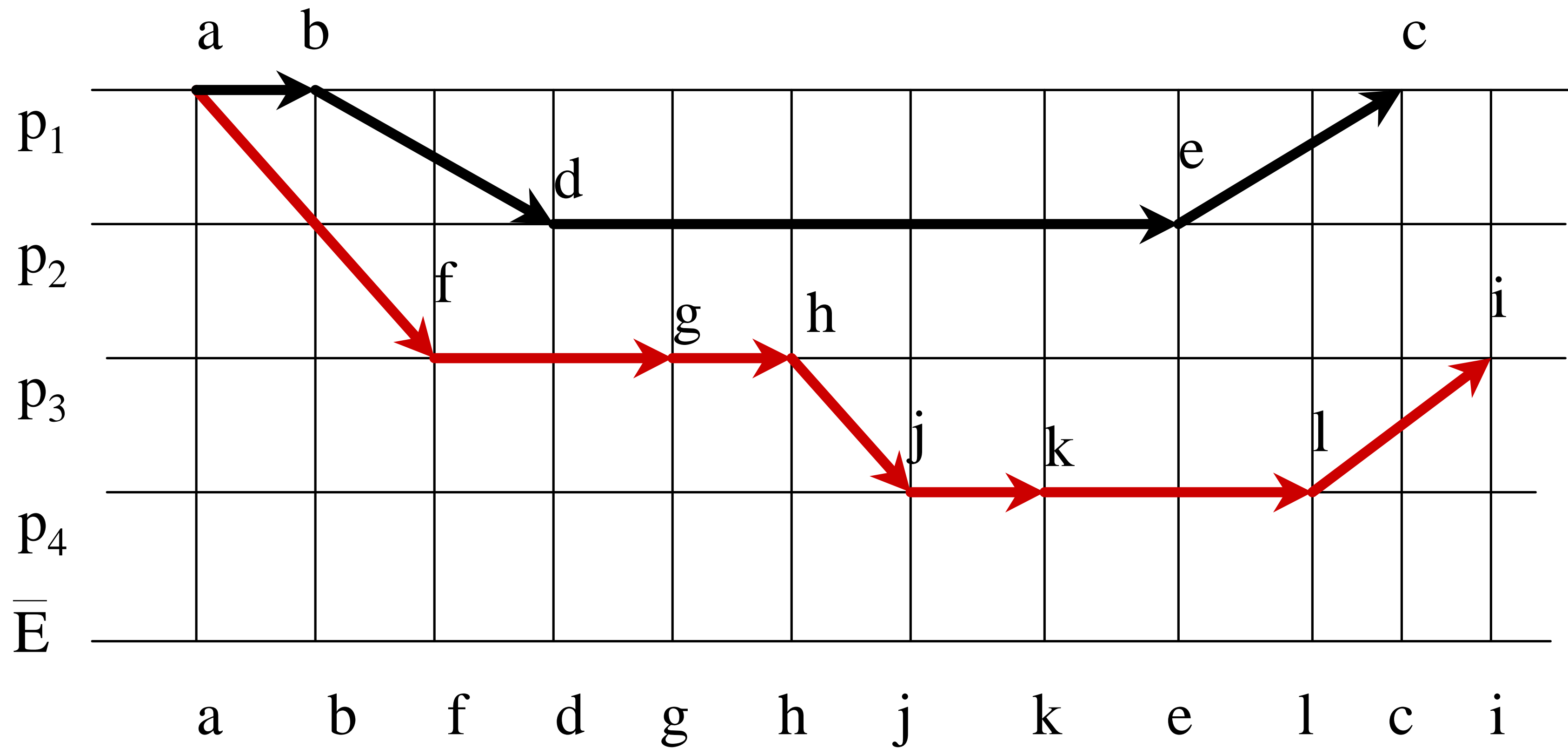


Equivalence of executions

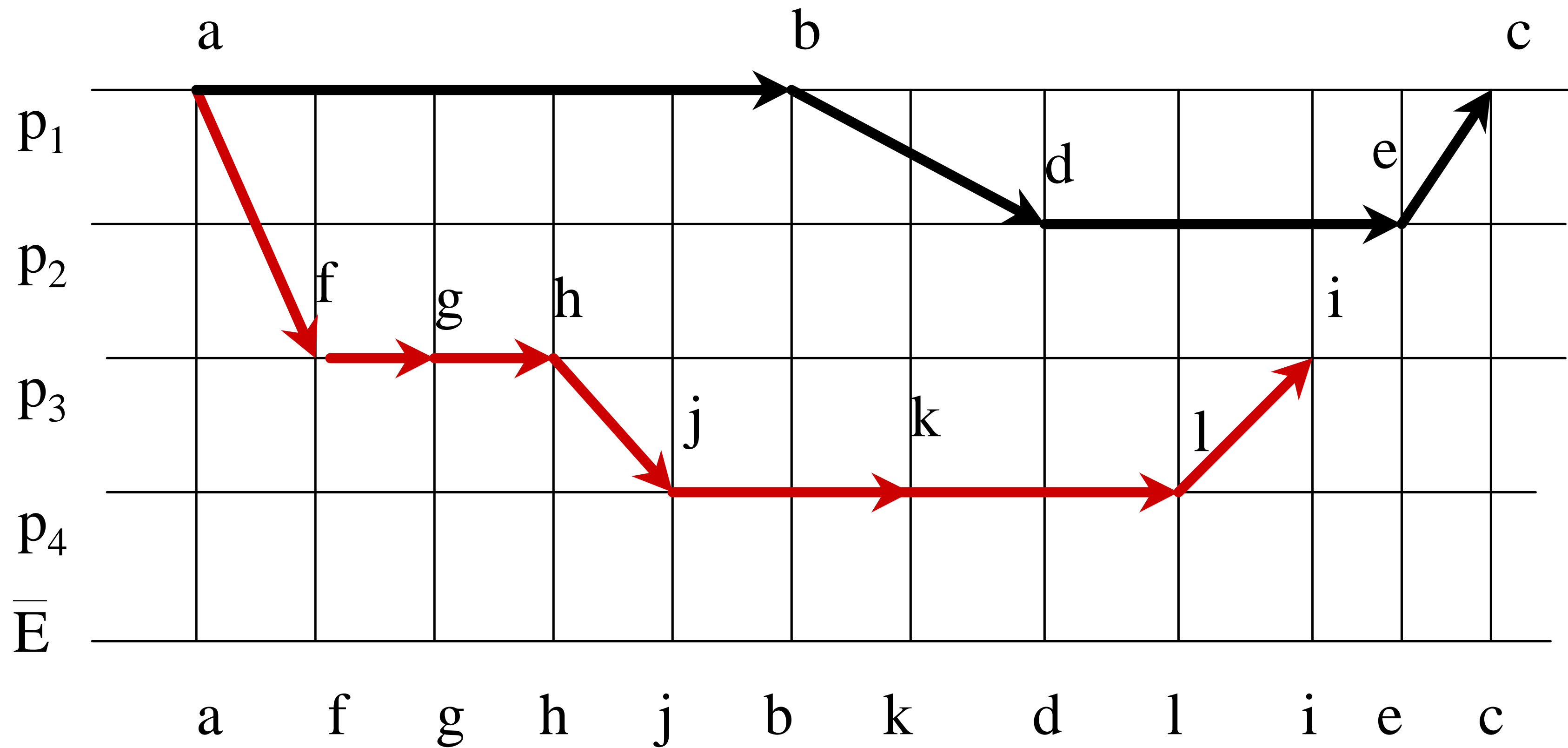
$E = \langle \gamma_0, \gamma_1, \cdots \rangle$ is an execution,
$\overline{E} = \langle e_0, e_1, \cdots \rangle$ is the associated sequence of events,
$f = \langle f_0, f_1, \cdots \rangle$ is a permutation of $\overline{E}$ that is consistent with the causal order of the events of E,
i.e. if $f_i \preceq f_j$ then $i \leq j$

Theorem:
f defines a unique execution F, starting from the initial configuration of E.
F has as many events as E.
The last configuration of F is the same as the last configuration of E.

Example 1



Example 1



Equivalent executions

Equivalent executions form an equivalence class $\approx$.
A computation of a distributed algorithm is an equivalence class of executions of the algorithm.

Logical clocks

- In distributed systems clocks can be defined that express causality.

Definition
let Θ be a function from the set of events to an ordered set (e.g. natural number).
Θ is a logical clock if $a \prec b$ implies $\Theta(a) < \Theta(b)$.

Lamport's clock function:
It assigns to an event a the length k of the longest sequence $\langle e_1, \cdots, e_k \rangle$ of events:

$$e_1 \prec e_2 \prec \cdots \prec e_k = a$$

The algorithm

- The basic idea
  - If a is an internal or send event, a’ is the previous event in the same process, then $\Theta_L(a) = \Theta_L(a') + 1$.
  - If a is a receive event, a’ is the previous event in the same process, and b is the send event corresponding to a, then $\Theta_L(a) = \max(\Theta_L(a'), \Theta_L(a')) + 1$.
  - If a is the initial event, $\Theta_L(a')=0$.
- When logical clocks are used.