Distributed Computing

Distributed System

- A distributed system is a network that consists of autonomous computers that are connected using a distribution middleware.

- They help in sharing different resources and capabilities to provide users with a single and integrated coherent network.

- Main features:
  - Several autonomous nodes, each of which has its own local memory
  - Nodes communicate with each other by message passing

Properties

- Location Trasparency: resource-centric
- Concurrent computations on different nodes, no shared memory
- Absence of a global state
- Fault tolerance: the system must continue operating properly in case of failure of some of its components.
- Heterogeneity in hardware and software

Architectures

- Client-server architectures: clients contact the server for data; results are committed back to the server when they represent a permanent change.
- Three-tier: architectures that move the client intelligence to a middle tier so that stateless clients can be used (e.g web applications)
- N-tier architectures: they forward requests to other enterprise services (e.g. application servers)
- Peer-to-peer: architectures where there are no special machines that provide a service or manage the network resources: responsibilities are uniformly divided among all machines known as peers.
  Peers can serve both as clients and as servers

Communication and Coordination

- Master/slave:
  Processes communicate directly with one another

- Database-centric:
  Communication via a shared database

Synchronous Distributed System

- Upper Bound on Message Delivery
- Ordered Message Delivery
- Notion of Globally Synchronized Clocks
- Lock Step Based Execution

Asynchronous Distributed System

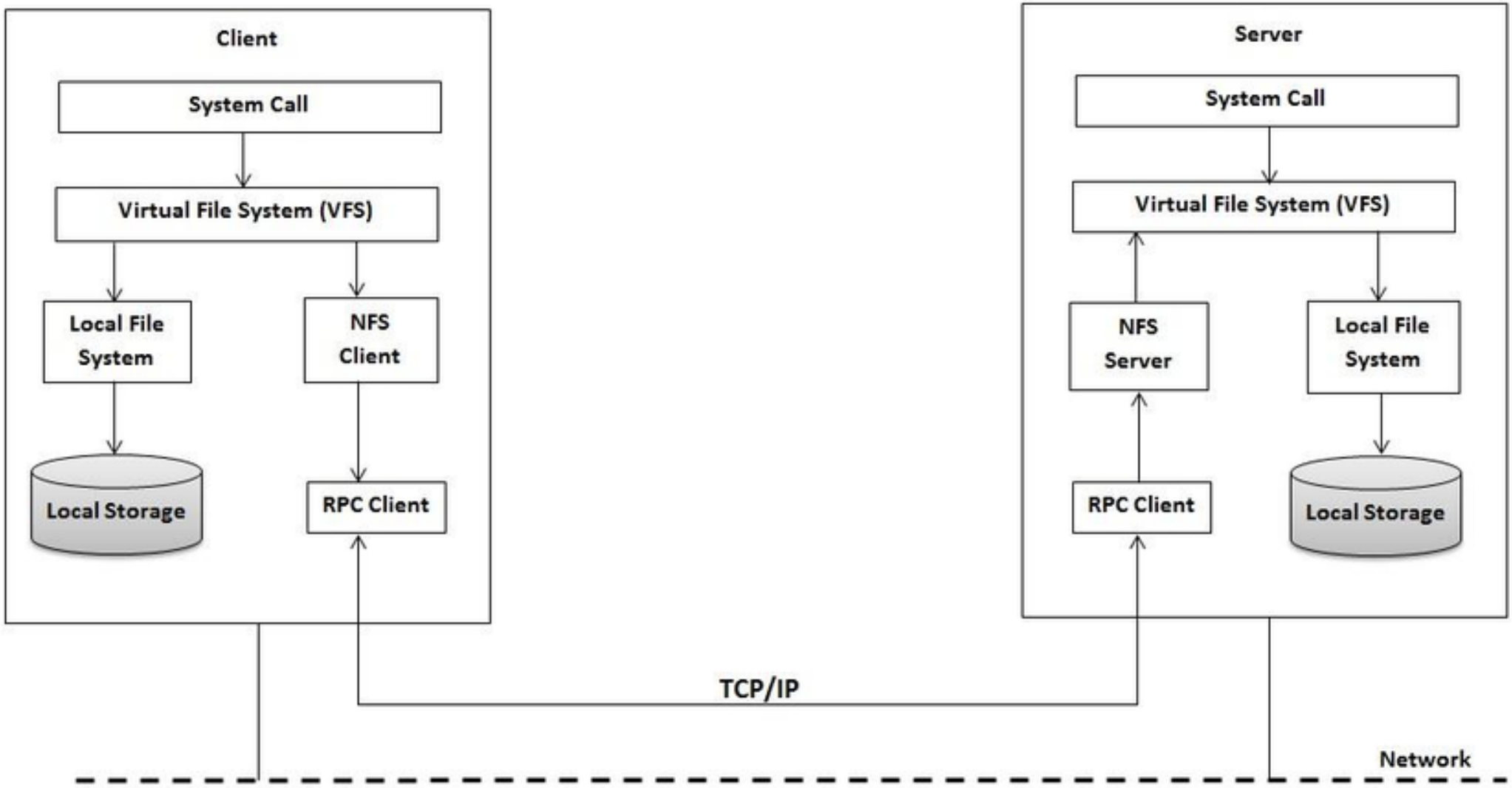
- Clock may not be accurate and can be out of sync
- Messages can be delayed for arbitrary period of times

Desiderata

- Interoperability
- Integration
- Flexibility (deal with modifications at runtime)
- Modularity
- Scalability
- Quality of service and availability
- Transparency (w.r.t. data, location, migration, replication, concurrency, failures)

Examples

Network File System



Distributed Objects

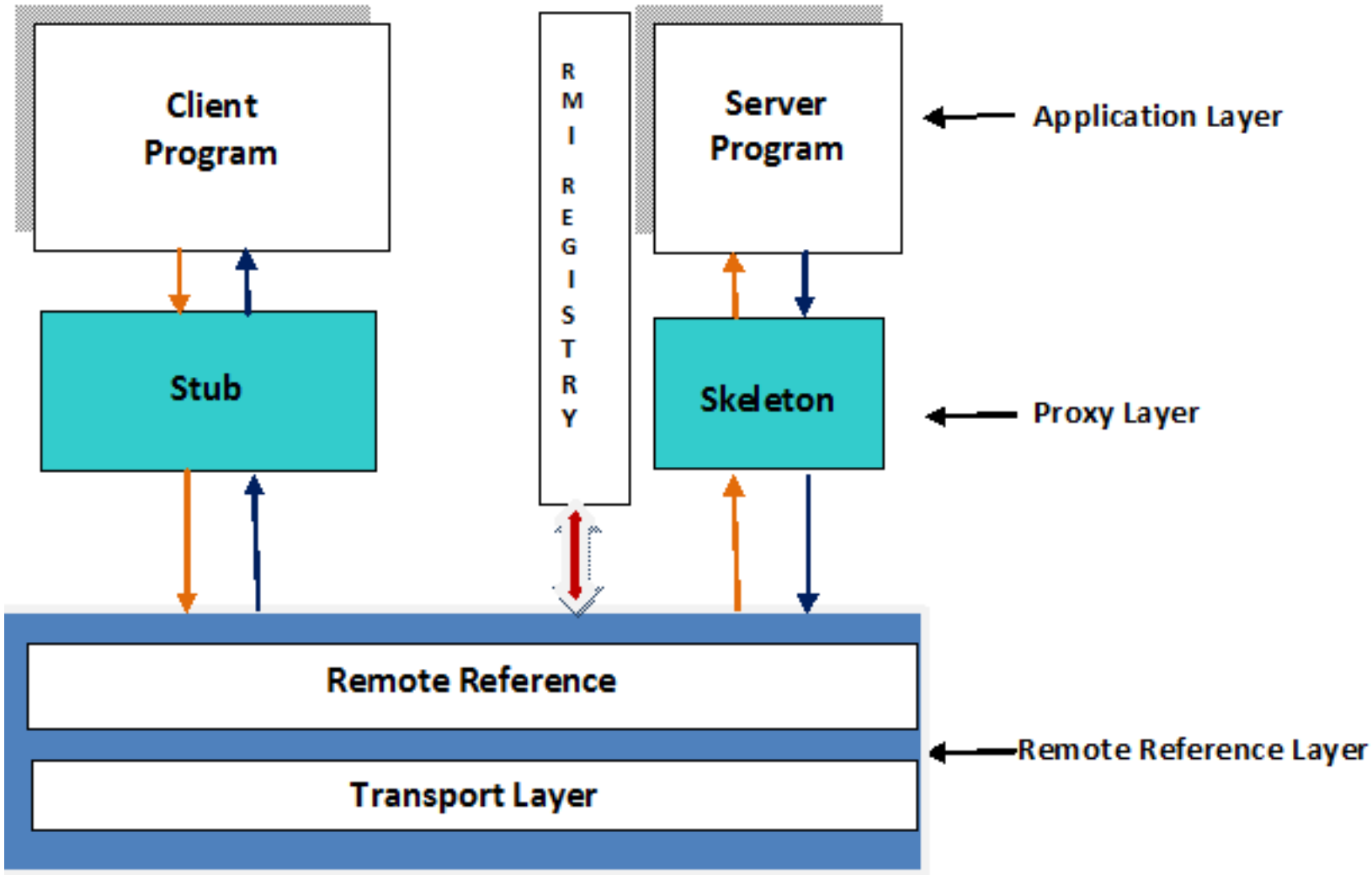
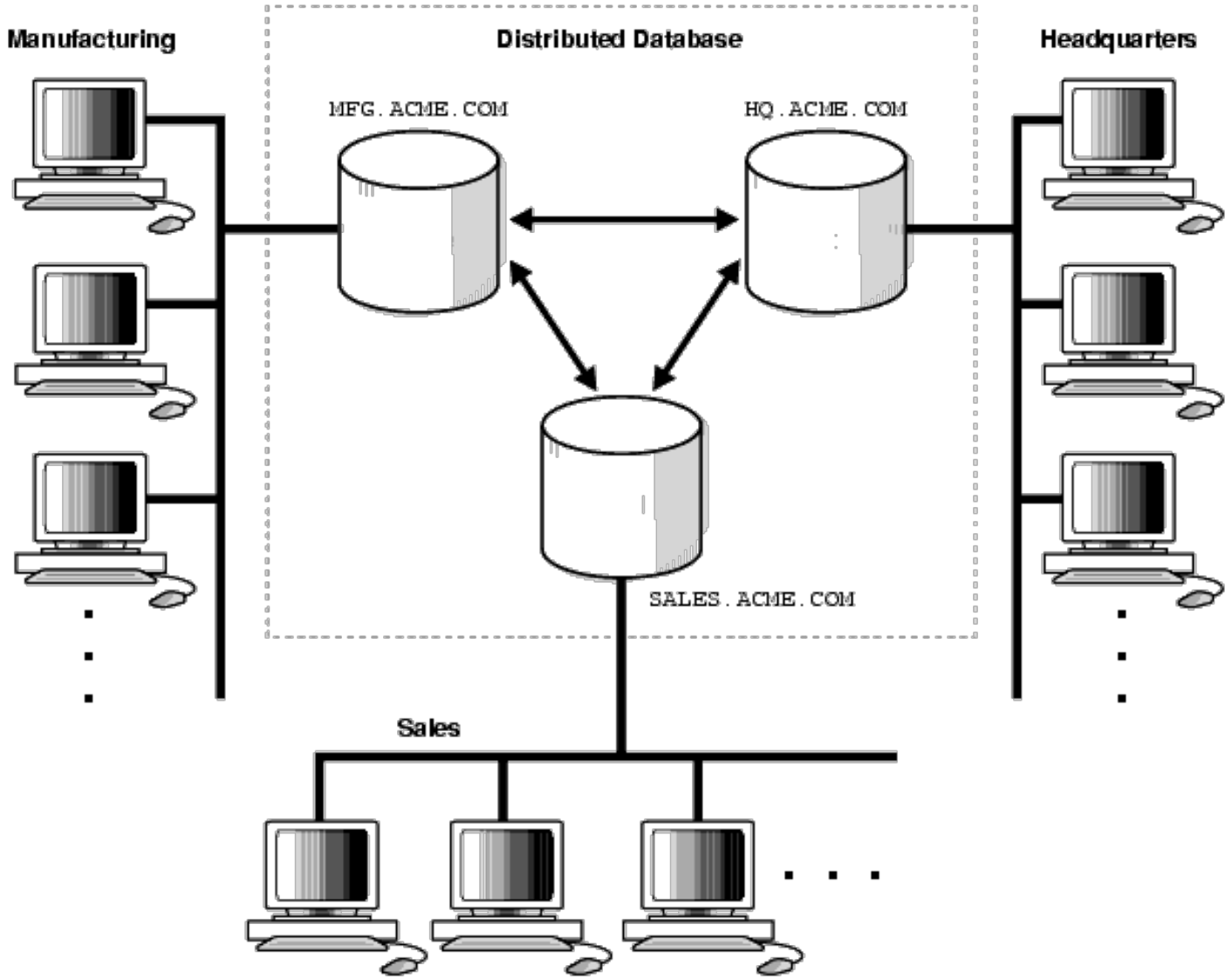
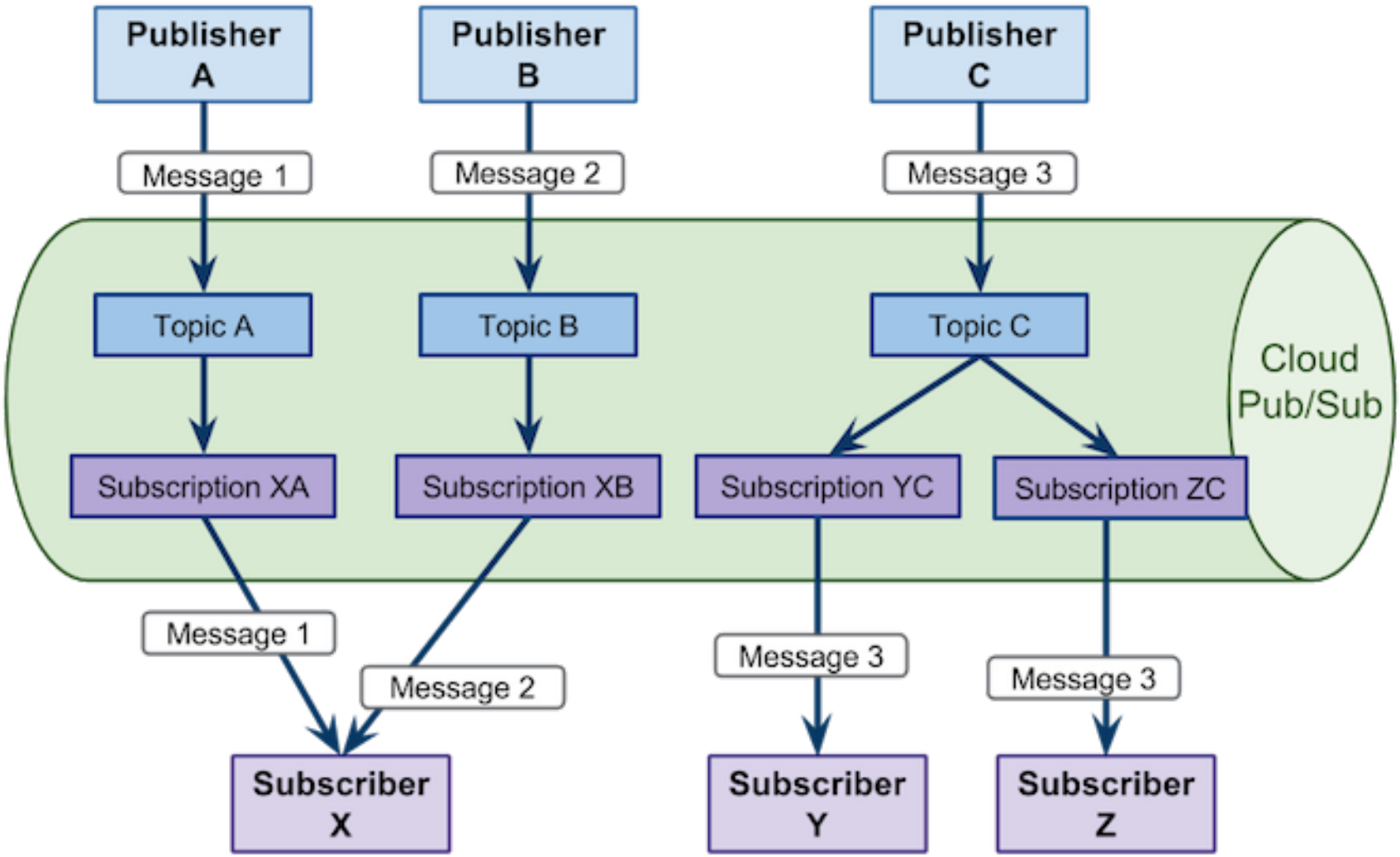


Fig: RMI Architecture

Distributed Database



Pub/Sub architectures



Classical Dist. Comp. Problems

Classical problems for distributed systems:

- Leader election
- Mutual exclusion
- Clock synchronisation
- Global snapshot
- Consensus in presence of failures
- Routing/route maintenance

Assumptions:

- synchronous/asynchronous
- message loss/delays
- node/byzantine faults

Distributed Algorithms

Distributed algorithms are often formulated as algorithms working on graphs (shortest path, minimum cost, spanning tree, vertex cover,…)

- Network: Graph
- Node: Process
- Edge: Communication link via message passing

Distributed Algorithms need Multithreaded Programs

Distributed systems are based on client-server and peer-to-peer architectures.

In both cases each node may need to both send and receive data.

In some situations we can sequentially alternate send and receive, e.g., hand-shaking protocols

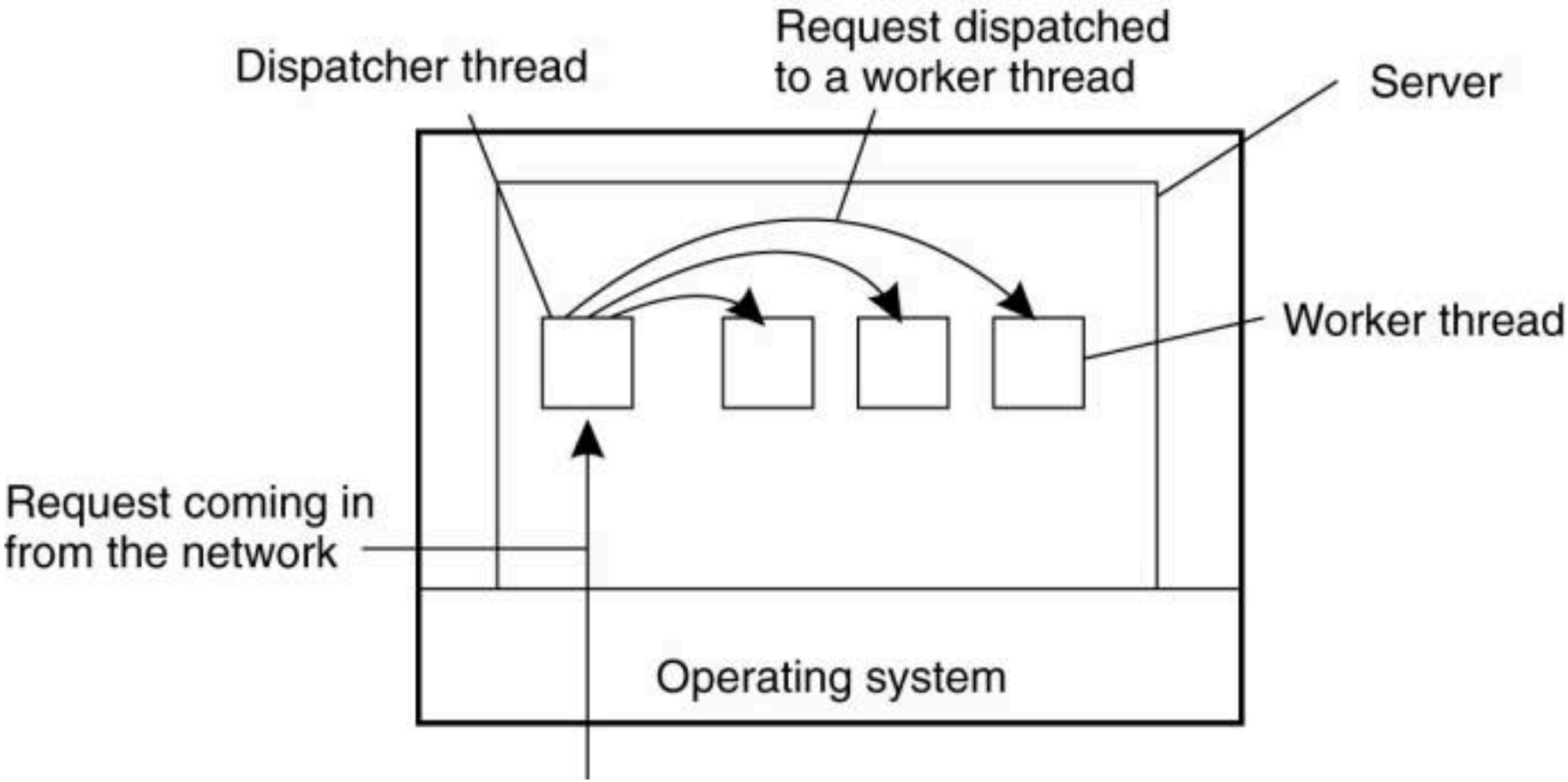
A: "how are you?"
B: “fine and you?"
A: "everything is ok, thanks"
...

```
A:
send (B,"how are you?")
forever {
—receive(B,x);
---analyze(x,answer);
---send(B,answer);
}
```

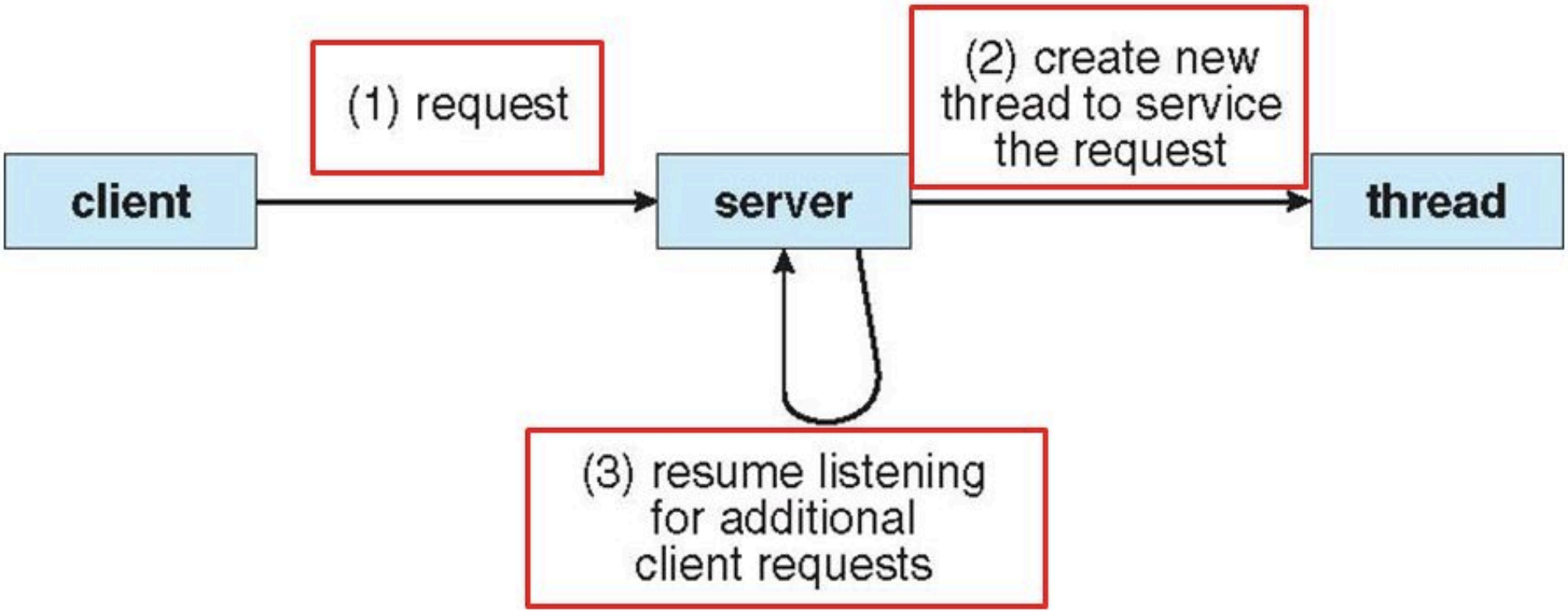
```
B:
forever {
—receive(A,x);
---analyze(x,answer);
---send(A,answer);
}
```

However in a client-server architecture a server has to answer multiple clients.
One possible architecture to avoid to create bottlenecks (i.e. serve clients sequentially)
is to use multithreaded programming

Multithreaded Server



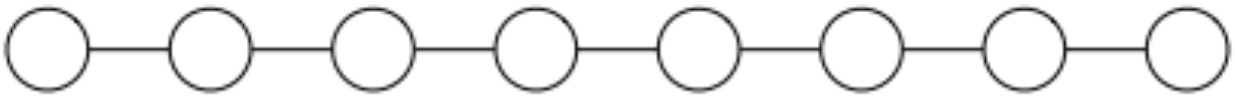
Multithreaded Server Architecture



How to Design Distributed Algorithms

Example: Colouring Paths

Imagine that we have n computers (or *nodes* as they are usually called) that are connected to each other with communication channels so that the network topology is a *path*:



The computers can exchange messages with their neighbours. All computers run the *same* algorithm — this is the *distributed algorithm* that we will design. The algorithm will decide what messages a computer sends in each step, how it processes the messages that it receives, when it stops, and what it outputs when it stops.

Colouring Paths

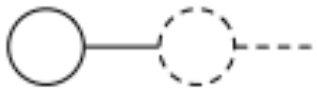
In this example, the task is to find a proper *colouring* of the path with 3 colours. That is, each node has to output one of the colours, 1, 2, or 3, so that neighbours have different colours — here is an example of a proper solution:

Colouring Paths

It looks like a simple problem to solve

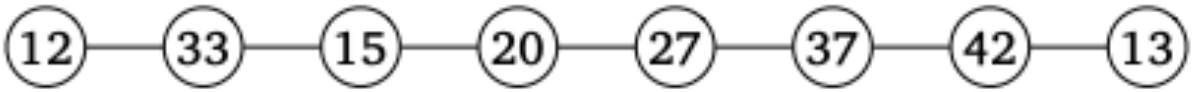
However nodes have no global knowledge of the network

Indeed, when we start a networked computer, it is typically only aware of itself and the communication channels that it can use. In our simple example, the endpoints of the path know that they have one neighbour:



All other nodes along the path just know that they have two neighbours:

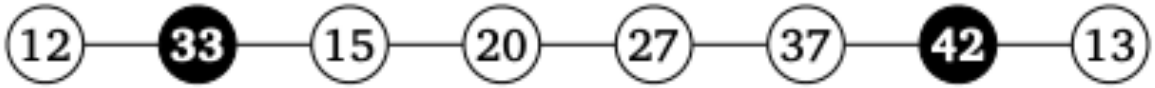
Colouring Paths when Node have Unique Ids



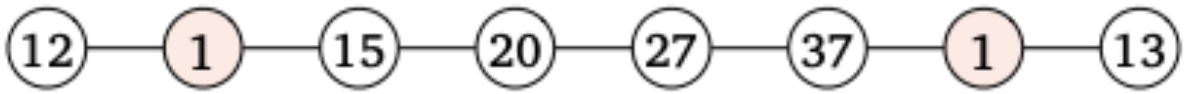
With the help of unique identifiers, it is now easy to design an algorithm that colours a path. Indeed, the unique identifiers already form a colouring with a large number of colours! All that we need to do is to reduce the number of colours to 3.

Colouring Paths when Node have Unique Ids

We can use the following simple strategy. In each step, a node is active if it is a "local maximum", i.e., its current colour is larger than the current colours of its neighbours:

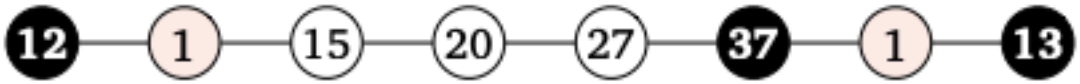


The active nodes will then pick a new colour from the colour palette $\{1, 2, 3\}$, so that it does not conflict with the current colours of their neighbours. This is always possible, as each node in a path has at most 2 neighbours, and we have 3 colours in our colour palette:

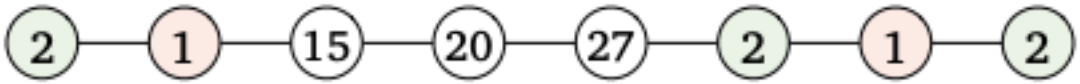


Colouring Paths when Node have Unique Ids

Then we simply repeat the same procedure until all nodes have small colours. First find the local maxima:

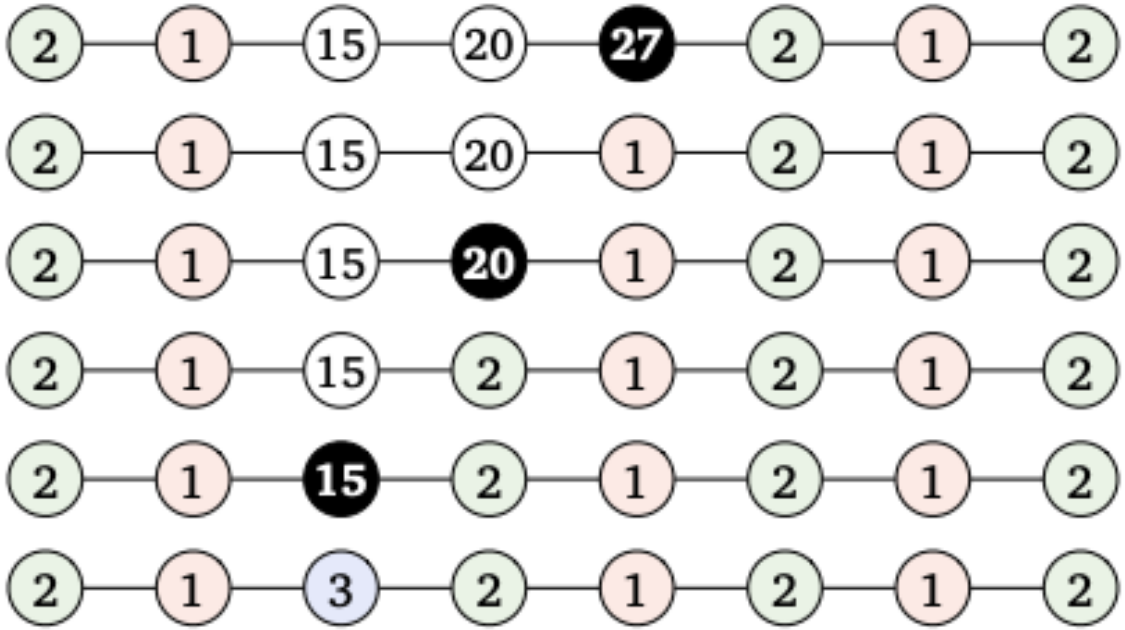


And then recolour the local maxima with colours from $\{1, 2, 3\}$:



Colouring Paths when Node have Unique Ids

Continuing this way we will eventually have a path that is properly coloured with colours $\{1, 2, 3\}$:



Algorithm

Repeat forever:

- Send message c to all neighbours.
- Receive messages from all neighbours.
  Let M be the set of messages received.
- If $c \notin \{1, 2, 3\}$ and $c > \max M$:
  Let $c \leftarrow \min(\{1, 2, 3\} \setminus M)$.

Properties

- The nodes are state machines that repeatedly send messages to their neighbours, receive messages from their neighbours, and update their state — all nodes perform these steps synchronously in parallel.
- Some of the states are stopping states, and once a node reaches a stopping state, its no longer changes its state.
- Eventually all nodes have to reach stopping states, and these states must form a correct solution to the problem that we want to solve.

Randomised Algorithm

So far we have used unique identifiers to break symmetry. Another possibility is to use randomness. Here is a simple randomised distributed algorithm that finds a proper 3-colouring of a path: nodes try to pick colours from the palette $\{1, 2, 3\}$ uniformly at random, and they stop once they succeed in picking a colour that is different from the colours of their neighbours.

Randomised Algorithm

- **Initialise: state = unhappy, colour = 1**
- **While state = unhappy:**
  - pick a new random colour from {1, 2, 3}
  - compare colours with neighbours
  - if different, set state = happy

Randomised Algorithm

Fix a positive constant C. Consider what happens if we run the algorithm for

$$k = (C+1)\log_{3/2} n$$

Now the probability that a given node u has not stopped after k steps is at most

$$(1-1/3)^k = \frac{1}{n^{C+1}}.$$

By the union bound, the probability that there is a node that has not stopped is at most $1/n^C$. Hence with probability at least $1-1/n^C$, all nodes have stopped after k steps.

Union bound: $\mathbb{P}\left(\bigcup_i A_i\right) \leq \sum_i \mathbb{P}(A_i).$

Randomised Algorithm

Let us summarise what we have achieved: for any given constant C, there is an algorithm that runs for $k = O(\log n)$ rounds and produces a proper 3-colouring of a path with probability $1 - 1/n^C$. We say that the algorithm runs in time $O(\log n)$ *with high probability* — here the phrase "high probability" means that we can choose any constant C and the algorithm will succeed at least with a probability of $1 - 1/n^C$. Note that even for a moderate value of C, say, $C = 10$, the success probability approaches 1 very rapidly as n increases.

Lamport's Logical Clocks

Lamport, L. (1978).
"Time, clocks, and the ordering of events in a distributed system"
Communications of the ACM . **21** (7): 558–565

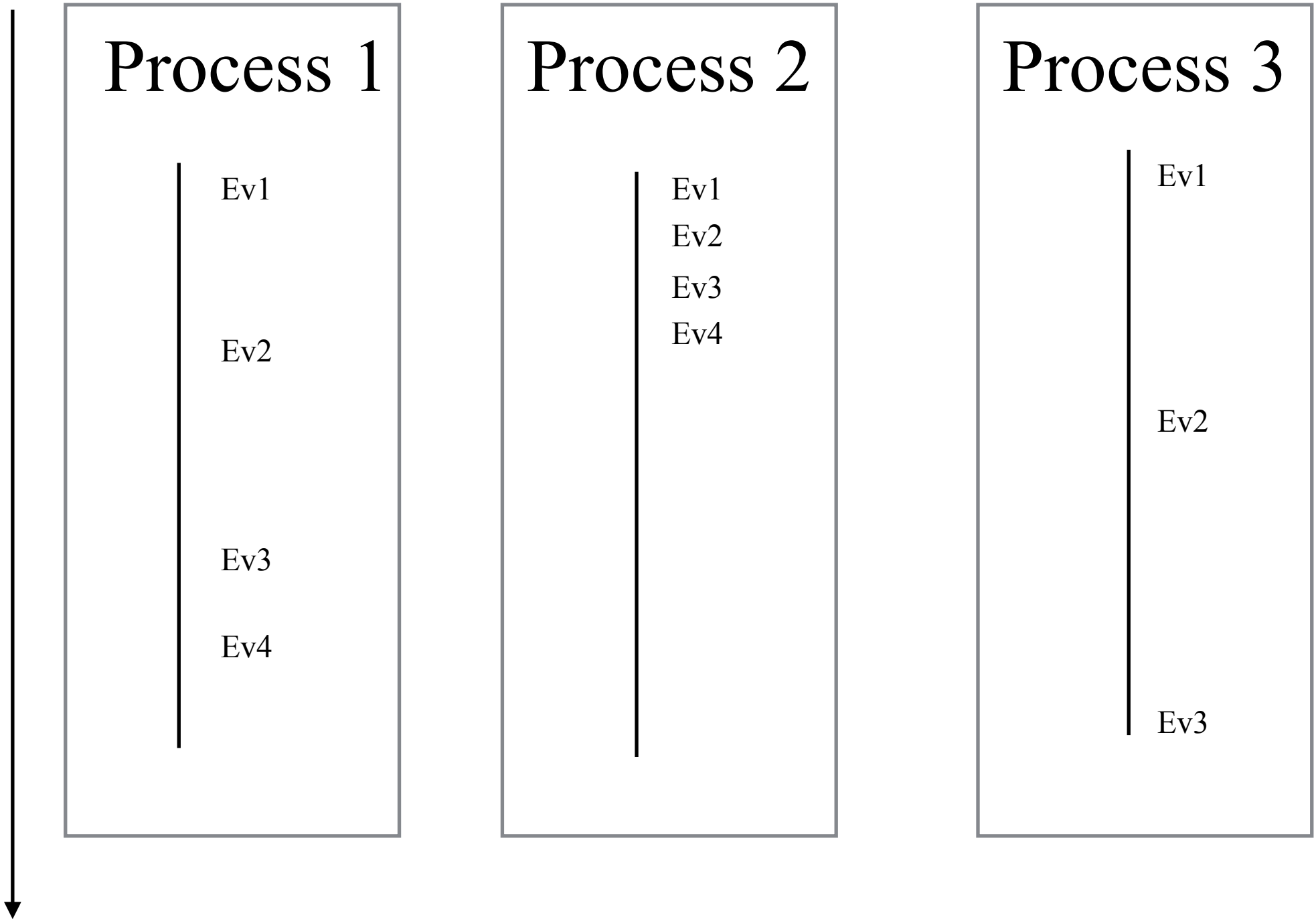
Logical Clocks

Algorithms based on logical clocks make use of timestamps to sort operations on different nodes of a network (e.g. versioning, consistency, merge, etc)

We need to adjust individual clock in order to use them a global timestamps

Different versions:

- Scalar clocks
- Vectorial clocks

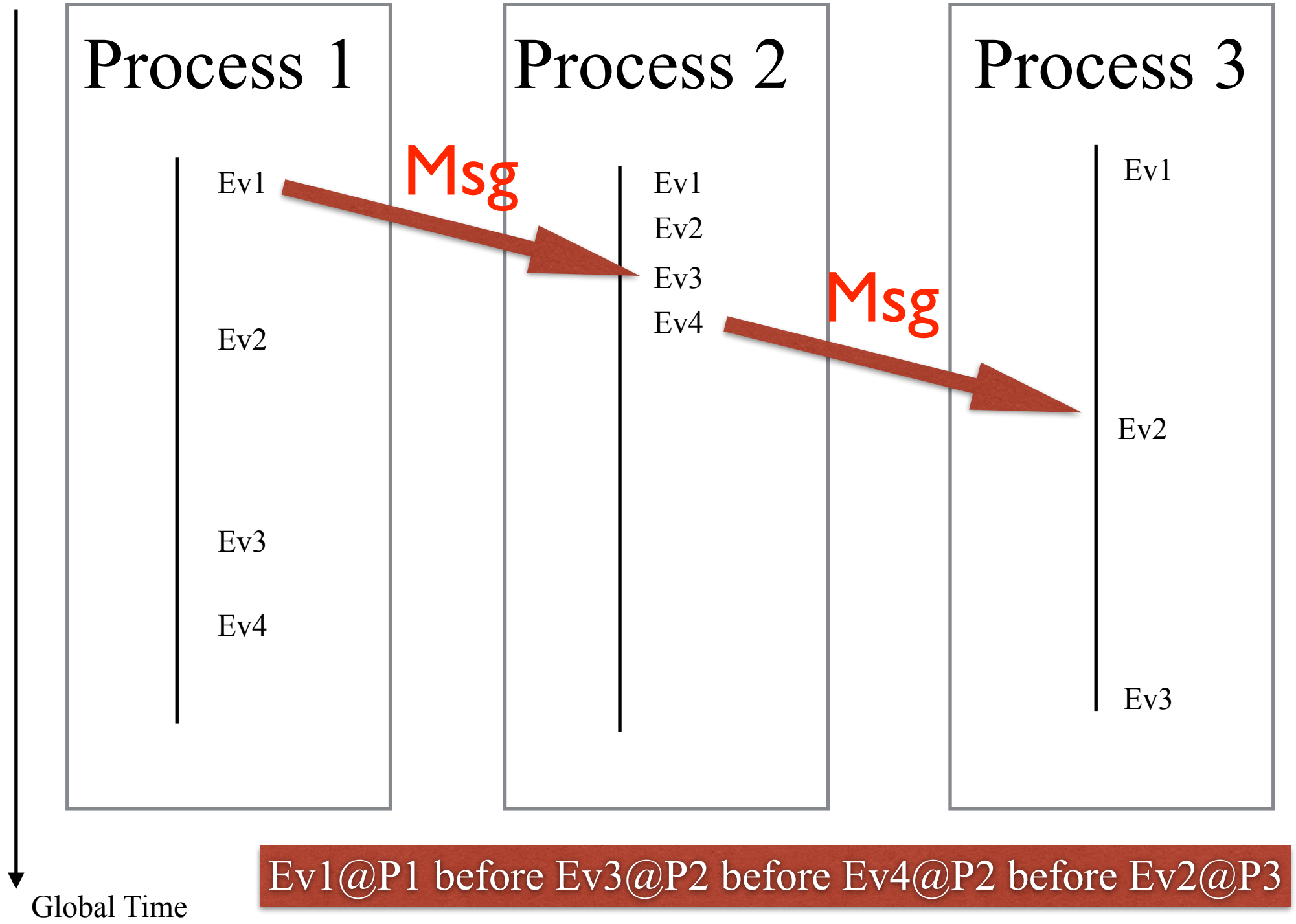

Process 1
Ev1
Ev2
Ev3
Ev4
Process 2
Ev1
Ev2
Ev3
Ev4
Process 3
Ev1
Ev2
Ev3
Global Time

Happens Before

The relation a → b defines the event "a happened before b" in a set of distributed nodes

- if a and b are events in the same node/process then a → b
- if a is the event of sending message "m" and b is the event of its reception in a different node then a → b
- if a → b and b → c, then a → c (transitive)

We would like to find a way to assign to each event "e"
a timestamp C(e) such that if a → b then C (a) < C (b)


Process 1
Process 2
Process 3
Ev1
Ev2
Ev3
Ev4
Msg
Ev1
Ev2
Ev3
Ev4
Msg
Ev1
Ev2
Ev3
Global Time
Ev1@P1 before Ev3@P2 before Ev4@P2 before Ev2@P3

Process@Node j

local timestamp
int clock=0;

for each internal event
clock ← clock+1

for each send
clock ← clock+1;
send(...,clock,...);

for each receive
receive(...,timestamp,...,);
clock ← max(clock,timestamp)+1;

In other words C(e@j)= value of clock variable in j when e occurs in j

Properties

By construction,

$$a \rightarrow b \text{ implies } C(a) < C(b)$$

Properties

In general

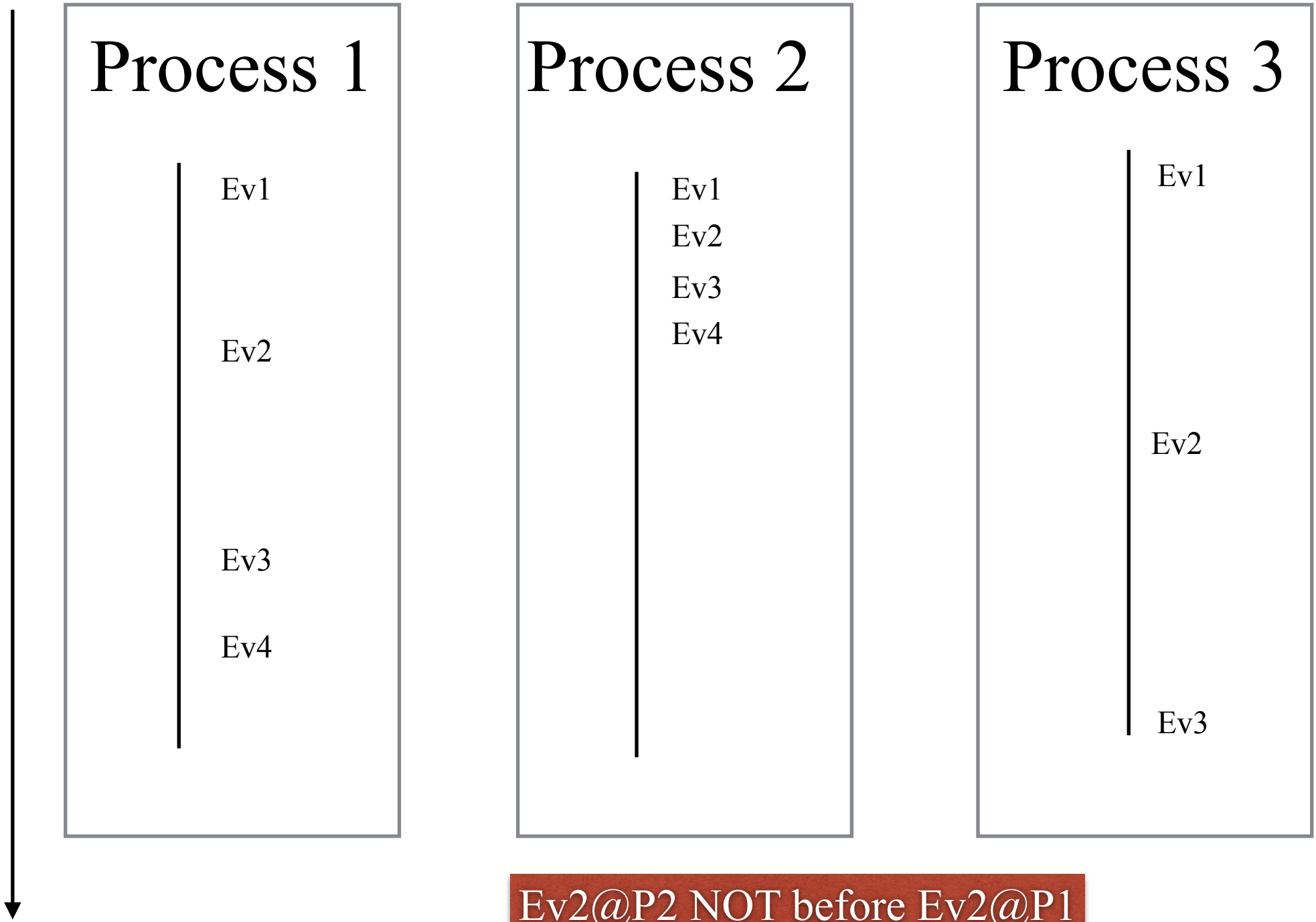
$C(a) < C(b)$ does not imply $a \rightarrow b$

Properties

In general

$$C(a) < C(b) \text{ does not imply } a \rightarrow b$$

In other words scalar timestamp cannot be used to identify concurrent events


Process 1
Ev1
Ev2
Ev3
Ev4
Process 2
Ev1
Ev2
Ev3
Ev4
Process 3
Ev1
Ev2
Ev3
Global Time
Ev2@P2 NOT before Ev2@P1

Vectorial Clocks (VC)

Every node maintains a local representation of the logical clocks of all other nodes (for N nodes a vector of N elements)

V[i] in node j = *what node j knows about the logical clock of process* i

Node with index I

local vector int V[N]={ 0,...,0 } ;

Internal event

V[i] ← V[i]+1

send

V[i] ← V[i]+1;
send(...,V,...);

receive

receive(...,T,...,);
for every j : 1 ,…N :
V[j] ← max(V[j],T[j]);
V[i] ← V[i]+1;

Node with index I

local vector int V[N]={ 0,...,0 } ;

Internal event

V[i] ← V[i]+1

send

V[i] ← V[i]+1;
send(...,V,...);

receive

receive(...,T,...,);
for every j : 1 ,…N :
V[j] ← max(V[j],T[j]);
V[i] ← V[i]+1;

Gossiping!
Node i sends his vector to Node j …

Node with index I

local vector int V[N]={ 0,...,0 } ;

Internal event

V[i] ← V[i]+1

send

V[i] ← V[i]+1;
send(...,V,...);

receive

receive(...,T,...,);
for every j : 1 ,…
V[j] ← max(V[
V[i] ← V[i]+1;

Partial View of Global State!
A node knows only what receives

Vector Clocks Ordering

$V \leq W$ iff $V[i] \leq W[i]$ for all $i : 1, \ldots, N$
$V < W$ iff $V \leq W$ there exists i s.t. $V[i] < W[i]$
$V = W$ iff $V \leq W$ and $W \leq V$.

Let C(e) be the vector clock when e occurs

Then the following property holds

$C(a) < C(b)$ if and only if $a \rightarrow b$

Summary

- Distributed Systems = Set of cooperating nodes
- Distributed algorithms = each node executes a concurrent program that exchange messages with other nodes
- A distributed algorithm executes in rounds, in each round nodes send messages in parallel and change state in accord with received messages
- Complexity = communication cost (number of messages) and number of rounds