Distributed and Parallel Computing

Lecture 10

Alan P. Sexton

University of Birmingham

Spring 2019

Distributed Computing

By *Distributed Computing* we mean:

- Computing on multiple *nodes* (i.e. *processes*)
- Each node has a unique *identifier*
- Nodes are connected via a *network* of *channels* (i.e. *edges*)
- Nodes do not share memory or a global clock
- There can be 0 or 1 channels between any two nodes
- The network is assumed to be *strongly connected*: there is a (possibly multi-hop) path between every two nodes
- The network may or may not be *complete*: i.e. have a channel from every node to every other node
  - Note difference from *strongly connected*
- Channels can be *directed* (messages travel one way only) or *undirected* (messages travel in either direction)
- Communication is via passing *messages* over *channels*
- Channels are not necessarily *First-In-First-Out (FIFO)*: messages sent on the same channel may overtake each other

More Definitions

- A process knows:
  - its own state
  - the messages it sends and receives
  - its direct neighbours in the network
- Underlying communication protocol is reliable
  - No messages corrupted, duplicated or lost
- Communication is *asynchronous*: there is an arbitrary, non-deterministic but finite delay between sending and receiving a message
- Parameters:
  - N: the number of nodes
  - E: the number of edges
  - D: the diameter of the network: the longest "shortest path"
    - the number of edges in the longest path chosen from the set of shortest paths between every pair of nodes in the graph

Failures in a Distributed System

Failure Rate (FR): The number of failures per unit time
Mean Time Before Failure (MTBF): $\frac{1}{\mathrm{FR}}$
Example: Distributed system with 1000 nodes, all of which are critical. MTBF of each node of 10,000 hours $\approx$ 60 weeks. What is the MTBF of the system?

Failures in a Distributed System

Failure Rate (FR): The number of failures per unit time
Mean Time Before Failure (MTBF): $\frac{1}{\mathrm{FR}}$
Example: Distributed system with 1000 nodes, all of which are critical. MTBF of each node of 10,000 hours $\approx$ 60 weeks. What is the MTBF of the system?

- Given a system for which all sub-systems are critical:

$$\mathrm{FR}_{\mathrm{sys}} = \sum_{s \in \mathrm{sys}} \mathrm{FR}_s$$

Failures in a Distributed System

Failure Rate (FR): The number of failures per unit time
Mean Time Before Failure (MTBF): $\frac{1}{\mathrm{FR}}$
Example: Distributed system with 1000 nodes, all of which are critical. MTBF of each node of 10,000 hours $\approx$ 60 weeks. What is the MTBF of the system?

- Given a system for which all sub-systems are critical:

$$\mathrm{FR}_{\mathrm{sys}} = \sum_{s \in \mathrm{sys}} \mathrm{FR}_s$$

Hence:

$$\mathrm{FR}_{\mathrm{node}} = \frac{1}{10000} \text{ failures per hour}$$

$$\mathrm{FR}_{\mathrm{sys}} = \sum_{\mathrm{node} \in \mathrm{sys}} \frac{1}{10000} \text{ failures per hour}$$

$$= 1000/10000 = 0.1 \text{ failure per hour}$$

$$\mathrm{MTBF}_{\mathrm{sys}} = 10 \text{ hours}$$

Distributed Computing Issues

- Parallel GPU computing is all about efficiency
- Distributed computing is all about uncertainty: Without explicit communication and computation, a node does not know
  - What time the other nodes think it is
  - What state the other nodes are in
  - Whether the other nodes have finished their computations
  - Whether it is safe to access a shared resource
  - Whether any of the nodes have failed (crashed)
  - Whether all the nodes are mutually waiting on each other (deadlock)
  - Whether it is safe to delete/destroy a shared resource when this node no longer needs it

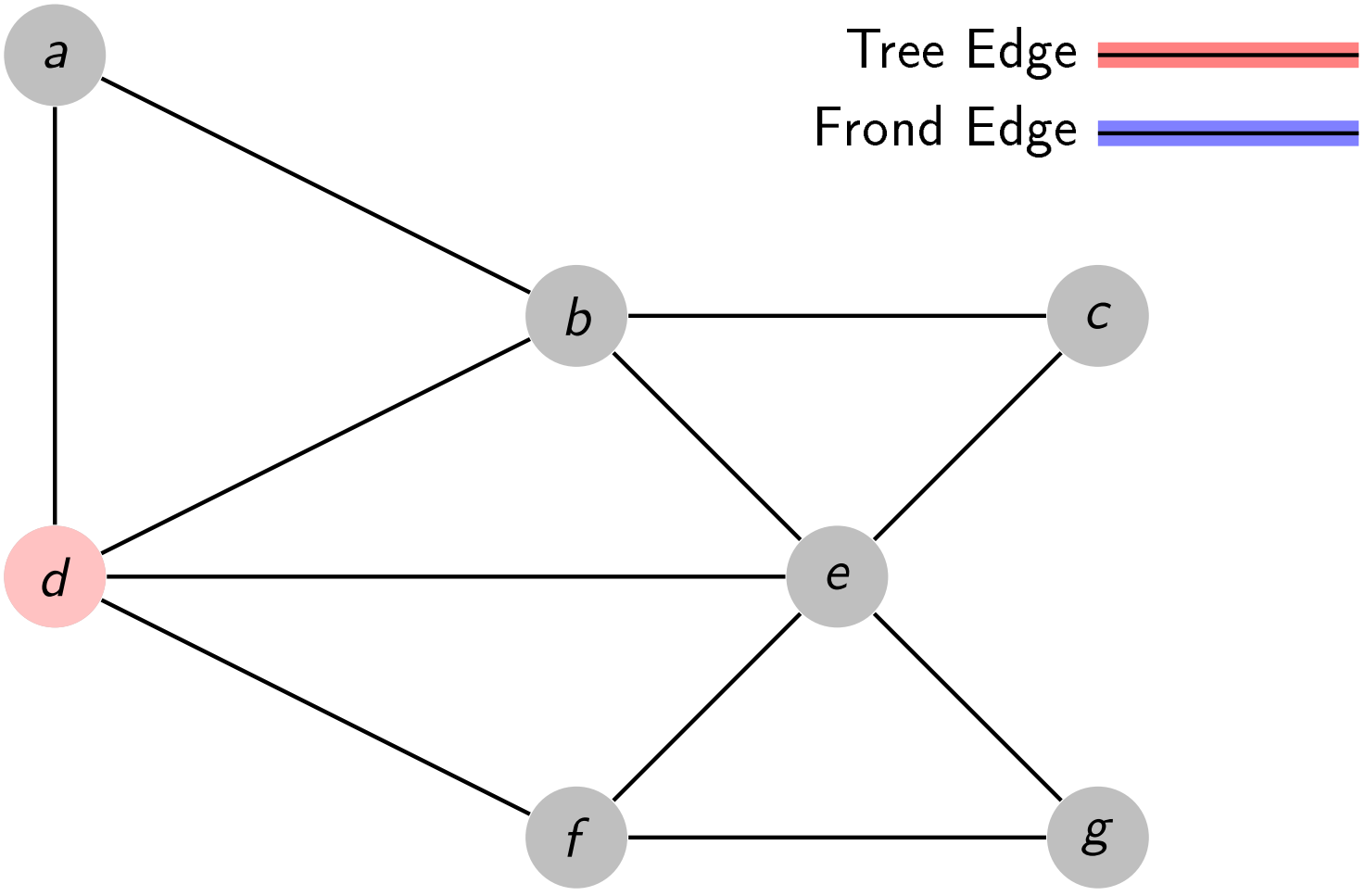
Spanning Tree of a Network

A *Spanning Tree*

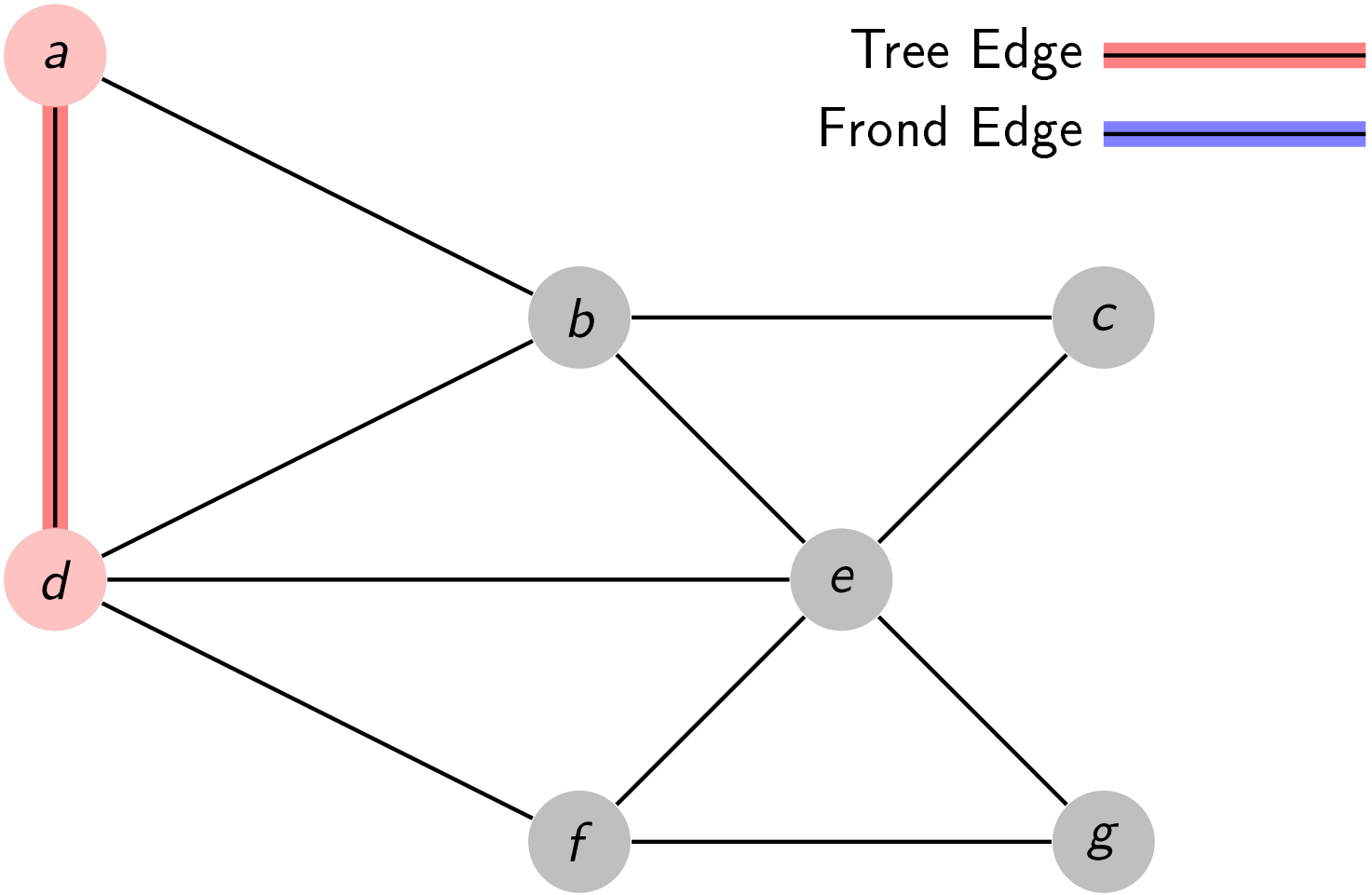
- Contains all the nodes of a network
- Its edges are a subset of the network edges
- Has no cycles
- Is undirected

Tree Edges: edges in the spanning tree
Frond Edges: edges in the network but not in the spanning tree
Sink Tree: a tree made by making all the edges of a spanning tree directed from *child* nodes to *parent* nodes terminating at the *root* node

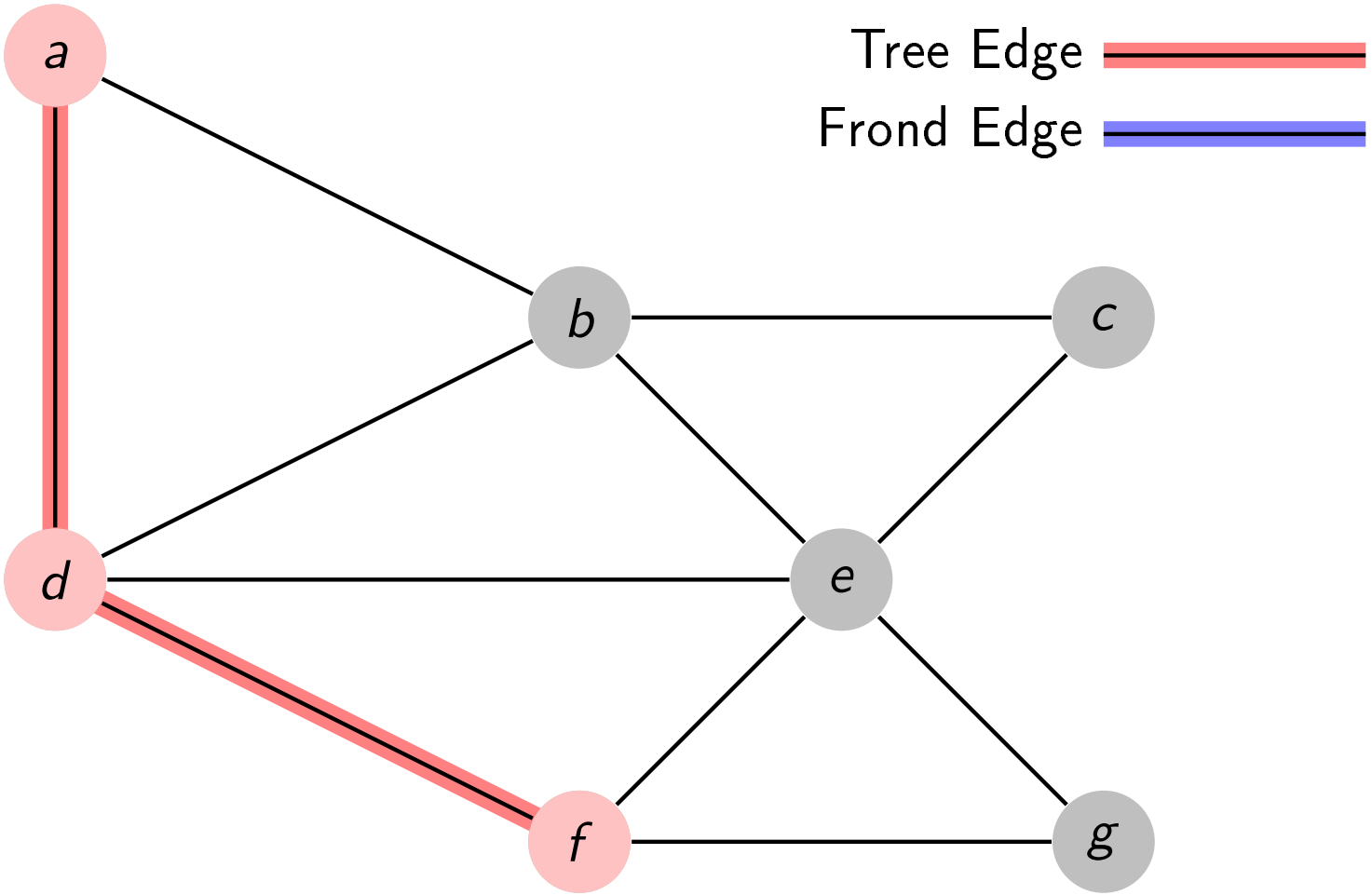
Spanning Tree Example



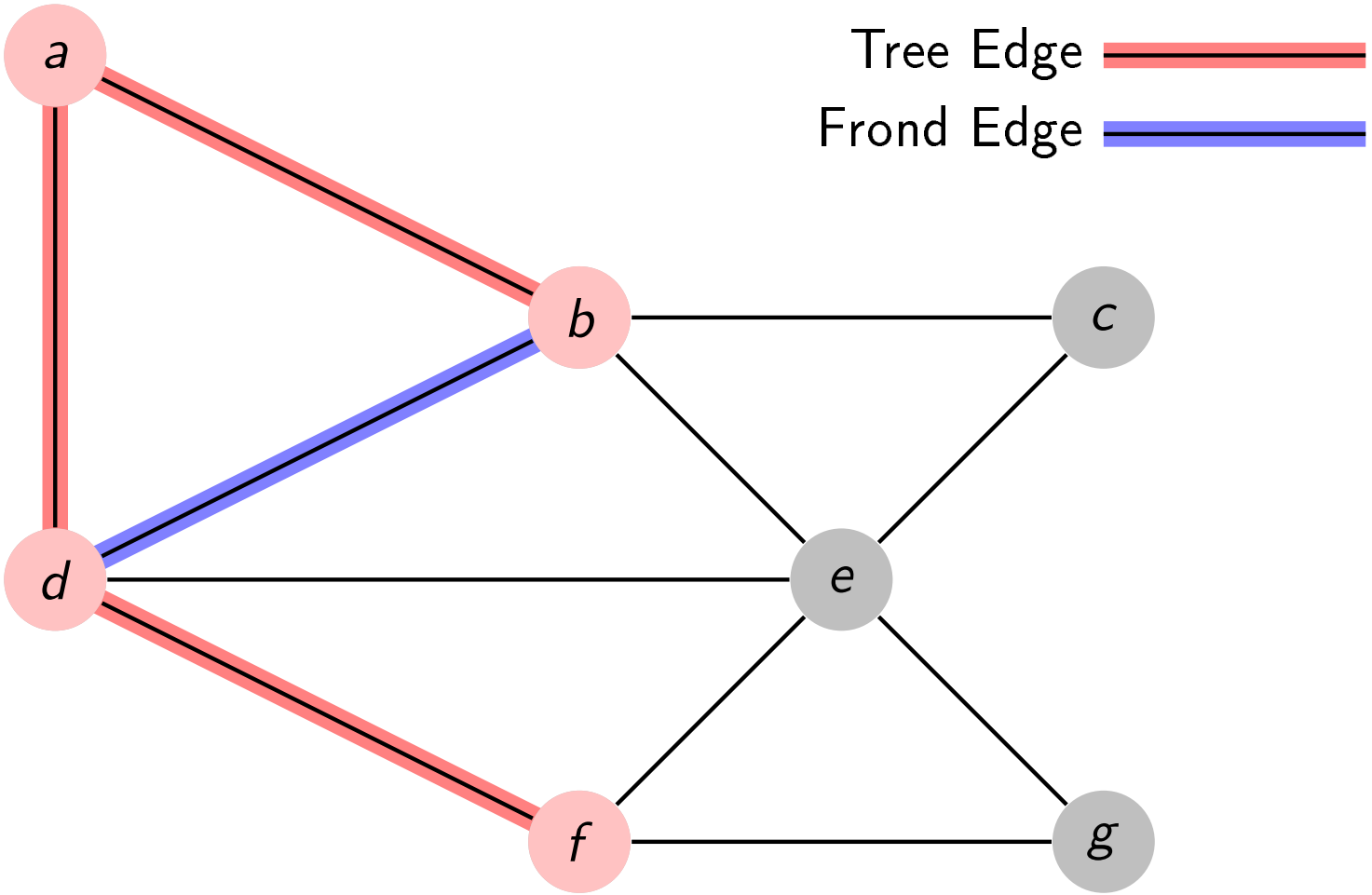
Spanning Tree Example



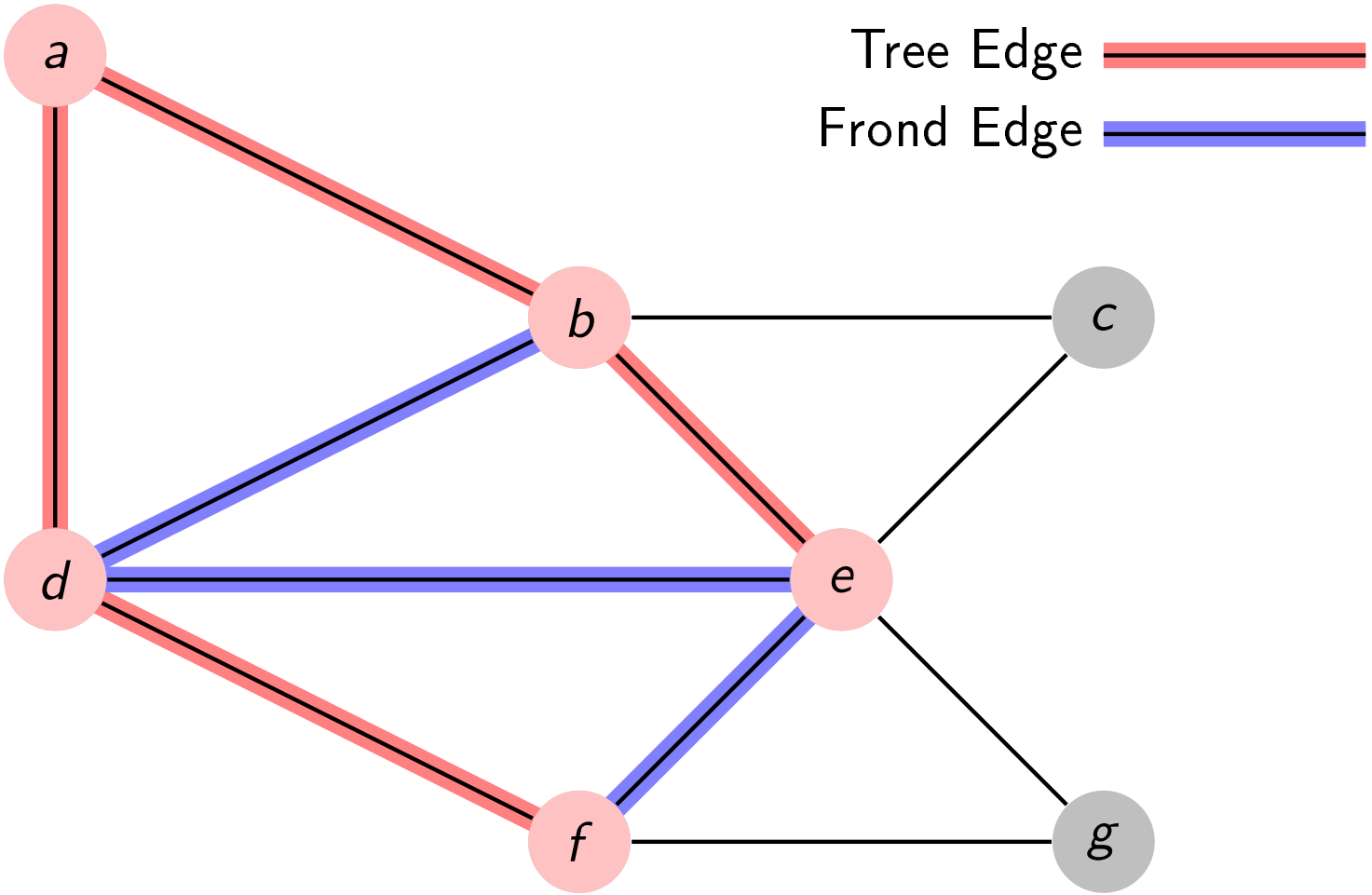
Spanning Tree Example



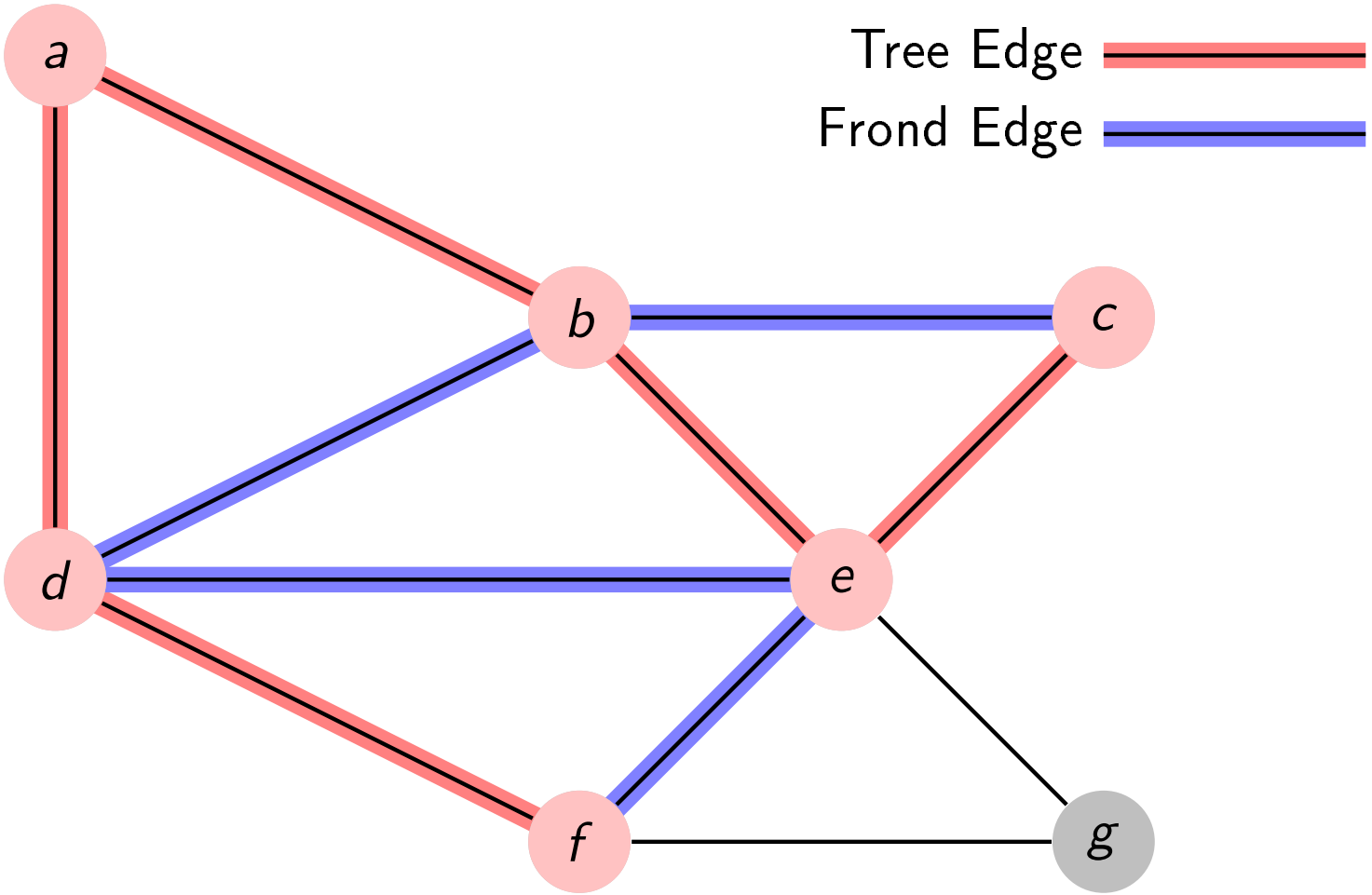
Spanning Tree Example



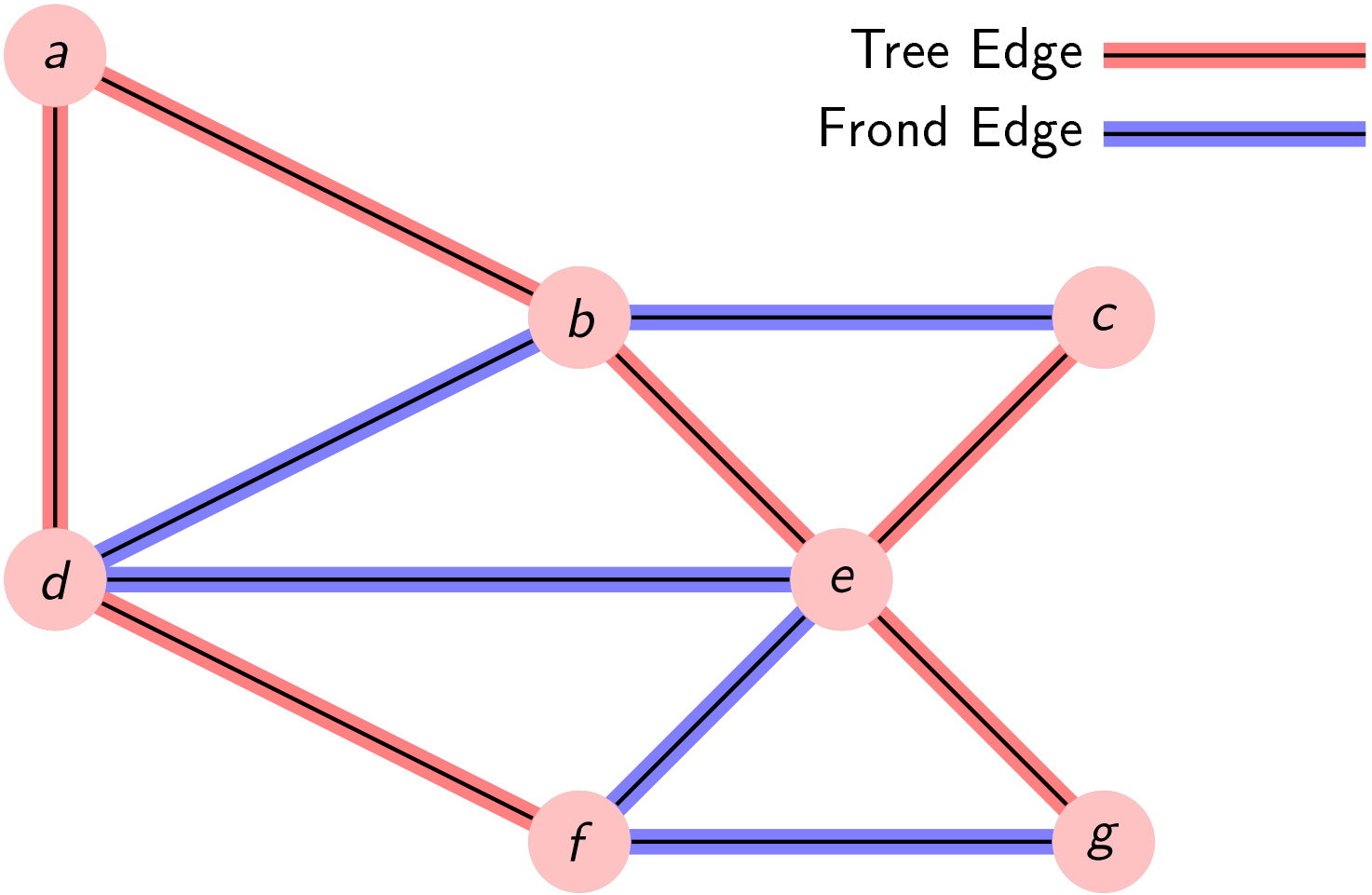
Spanning Tree Example



Spanning Tree Example



Spanning Tree Example



Why are Spanning Trees Important?

- Nodes only see immediate neighbours, not the whole topology
- How can a node distribute a message to the whole network?

Why are Spanning Trees Important?

- Nodes only see immediate neighbours, not the whole topology
- How can a node distribute a message to the whole network?
  - Send to all neighbours recursively

Why are Spanning Trees Important?

- Nodes only see immediate neighbours, not the whole topology
- How can a node distribute a message to the whole network?
  - Send to all neighbours recursively
    - Never terminates

Why are Spanning Trees Important?

- Nodes only see immediate neighbours, not the whole topology
- How can a node distribute a message to the whole network?
  - Send to all neighbours recursively
    - Never terminates
  - Send to all neighbours recursively, stop if already received

Why are Spanning Trees Important?

- Nodes only see immediate neighbours, not the whole topology
- How can a node distribute a message to the whole network?
  - Send to all neighbours recursively
    - Never terminates
  - Send to all neighbours recursively, stop if already received
    - Wasteful: every node receives message multiple times

Why are Spanning Trees Important?

- Nodes only see immediate neighbours, not the whole topology
- How can a node distribute a message to the whole network?
  - Send to all neighbours recursively
    - Never terminates
  - Send to all neighbours recursively, stop if already received
    - Wasteful: every node receives message multiple times
  - Send to spanning tree root, which sends to all tree children recursively
- Actual algorithms are more sophisticated, but depend on spanning trees for efficiency and guarantees of correctness

Transition Systems

The behaviour of a *distributed algorithm* is given by a *transition system*:

- A set of *configurations* ($\lambda, \delta \in \mathcal{C}$): each configuration is a *global* state of a distributed algorithm. $\mathcal{C}$ is the set of all possible configurations of an algorithm.
- A binary *transition* relation on $\mathcal{C}$. Each transition $\lambda \rightarrow \delta$ is a step that changes the global state from one configuration to another.
- A set $\mathcal{I}$ of *initial* configurations representing the possible starting configurations for the system.

Transition System Definitions

A number of definitions apply to a transition system:

- A configuration is called *terminal* if there are no transitions out of that configuration.
- An *execution* of the distributed algorithm is a sequence of configurations beginning with an initial configuration, connected via transitions, and which is either infinite or ends in a terminal configuration.
- A configuration is *reachable* if there is a possible execution that includes that configuration.

States and Events

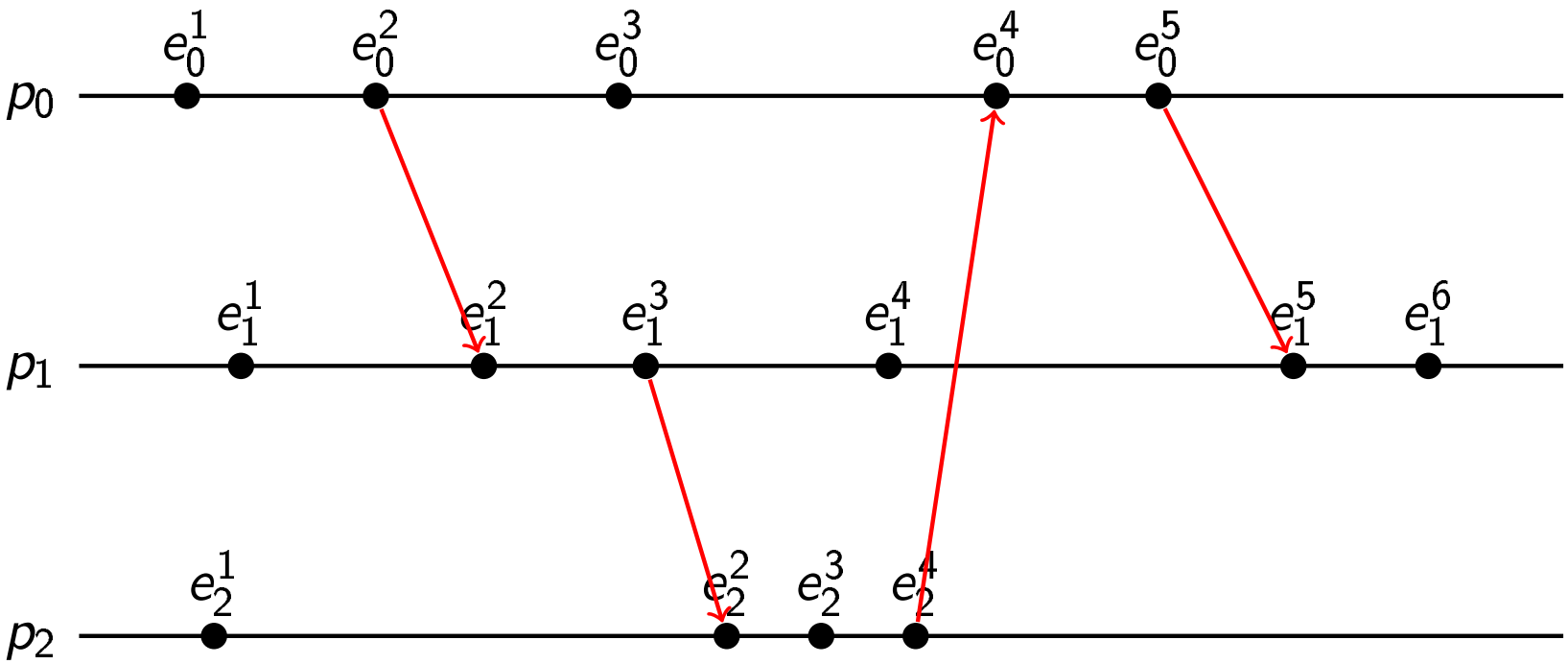
A configuration is composed of the set of local states of each node and the messages in transit between the nodes.
Transitions are connected with events:

- *internal*: some event internal to an individual process such as reading or writing a variable. Internal events affect only the local state of the process involved
- *send*: A message is sent from one process to another (causing a future *receive* event at the other process)
- *receive*: A message is received from another process

We will restrict ourselves to asynchronous systems: events never occur exactly simultaneously.

- A process is an *initiator* if its first event is an *internal* or *send* event.
- A *centralized* algorithm has precisely one initiator
- A *decentralized* algorithm can have multiple initiators

Diagram of an Execution



Assertions

Assertions are predicates on configurations of a distributed algorithm

- *Safety Property*: an assertion that is required to be true in every *reachable* configuration of every execution
  - Typically used to assert that nothing bad will happen
  - If the assertion fails, there will a *finite witness*: if something bad happens on an infinite run, it already happens on a finite prefix.
  - e.g. all results produced are correct
- *Liveness Property*: an assertion that is required to be true in some configuration in every execution after some point
  - Typically used to assert that something good will eventually happen
  - If the assertion fails, there will **NOT** be a *finite witness*: no matter what happens along a finite run, something good could still happen later.
  - e.g. a result *will* be produced

Partial vs Total Order

A *Total Order on A* defines a relationship $\leq$ on a set A such that for all elements a, b, c in A:

- $a \leq a$ (reflexivity)
- $a \leq b$ and $b \leq a \rightarrow a = b$ (antisymmetry)
- $a \leq b$ and $b \leq c \rightarrow a \leq c$ (transitivity)
- $a \leq b$ or $b \leq a$ (totality) note: totality implies reflexivity

For example:

- The usual order on integers: $1 \leq 2$
- Lexicographic (dictionary) order on strings: `"Jones"` $\leq$ `"Smith"`

A *Partial Order* has the first 3 properties above but does not have totality. For example:

- Prefix order on strings: `"abc"` $\leq$ `"abcd"`, `"abc"` $\not\leq$ `"xyz"`, `"xyz"` $\not\leq$ `"abc"`,
- Subset order on sets: $\{1,3\} \subseteq \{1,2,3\}, \{1,3\} \not\subseteq \{1,2\}, \{1,2\} \not\subseteq \{1,3\}$

Order

In each configuration, the possible events that can occur on different processes can occur in any order.

Causal Order ($\prec$) on events: $a \prec b$ if and only if a must occur before b in any execution:

- If a and b are events in the same process and a occurs before b, then $a \prec b$
- If a is a send and b is the corresponding receive, then $a \prec b$
- If $a \prec b$ and $b \prec c$ then $a \prec c$

Order

In each configuration, the possible events that can occur on different processes can occur in any order.

Causal Order ($\prec$) on events: $a \prec b$ if and only if a must occur before b in any execution:

- If a and b are events in the same process and a occurs before b, then $a \prec b$
- If a is a send and b is the corresponding receive, then $a \prec b$
- If $a \prec b$ and $b \prec c$ then $a \prec c$

Concurrent events: distinct events in an execution that are not causally related.

- With the above definition, concurrent events may not have occurred at the same physical time. However, a distributed system cannot tell the difference, so we can treat them as if they have.

Computations

If a sequence of configurations starting with λ and ending with δ in an execution is triggered by concurrent events, then changing the order of those events will change the intermediate configurations, but the new sequence will still start with λ and end with δ.
For example, given nodes a, b, c, d:

a sends to b	a sends to b
c sends to d	b receives from a
b receives from a	c sends to d
d receives from c	d receives from c

Executions are too specific: They distinguish between insignificant differences
We will be more concerned with sets of executions which differ only by reorderings of concurrent events
Computation: a set of executions equivalent up to permutations of sequences of concurrent events.

Logical Clocks

It is difficult and expensive to simulate a shared global real time clock in a distributed system. A logical clock is easy and efficient to maintain and is sufficient for many purposes that a global real time clock is used for.
A *logical clock* ($C(a)$) maps events to a partially ordered set (usually integers) such that

$$a \prec b \Rightarrow C(a) < C(b)$$

Implementation requires each process maintains data structures to support:

- A *local logical clock*: measures this process's own progress
- A *global logical clock*: this process's view of the logical global time, used to update the local logical clock to a globally consistent logical time.

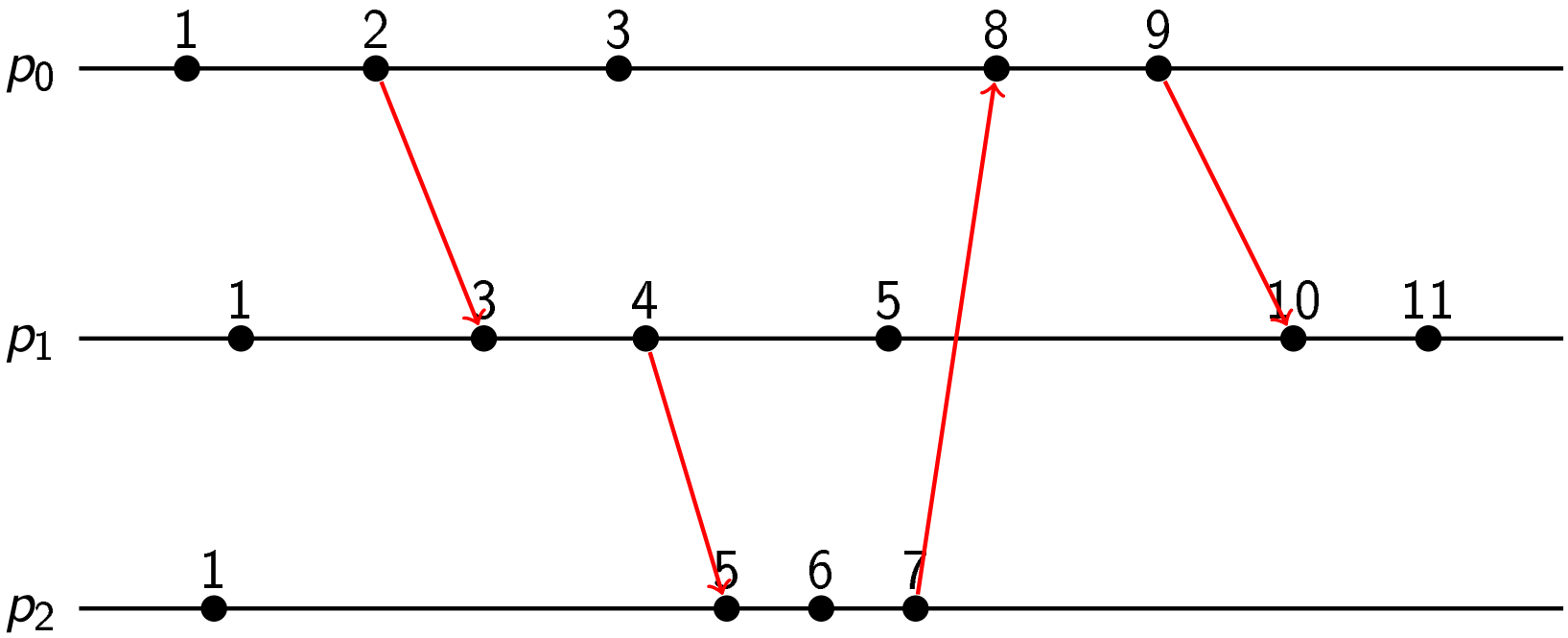
Lamport's Clock (1978)

Lamport's Clock, $\mathrm{LC}(a)$, is a distributed algorithm used to assign numbers to all events so that the natural order of the numbers respects the causal order of those events. It combines both the local and global logical clocks into a single integer variable.

Each process executes the following:

- Let a be an event and let k be the clock value of the previous event ($k = 0$ if there was no previous event)
  - If a is an internal or send event, $\mathrm{LC}(a) = k + 1$
  - If a is a receive event, and b the send event corresponding to a, then $\mathrm{LC}(a) = \max(k, \mathrm{LC}(b)) + 1$
    (The message carries the value $\mathrm{LC}(b)$ with it)

Example of Lamport's Clock



Properties of Lamport's Clock

Thus $\mathrm{LC}(a)$ assigns to a the length of a longest causality chain (there may be more than one) in the computation to the occurrence of a.
If $a \prec b$ then there is a causality chain from a to b and therefore $\mathrm{LC}(a) < \mathrm{LC}(b)$, so Lamport's clock *IS* a logical clock.

- Events with the same logical time are not causally related.
  - p_0^3 and p_1^2 are not causally related but both have the same LC time of 3.
- If needed we can impose a total ordering by breaking ties with the process identifier
  - Useful for liveness properties: e.g. serve requests according to total order

Properties of Lamport's Clock

- Lamport's Clock is consistent with causality but not *strongly consistent*:
  - $\mathrm{LC}(a) < \mathrm{LC}(b) \not\Rightarrow a \prec b$
  - Events which are not causally related may be ordered in LC time.
    - $\mathrm{LC}(p_0^3) = 3 < \mathrm{LC}(p_1^4) = 5 < \mathrm{LC}(p_2^3) = 6$, but all three events are concurrent.
  - This is because it uses one variable to store two different pieces of information: the local time and the global time. $\mathrm{LC}(p_2^2) = 5$ but now p_2 has no record that the latest time it has seen at p_1 is 4 or at p_0 is 2.
- Sometimes, we want a strongly consistent logical clock

Vector Clock

Each process i maintains a vector $v_i[0, ..., N-1]$ of integers of the same length as the number of processes in the system

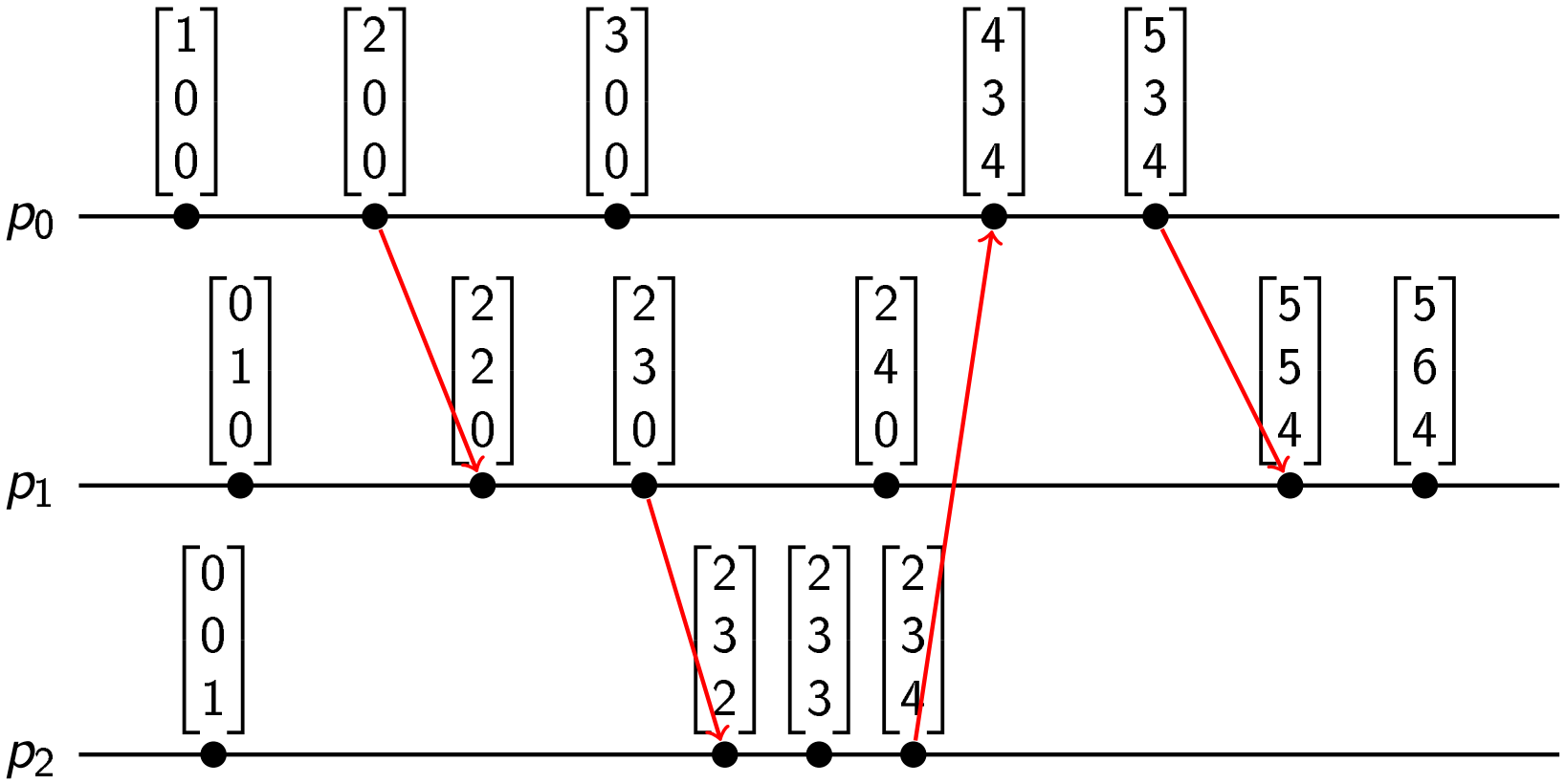
- $v_i[i]$ is the local logical clock of process i
- $v_i[j]$, where $i \neq j$, is process i's most recent knowledge of process j's local time

Each process i executes the following:

- Initialize all clocks to the 0 vector
- Let a be an event
  - If a is an internal or send event, $v_i[i] = v_i[i] + 1$
  - If a is a receive event, and m is the vector clock sent with the message, then

```
for (int j = 0 ; j < N ; j++)
    v_i[j] = max(v_i[j], m[j])
v_i[i] += 1
```

Vector Clock Example



Properties of Vector Clocks

Define the order on vector clocks to be:

$$
\begin{aligned}
u = v &\iff \forall i \,.\, u[i] = v[i] \\
u \leq v &\iff \forall i \,.\, u[i] \leq v[i] \\
u < v &\iff u \leq v \wedge (\exists i \,.\, u[i] < v[i]) \\
u \| v &\iff u \not\leq v \wedge v \not\leq u
\end{aligned}
$$

Vector clocks are *strongly consistent* with causality:

$$a \prec b \iff \mathrm{VC}(a) < \mathrm{VC}(b)$$

Because they track causal dependencies exactly, vector clocks have many applications where Lamport clocks are insufficient:

- Distributed debugging
- Global breakpointing
- Consistency of checkpoints in optimistic recovery
- Implementations of causal distributed shared memory
- etc.

Centralised Mutual Exclusion in a Distributed System

Sometimes there are scarce resources that can only be used by one node at a time. In order to manage access to the resource, the nodes in a distributed system need to cooperate on their access. If we elect one node to be the coordinator, a solution is simple:

- To gain access to the resource, nodes must request access from the coordinator by sending a *request* message.
- If no other node is currently using the resource, the coordinator grants access by responding with a *grant* message, upon receipt of which the requestor can use the resource.
- If another node is currently using the resource, the coordinator queues the request and does not respond (yet), thus causing the requestor to wait.
- When a node finishes with the resource, it stops using it and sends a *release* message to the coordinator to tell it so.
- When the coordinator receives that release message, it takes the first request message off its queue (if any) and sends a *grant* message to the sender

Centralised Mutual Exclusion

Advantages:

- It works!
- Easy to implement
- Fair: access granted in order of requests
- No starvation: no node waits forever (assuming requesting nodes don't hold on to the resource forever)
- Only 3 messages per use of the resource

Disadvantages:

- Coordinator is a single point of failure
- In a large system, the coordinator can become a bottleneck

Distributed Mutual Exclusion (Ricart-Agrawala)

To request a resource, a node sends a message to all nodes requesting the resource and including its node number and the local logical clock value.
On receipt of a request, the recieving nodes reacts according to the state it is in:

- Not using and does not want to use the resource: it sends an OK message back to the sender
- Already has access: it does not reply but queues the request
- Wants the resource but does not have it yet: It compares the logical clock value of the message with that of its own request message. If the incoming message has the lower clock value, it sends back an OK message. Otherwise it queues the other message and sends nothing.

After sending out a request message, the sender waits for an OK from everyone else. When it has them, it can access the resource. When finished, it pops all requests off its queue and sends OK to each of them.

Distributed Mutual Exclusion (Ricart-Agrawala)

Advantages:

- It Works
- Fair
- No starvation

Disadvantages:

- $2(N-1)$ messages
- N points of failure
- Every node needs to keep track of all the nodes in the system