# 1. Concepts of Distributed Systems

Concurrence, Parallelism and Distributed Systems (CPDS)
Facultat d'Informàtica de Barcelona (FIB)
Universitat Politècnica de Catalunya (UPC)
2018/2019 Q2

FIB

# Instructor

- Instructor: Jordi Guitart
- E-mail: jguitart@ac.upc.edu
- Office: C6-205
- Office Hours: Monday 15:00pm — 18:00pm
  Thursday 15:00pm — 18:00pm
- Arrange an appointment by mail

# Contents

- **Definition of a distributed system**
- Challenges of distributed systems
- Architectures for distributed systems

# Definition of a distributed system

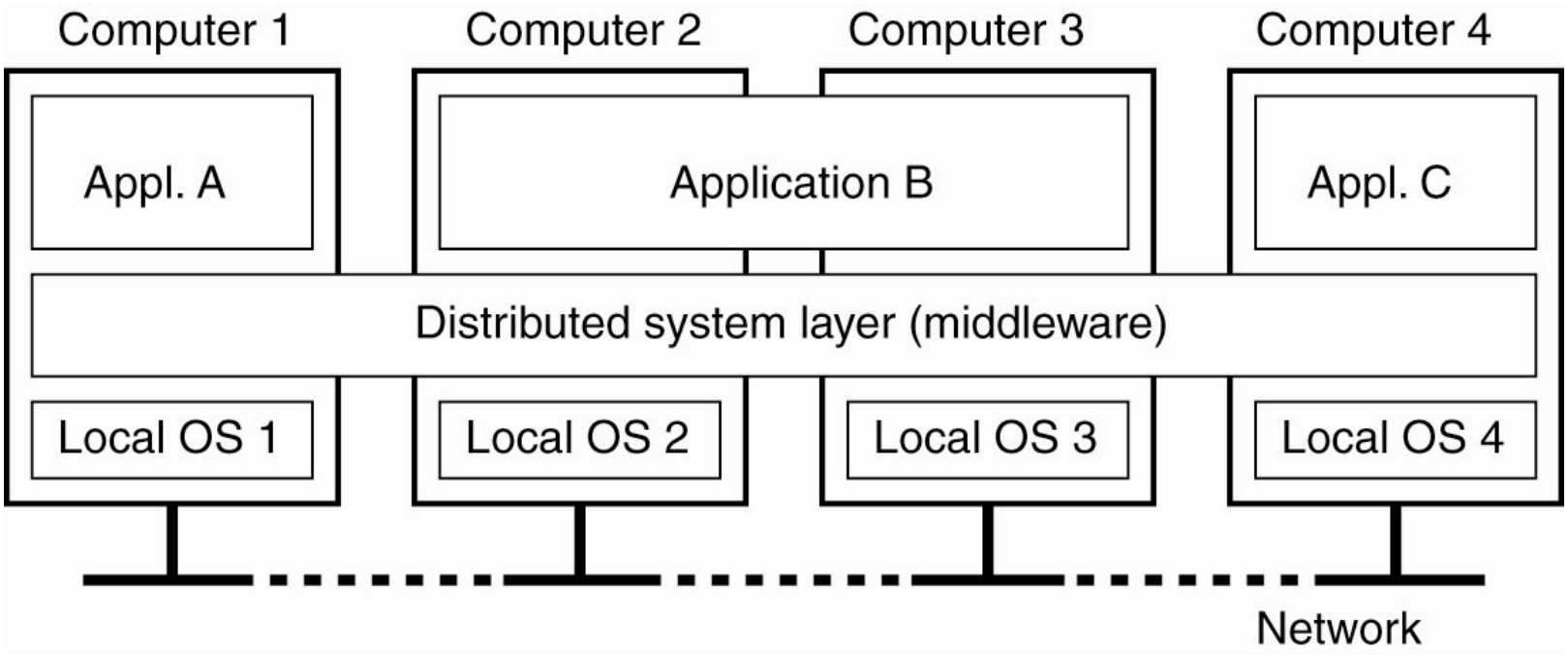
- It is one in which the failure of a computer you didn't even know existed can render your own computer unusable
  - [Lamport87]
- This apparently humorous definition entails the basic aspects of a distributed system:
  - The transparency of distribution ("a computer you didn't even know existed")
  - The interdependency of components ("the failure ... can render your own computer unusable")

# Definition of a distributed system

- A collection of autonomous computing elements that appears to its users as a single coherent system
  - [Tanenbaum]
- A system where components located at networked computers communicate and coordinate their actions only by passing messages
  - [Coulouris]

# Definition of a distributed system

- (1) independent networked computers but
- (2) single-system view: **middleware**



# Definition of a distributed system

- Consequences of the definition:

1. Multiple heterogeneous computers
2. Multiple autonomous processes
3. Processes execute concurrently
4. Knowledge on each computer is local
5. Process address space is not shared
6. Only communication is by message passing
7. Network delays in process communication

# Definition of a distributed system

- Consequences of the definition:

8. Independent (imperfect) clocks (no global time and perfect synchronization unfeasible)
9. Processes can fail independently
10. Processes may not always be available (network partitions, process disconnections)
11. High probability of failures (it grows with number of components)

# Definition of a distributed system

- Why do we build distributed systems?
  - Functional: computers have different functionality
    - e.g. clients vs. servers
  - Load distribution: work divided among computers
    - e.g. P2P, distributed computing systems (SETI@HOME)
  - Fault tolerance: replication
  - Accommodate a natural geographic distribution
    - e.g. weather stations, remote resources
  - Resource sharing
    - e.g. printer in a LAN
  - Economic: many computers cheaper than one big

# Contents

- Definition of a distributed system

- **Challenges of distributed systems**

- Architectures for distributed systems

# Challenges of distributed systems

1. Heterogeneity
   - Need to accommodate different computers, programming languages, operating systems, …
2. Security
   - Need for confidentiality, integrity, authentication across administrative domains, …
   - Out of the scope of this course
3. Global view
   - Need to assemble meaningful global views of the system
   - Some problems that we will cover in Module III:

# Challenges of distributed systems

a) Global time
   - Only low accuracy system clock on each node
   - Results in clock skew (two clocks, two times) and clock drift (two clocks, two count rates)

   i. Synchronize machines within a given bound with a master (with a UTC receiver) or with one another

   ii. Agree on the **order** in which events occur rather than the **time** at which they occurred

b) Global state
   - Each process is independent
   - Perfect clock synchronization is not feasible

   i. We need to assemble a meaningful global state from local states recorded at different real times

# Challenges of distributed systems

4. Coordination
   - Need to coordinate actions in different machines, as they are running processes concurrently and autonomously (no single point of control)

   - Some problems that we will cover in Module III:

   a) Mutual exclusion
      - Processes coordinate their accesses to some shared resource (critical section) using solely message passing

   b) Leader election
      - Election of a unique coordinator of a group of processes (many algorithms need a process to act as coordinator)

# Challenges of distributed systems

c) Atomic multicast

- Agree on which messages a group of processes receive and in which order. For instance, in a bulletin board:

| Bulletin board: *os.interesting* | | |
|---|---|---|
| Item | From | Subject |
| 23 | A.Hanlon | Mach |
| 24 | G.Joseph | Microkernels |
| 25 | A.Hanlon | Re: Microkernels |
| 26 | T.L'Heureux | RPC performance |
| 27 | M.Walker | Re: Mach |
| end | | |

d) Consensus

- A set of processes agree on some value after one or more of them have proposed what that value should be

# Challenges of distributed systems

5. Asynchrony
   - Messages take (variable) time to be delivered
   - This results on two types of distributed systems:

   A. Synchronous distributed systems
      - Process execution time, message delay and clock drift rate have known bounds
      - **Timeouts** can be used to detect reliably process failures, lost messages, …

   B. Asynchronous distributed systems
      - Process execution time, message delay and clock drift rate are not bounded
      - No assumptions can be made about time intervals on any execution

# Challenges of distributed systems

6. Openness
   - Interoperability
     - Components from different manufacturers can work together by merely relying on each other's services
   - Portability
     - Component made for a given system can run unmodified on another one implementing the same interfaces
   - Extensibility
     - Ability to add new components or replace existing ones without affecting those that stay in place

⇒ Offer services according to standard interfaces describing their semantics and syntax

# Challenges of distributed systems

## 7. Transparency

- Ability of a distributed system of presenting itself as a single computer system

| Transparency | Description |
|---|---|
| **Access** | Enable local and remote objects to be accessed using identical operations by hiding differences in data representation and invocation mechanisms |
| **Location** | Enable objects to be accessed without knowledge of their physical location |
| **Mobility** | Allow the movement of objects and users within a system without affecting their operation |
| **Replication** | Allow multiple instances of an object to exist without knowledge of the replicas by users |
| **Concurrency** | Enable several users to operate concurrently on shared objects without interference between them |
| **Failure** | Mask from an object the failure and possible recovery of other objects |

# Challenges of distributed systems

- Transparency may be set as a goal, but fully achieving it is a different story
  A. There are communication latencies (due to distant locations) that cannot be hidden
  B. Completely hiding failures is impossible
     - In asynchronous systems, you cannot distinguish a slow node from a failing one
  C. Trade-off between transparency & performance
     - e.g. replication transparency requires keeping replicas up-to-date with the master copy
     - e.g. failure transparency tries repeatedly to contact a server before trying another one

# Challenges of distributed systems

- Transparency may be set as a goal, but fully achieving it is a different story
  D. Exposing distribution could be good, especially when location is important
    - e.g. use of location-based services on mobile phones (finding your nearby friends or the nearest restaurant)
    - e.g. use a busy nearby printer rather than an idle one in another building
    - e.g. deal with users in different time zones

# Challenges of distributed systems

8. Fault tolerance
   - A failure is any deviation of the observed behavior of a system from the specified behavior
   - Failures on a single machine often affect the **entire** system
   - Failures on distributed systems can be **partial** (only some nodes fail)
   - A **fault tolerant system** is able to meet its specifications even with partial failures
   - Important ability since the probability of failure grows with number of nodes in the system

# Fault tolerance

- Classification of failure models

| Type of failure | Description |
|---|---|
| **Crash failure**<br>*Fail-stop crash*<br>*Fail-silent crash* | Process halts and remains halted<br>Other processes can reliably detect the failure<br>Other processes may not be able to detect the failure |
| **Omission failure**<br>*Receive omission*<br>*Send omission* | Sent message never arrives at the other end<br>Process fails to receive incoming messages<br>Process fails to send outgoing messages |
| **Timing failure** | Process response lies outside a specified time interval |
| **Response failure**<br>*Value failure*<br>*State transition failure* | Process response is incorrect<br>The value of the response is wrong<br>The process deviates from the correct flow of control |
| **Arbitrary (Byzantine) failure** | Process may produce arbitrary messages at arbitrary times, commit omissions, stop, or take incorrect steps |

# Fault tolerance

- Use **failure detectors** to know about crash failures of processes
  - They are generally based on the use of timeouts
  - Setting timeouts properly is not easy
- Two implementation options:
  A. Heartbeats: I am alive!
  B. Pinging: Are you alive?: Yes!
- How reliable are the detectors?
  A. Unreliable
  B. Reliable

# Fault tolerance

**A. Unreliable**: a process can be …

- Unsuspected: Recent evidence that the process has not failed
  - e.g. a message has recently received from it
- Suspected: Hint that the process may have failed
  - e.g. no messages from the process from a nominal maximum time

**B. Reliable**: a process can be Unsuspected or

- Failed: the detector has determined that the process has failed
- These are only possible in synchronous systems

# Fault tolerance

- Hide the occurrence of failure from other processes using **redundancy**. Three types:
  1. Information redundancy
     - Add extra bits to allow for error detection/recovery
       - e.g. parity bits, cyclic redundancy checks (CRC)
  2. Physical redundancy
     - **Replicate** hardware and/or software
       - e.g. process replication
     - We will cover this in Module III
  3. Time redundancy
     - Retry an operation until a reply is received
       - e.g. remote method invocations

# Fault tolerance

- Time redundancy works well if the operation is **idempotent**
  - Can be performed repeatedly with the same effect as if it had been performed exactly once
    - Pure read operations: e.g. loading a static web page
    - Strict overwrite operations: e.g. update your billing address in an online shop
  - What if the operation is non-idempotent?
    - e.g. electronic transfer of money
    - This requires adding 'Duplicate filtering' and 'Retransmission of results'
    - The server must be stateful

# Fault tolerance

- Duplicate filtering
  - Client assigns a unique identifier to each request
  - Server filters out duplicate requests to avoid re-executing the operations
- Retransmission of results
  - Server keeps a history of prior results to resend lost replies without re-executing the operations
  - How to avoid the history to become huge?
    - If clients can make only one request at a time, server can interpret each request as an ACK of its prior reply
      - History must contain only the last reply message
    - Messages are also discarded after a period of time

# Challenges of distributed systems

9. Scalability
   - Ability to support the growing in the number resources/users, the distance between nodes, or the number of administrative domains

   A. Number of users and/or resources
      - **Size scalability**

   B. Maximum distance between nodes
      - **Geographical scalability**

   C. Number of administrative domains
      - **Administrative scalability**

# Scaling problems

A. Size scalability
   1. Centralized services and/or data
      - e.g. a single server providing a service becomes a bottleneck when the requests increase due to:
        - Limits on its computational capacity
        - Limits on its storage capacity and the I/O transfer rate between CPUs and disks
        - Limits on the network bandwidth with its users
   2. Centralized algorithms
      - e.g. optimal routing needs complete information about the load of all machines and lines. This information would overload the network

# Scaling problems

B. Geographical scalability
   1. Latency may easily prohibit synchronous client-server interactions in wide-area networks (WAN)
   2. Communication in WAN is less reliable and offers limited bandwidth
   3. WAN lack support for multipoint communication

C. Administrative scalability
   1. The various domains have to protect themselves against malicious attacks from each other
      - Components within a single domain can often be trusted by users within that domain

# Scaling techniques

1. Hide communication latencies
   - Use asynchronous communication
   - Helpful to achieve geographical scalability
   - ↓ Not every application can use this model
     - e.g. interactive applications
2. Partitioning and distribution
   - Split data and/or computations and spread them across multiple computers
     - e.g. move computations to clients (Java applets)
     - e.g. decentralized naming services (DNS)
     - e.g. decentralized information systems (WWW)

# Scaling techniques

3. Replication/caching
   - **Replicate** components across the system
     - e.g. replicated web servers and databases
     - e.g. mirrored web sites
     - e.g. file and web caches (in browsers and proxies)
   - Caching: decision is taken by client of a resource
   - Having multiple copies leads to inconsistencies
     - Strong consistency requires global synchronization on each modification, which precludes large-scale solutions
     - Need for synchronization is reduced if we can tolerate some inconsistencies, but this is application dependent
   - We will cover this in Module III

# Scaling techniques

4. Use decentralized algorithms
   - No machine has complete information about the system state
   - Decisions based only on machine local information
   - Failure of one machine does not ruin the algorithm
   - No implicit assumption that a global clock exists
   - e.g. routing in P2P systems

# Contents

- Definition of a distributed system
- Challenges of distributed systems
- **Architectures for distributed systems**

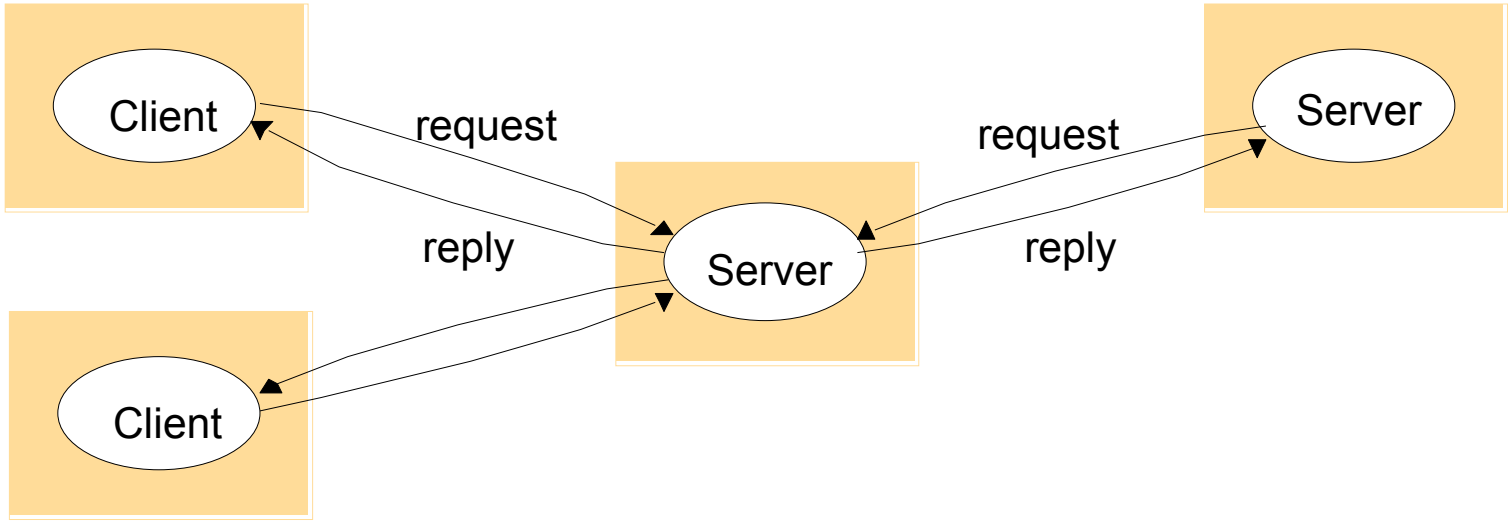
# Contents

- Definition of a distributed system
- Challenges of distributed systems
- **Architectures for distributed systems**
  - **System architectures**
  - Types of communication
  - Communication paradigms

# System architectures

## A. Client-server architectures

- Servers offer services and clients use them
- Based in the request-reply behavior
- Traditional (and successful) model until now
- Some scalability and robustness problems



# Client-server architectures

- Clients
  - Software combines user interfaces and solutions for achieving distribution transparency, e.g.:
    - Client-side stubs for remote method invocations
    - Invoke several replicas, pass a single response to client
    - Mask communication failures by retrying connections
- Servers
  - Sequential server serves one request at a time
    - Can service multiple requests by employing events and asynchronous communication
  - Concurrent server spawns a process or thread for each request (can also use a pre-spawned pool)

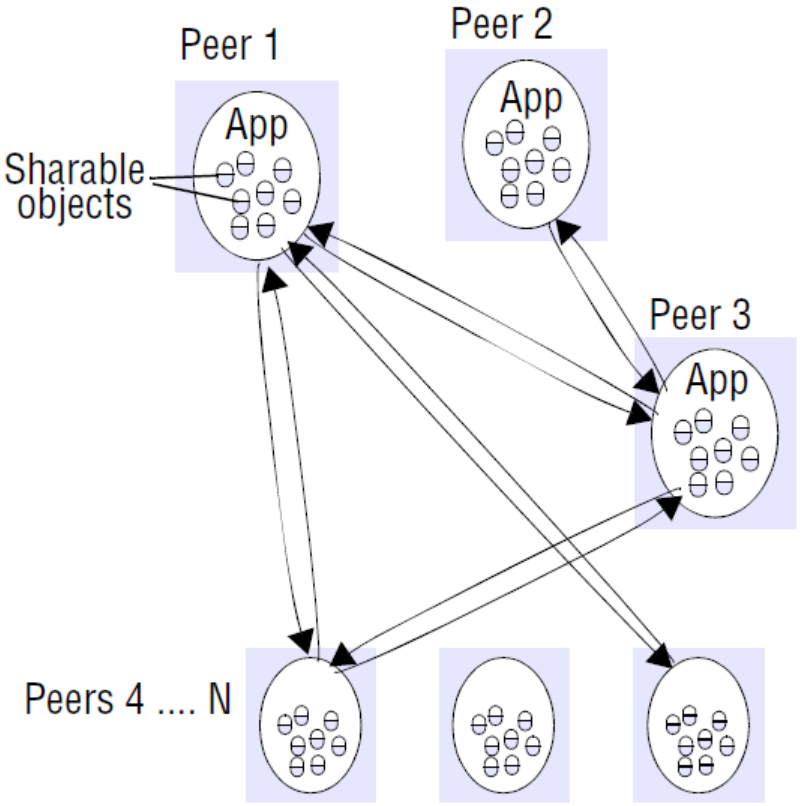
# Client-server architectures

- Servers
  - Stateless server does not keep data of its clients, and can change its state without informing clients
    - e.g. web server (but cookies can store client's info)
  - Stateful server keeps track of status of its clients
    - e.g. file server allowing client caching
      - Better client-perceived performance but complicates server recovery from a crash
  - Server can maintain a soft state for a limited time
- Servers can organize in tiered architectures
  - Servers may in turn be clients of other servers

# System architectures

## B. Peer-to-peer architectures

- Removes distinction between clients and servers
  - Nodes symmetric in function
- Overlay network
  - Data is routed over connections set up between the nodes
- No centralized control
- High availability & scalability
- Resilience to failures



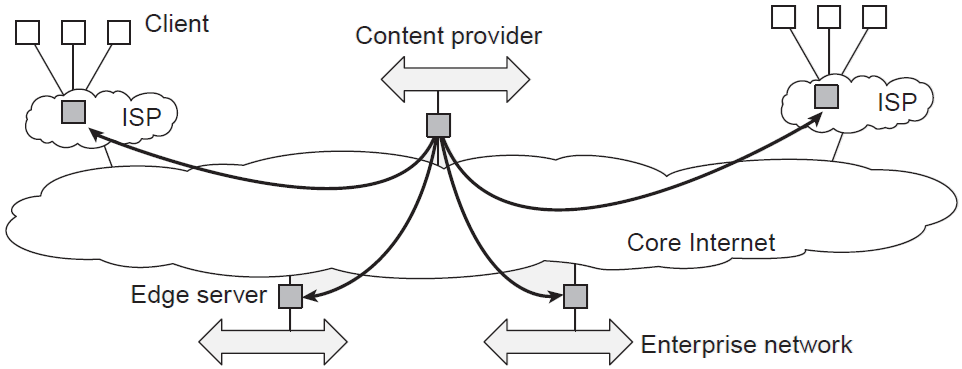
# Peer-to-peer architectures

1. Structured Peer-to-Peer architectures
   - Topology constructed deterministically
     - Nodes are organized following a specific distributed data structure and they are responsible for data items based only on their ID
     - Typically Distributed Hash Tables (DHT) (e.g. Chord)
2. Unstructured Peer-to-Peer architectures
   - Topology based on randomized algorithms
     - Each node picks a random set of nodes and becomes their neighbor and data items are assumed to be randomly placed on nodes
     - e.g. Gnutella

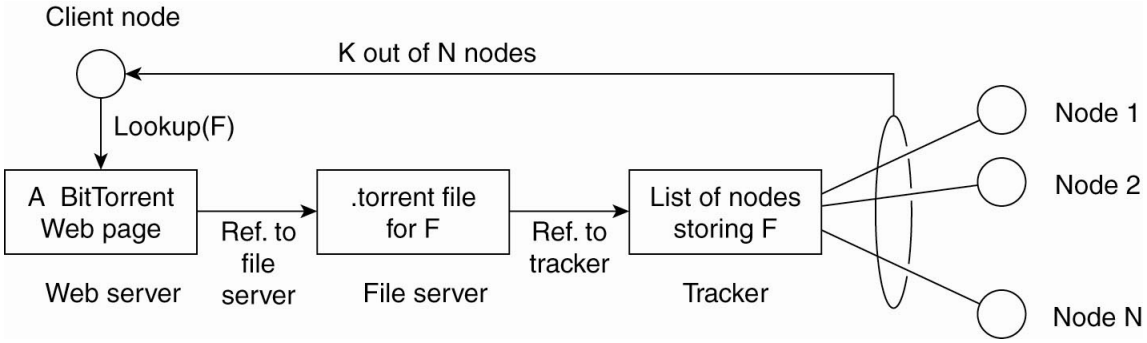
# System architectures

## C. Hybrid architectures

- Combination of client-server architectures with peer-to-peer solutions




e.g. edge-server systems (Content Distribution Networks)




e.g. collaborative systems (BitTorrent)

# Contents

- Definition of a distributed system
- Challenges of distributed systems
- **Architectures for distributed systems**
  - System architectures
  - **Types of communication**
  - Communication paradigms

# Types of communications

A. Direct communication
   - Senders explicitly direct messages/invocations to the associated receivers
   - Senders must know receivers identity and both must exist at same time
   - e.g. sockets, remote method invocations

B. Indirect communication
   - Communication through an intermediary with no direct coupling between senders and receivers
   - Indirect communication allows time and/or space **decoupling** between senders and receivers

# Types of communications

A. Space decoupling
   - Senders do not need to know who they are sending to
   - e.g. event-based systems (publish-subscribe)

B. Time decoupling
   - The sender and the receiver do not need to exist at the same time
   - e.g. e-mail
   - Time decoupling also allows distinguishing persistent from transient communications

# Types of communications

A. Persistent communications
   - The receiver does not need to be operational at the communication time
     - The message is **stored** at a communication server as long as it takes to deliver it at the receiver
   - e.g. e-mail

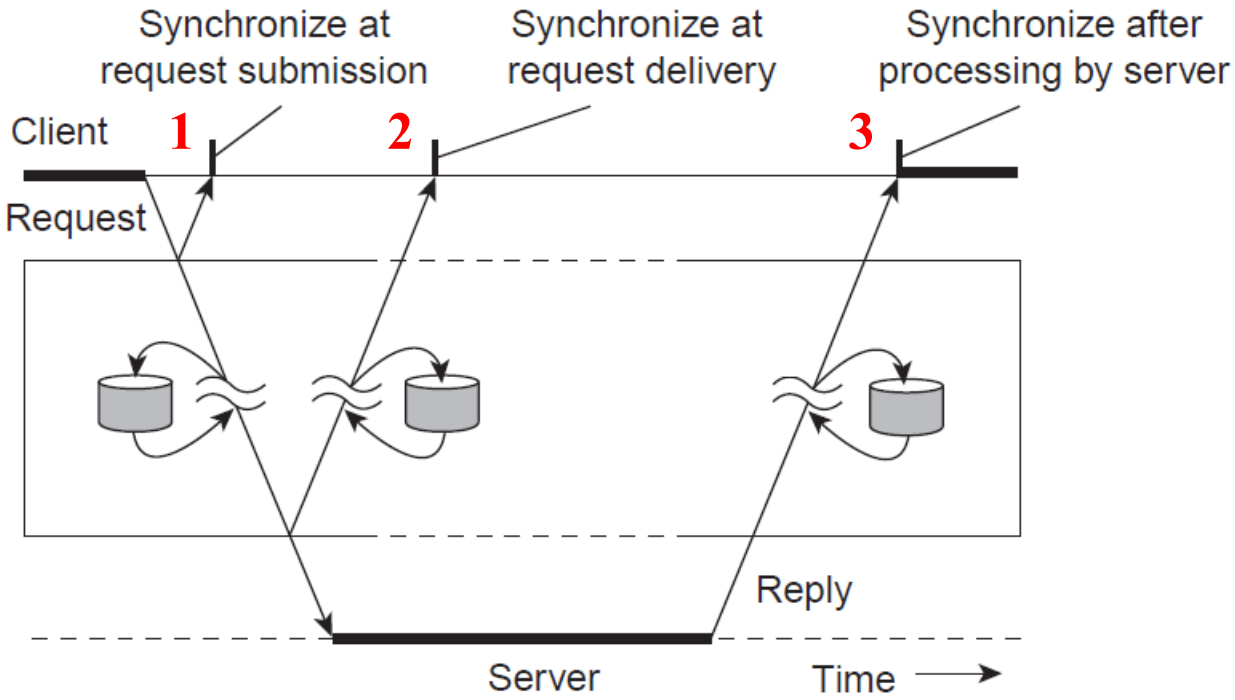
B. Transient communications
   - A message is **discarded** by a communication server as soon as it cannot be delivered at the next server, or at the receiver
   - e.g. sockets, remote method invocations

# Types of communications

A. Asynchronous communications
   - The sender **continues** with other work immediately upon sending a message to the receiver
   - e.g. publish-subscribe systems, e-mail

B. Synchronous communications
   - The sender **blocks, waiting** for a reply from the receiver, before doing any other work
   - This tends to be the default model for request-reply paradigms (e.g. RPC/RMI)

# Synchronous communications




1. **Submission-based:** Block until the middleware notifies that it will take over transmission of the request
2. **Delivery-based:** Block until request is delivered to recipient
3. **Response-based:** Block until recipient replies with response

# Types of communications

A. Discrete communications
   - Exchange of 'independent' units of information
   - Timing has no effect on correctness
   - e.g. e-mail, remote method invocations

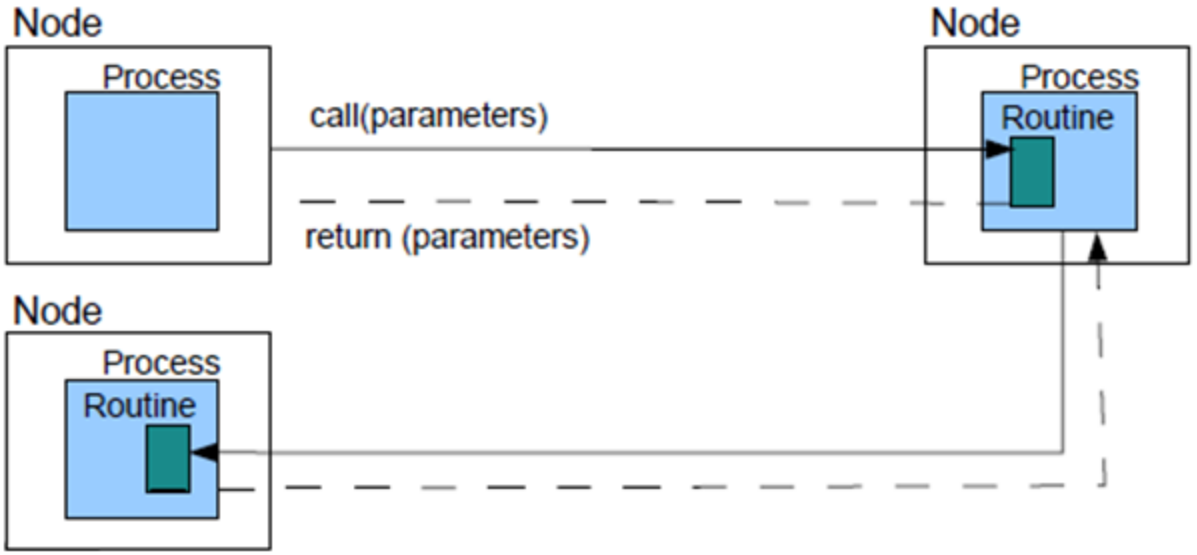
B. Continuous communications
   - Messages are related to each other by the order they are sent, or by a temporal relationship
   - Timing between data items must be preserved to interpret correctly the data
   - e.g. streams (audio, video, …)

# Contents

- Definition of a distributed system
- Challenges of distributed systems
- **Architectures for distributed systems**
  - System architectures
  - Types of communication
  - **Communication paradigms**

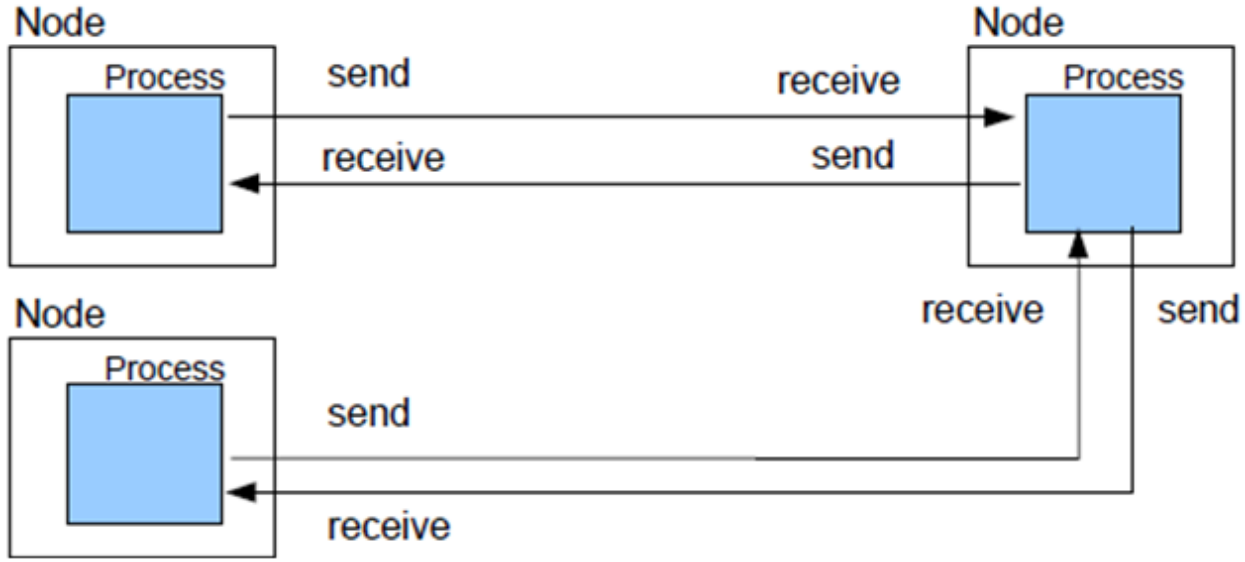
# Remote invocation

- Transparent extension to traditional programming: a node can call a function in another one as if it was local (e.g. RPC, RMI)
- Direct, transient, synchronous point-to-point interactions
- Middleware handles the marshaling/unmarshaling of parameters
- Protocol is sessionless and server is stateless about client
  - Function can change server's state, but client must maintain its state



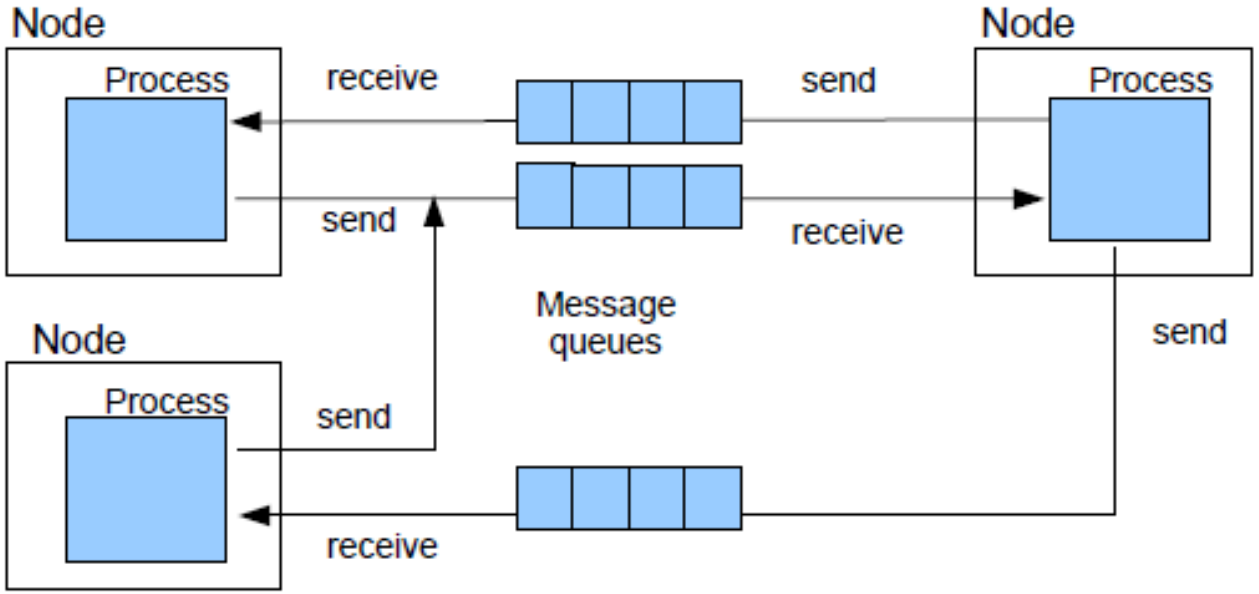
# Message passing

- Direct, transient networked communication between processes
  - e.g. sockets, MPI
- Generally synchronous and point-to-point
  - MPI supports also non-blocking and multipoint communication
- It is not middleware mediated
- Lacks transparency: exposes the network characteristics/issues



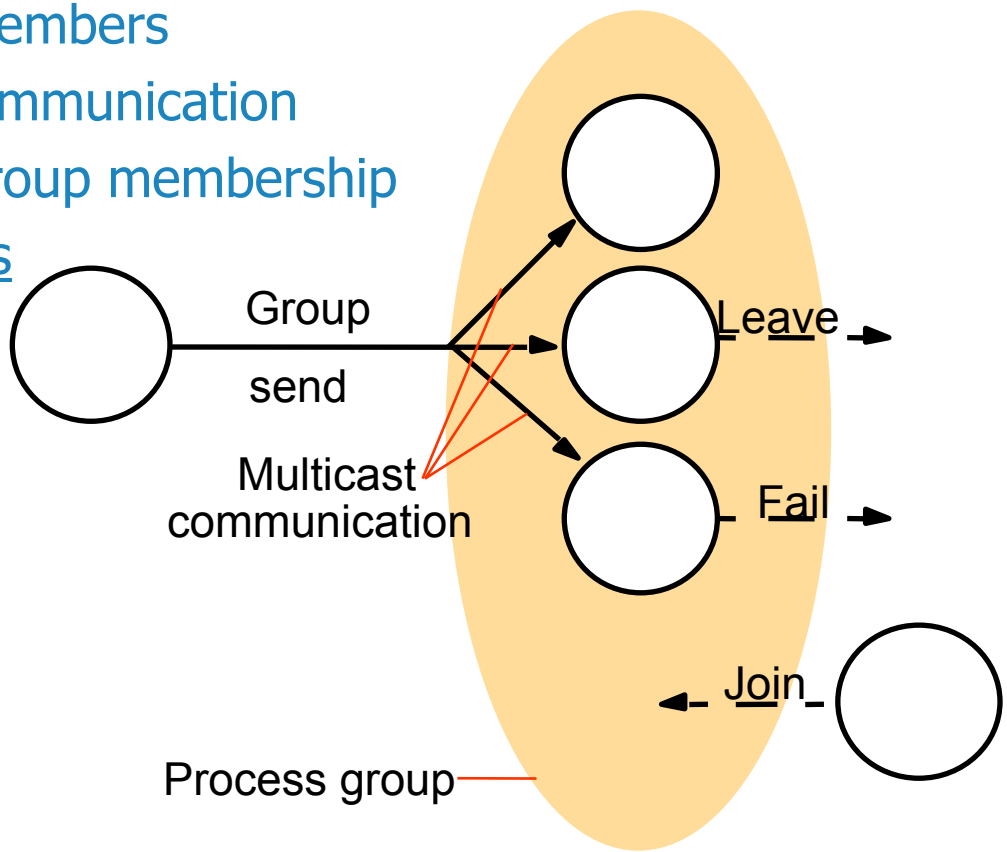
# Message queuing

- Sender puts message into a queue, receiver gets it from queue
  - e.g. e-mail, Message Oriented Middleware (Apache ActiveMQ*, RabbitMQ*)
- Persistent and asynchronous, by means of message queues
- Point-to-point (sender ⇔ queue ⇔ receiver)
- Middleware stores/forwards messages



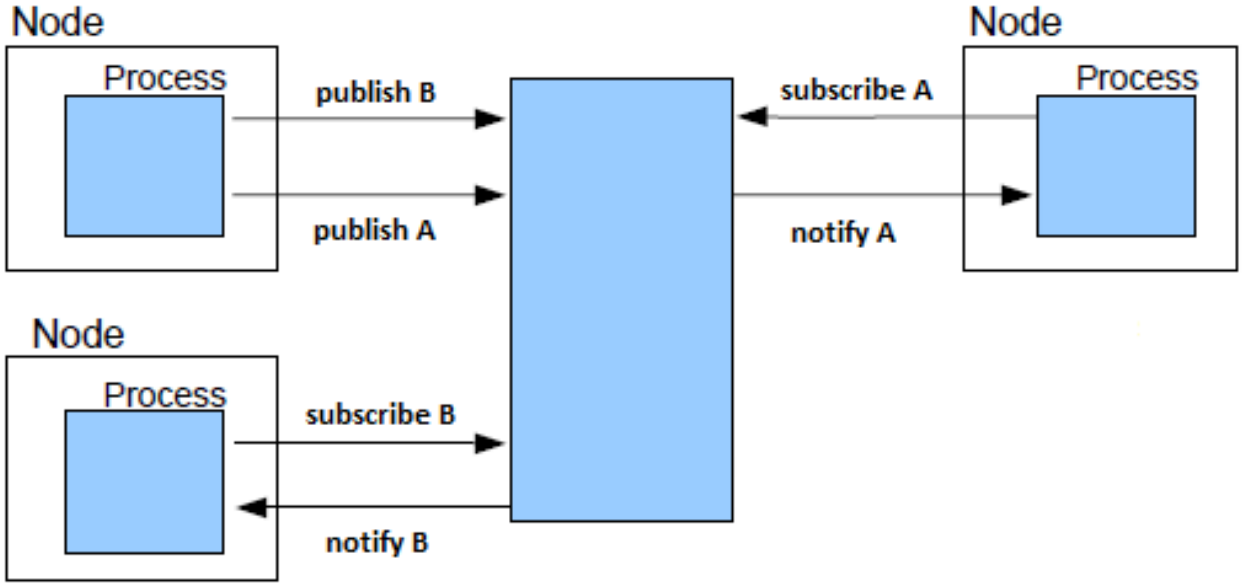
# Group communication

- Messages sent to a group via the group identifier: do not need to know the recipients (communication is space decoupled)
- Delivered to all group members
- One-to-many style of communication
- Middleware maintains group membership
- Transient & synchronous
- e.g. JGroups, Spread



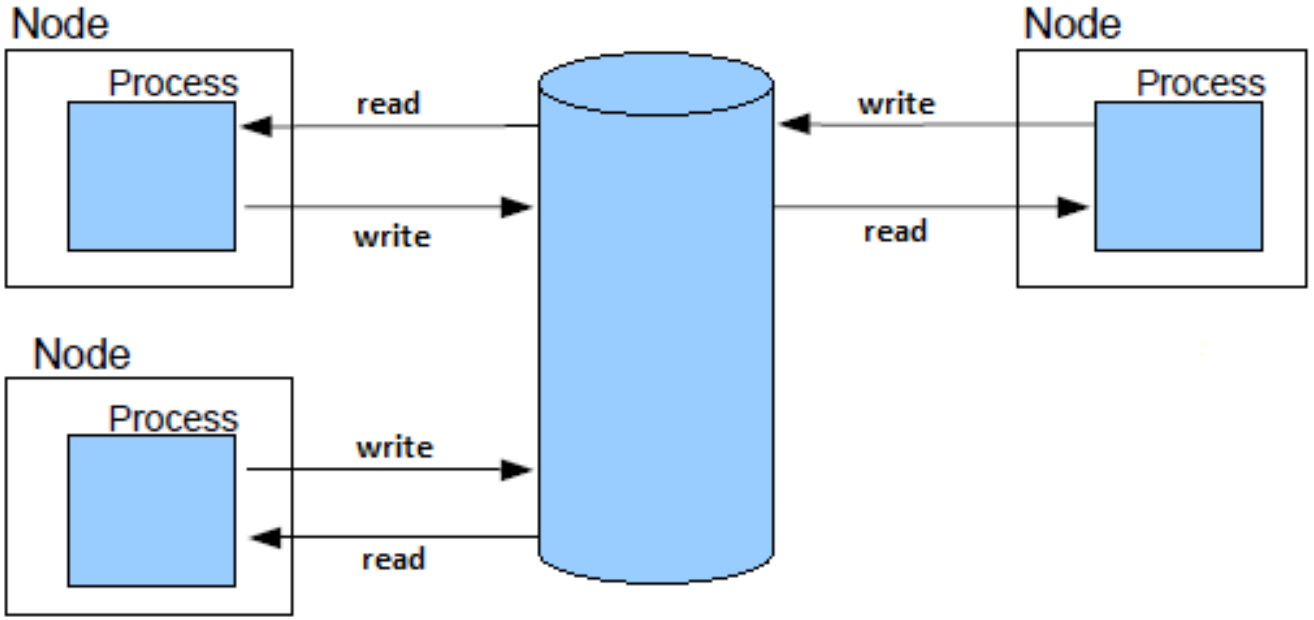
# Publish-subscribe

- Indirect, asynchronous communication by propagating events
  - Producers publish structured events, consumers express interest in events through subscriptions (e.g. Apache Kafka, Apache ActiveMQ*, Scribe)
- One-to-many style of communication (space decoupled)
- Middleware efficiently matches subscriptions against published events and ensures the correct delivery of event notifications



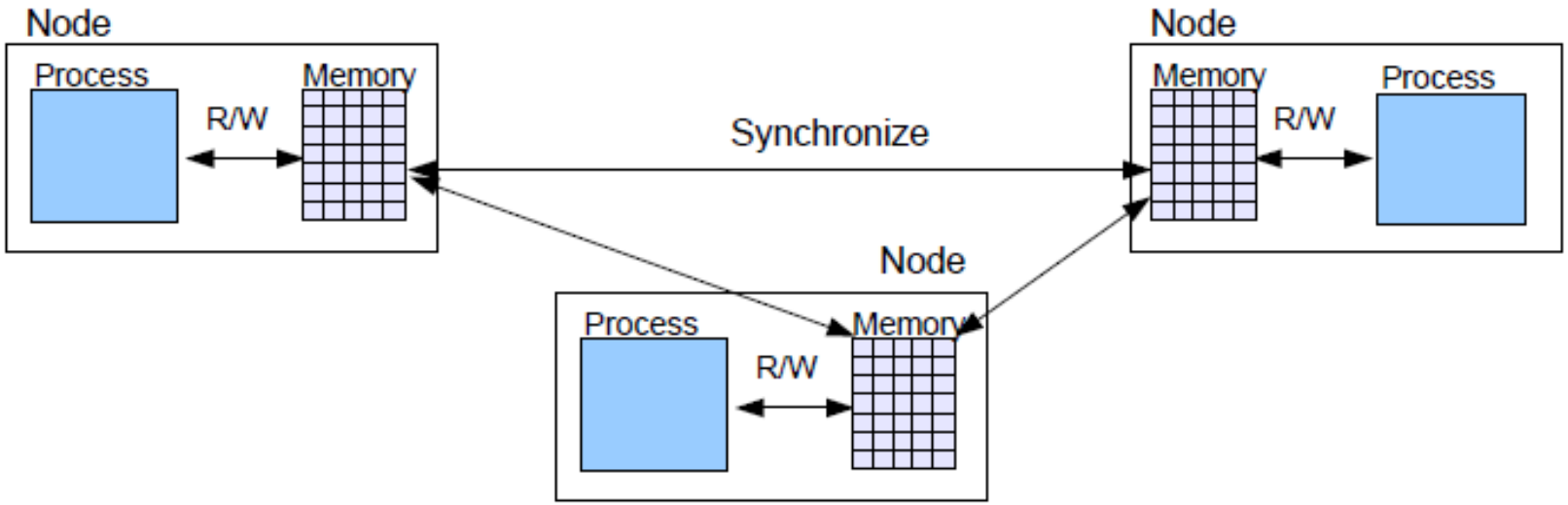
# Shared data space

- Persistent, asynchronous communication using a shared storage
  - e.g. JavaSpaces, which also provides space decoupling by means of pattern matching on contents
- Post items to shared space; consumers pick up at a later time
- One-to-many style of communication



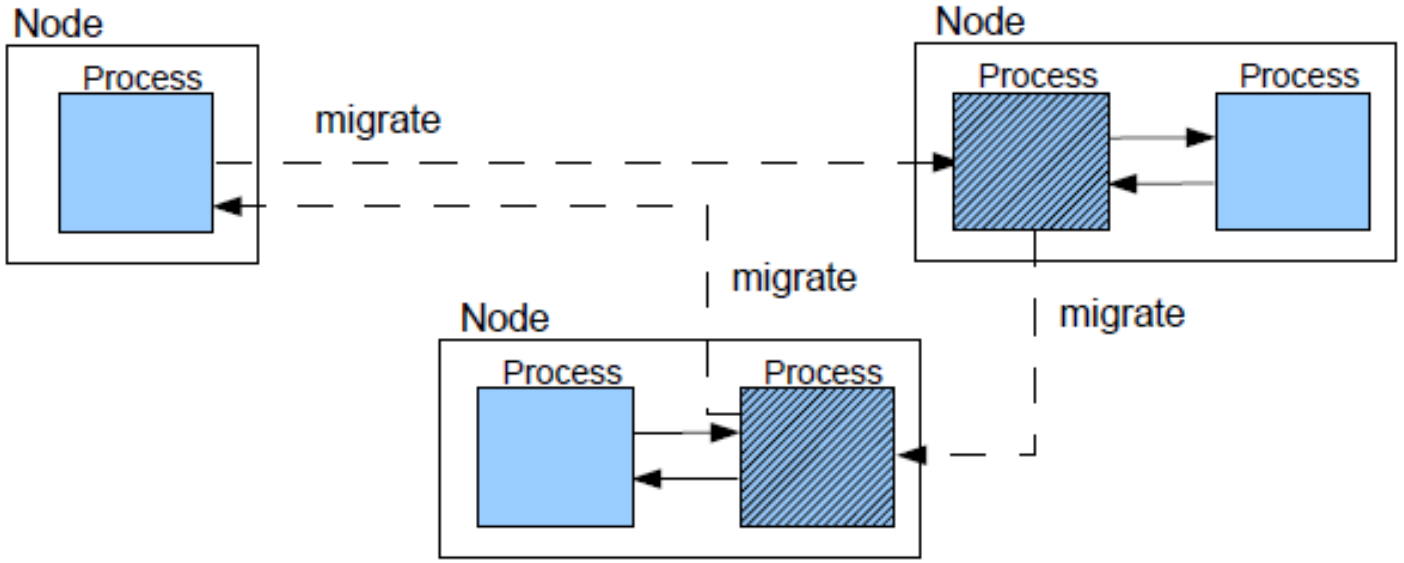
# Shared memory

- Share data between processes as if they were in their own local address spaces (extend traditional parallel programming model)
  - e.g. distributed shared memory (DSM): Treadmarks, Linda, Orca
- Indirect (space decoupled) and transient communication
- Interaction is multipoint (many components share memory)
- Middleware synchronizes and maintains the consistency of data



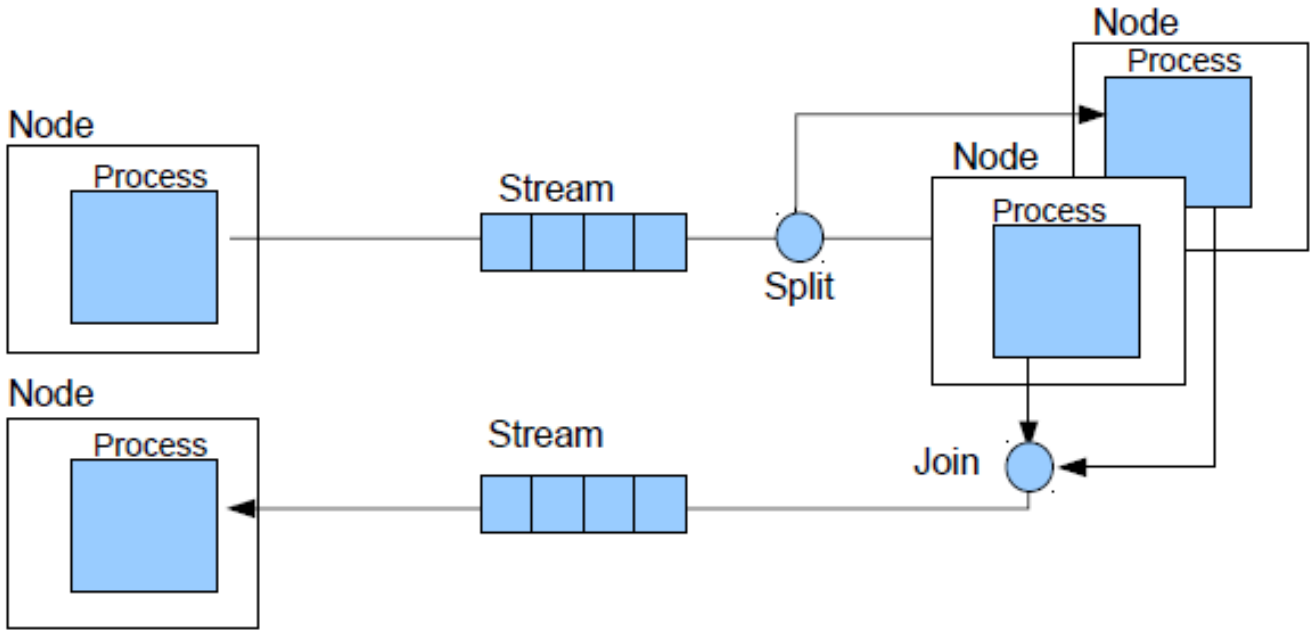
# Mobile code / agents

- Code or running processes travel from one node to another and interact locally with other components
  - e.g. code: web applets, JavaScript, Flash, ActiveX
  - e.g. agent: Java Agent Development Framework (JADE), VM migration
- Local interactions are faster, but it is a potential security threat
- Middleware transfers code and saves/restores process state



# Stream oriented

- Processing of large sequences of continuous data streams
  - e.g. distributed multimedia applications (video, audio), sensor data
- Direct, transient, multipoint communication
- Middleware ensures timing, coordinates flows (splits, joins) and handles issues such as congestion, delays, and failures



# Summary

- Definition/motivation for distributed systems
- Challenges: heterogeneity, no global view, security, coordination, asynchrony, openness, transparency, fault tolerance, scalability
- System architectures: client-server, peer-to-peer, hybrid
- Communication types: direct/indirect, space/time decoupling, persistent/transient, asynchronous/synchronous, discrete/continuous

# Summary

- Communication paradigms: remote invocation, message passing, message queuing, group communication, publish-subscribe, shared data spaces, shared memory, mobile code/agents, stream-oriented

# Contents

- Understand the *fundamental concepts* for building a distributed system from an algorithmic perspective

1. Concepts underlying distributed systems
2. Distributed algorithms
   A. Time and global states
   B. Coordination and agreement
3. Distributed shared data
   A. Distributed transactions
   B. Consistency and replication

# Bibliography

- Basic textbooks
  - A. S. Tanenbaum, M. van Steen. *Distributed Systems: Principles and Paradigms*, 2nd edition, Prentice Hall, 2007
  - G. Coulouris, J. Dollimore, T. Kindberg, G. Blair. *Distributed Systems: Concepts and Design*, 5th edition, Addison-Wesley, 2011
- Additional books
  - S. Ghost. *Distributed Systems: An Algorithmic Approach*, Second Edition, Chapman and Hall/CRC, 2014
  - F. Cesarini, S. Thompson. *Erlang Programming: A Concurrent Approach to Software Development*, O'Reilly, 2009
  - J. Armstrong. *Programming Erlang: Software for a Concurrent World*, 2nd edition, Pragmatic Programmers, 2013
  - F. Hebert. *Learn You Some Erlang for Great Good!*, No Starch Press, 2013