1. Concepts of Distributed Systems

Sistemes Distribuïts en Xarxa (SDX)
Facultat d'Informàtica de Barcelona (FIB)
Universitat Politècnica de Catalunya (UPC)
2024/2025 Q2

Contents

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

Definition of a distributed system

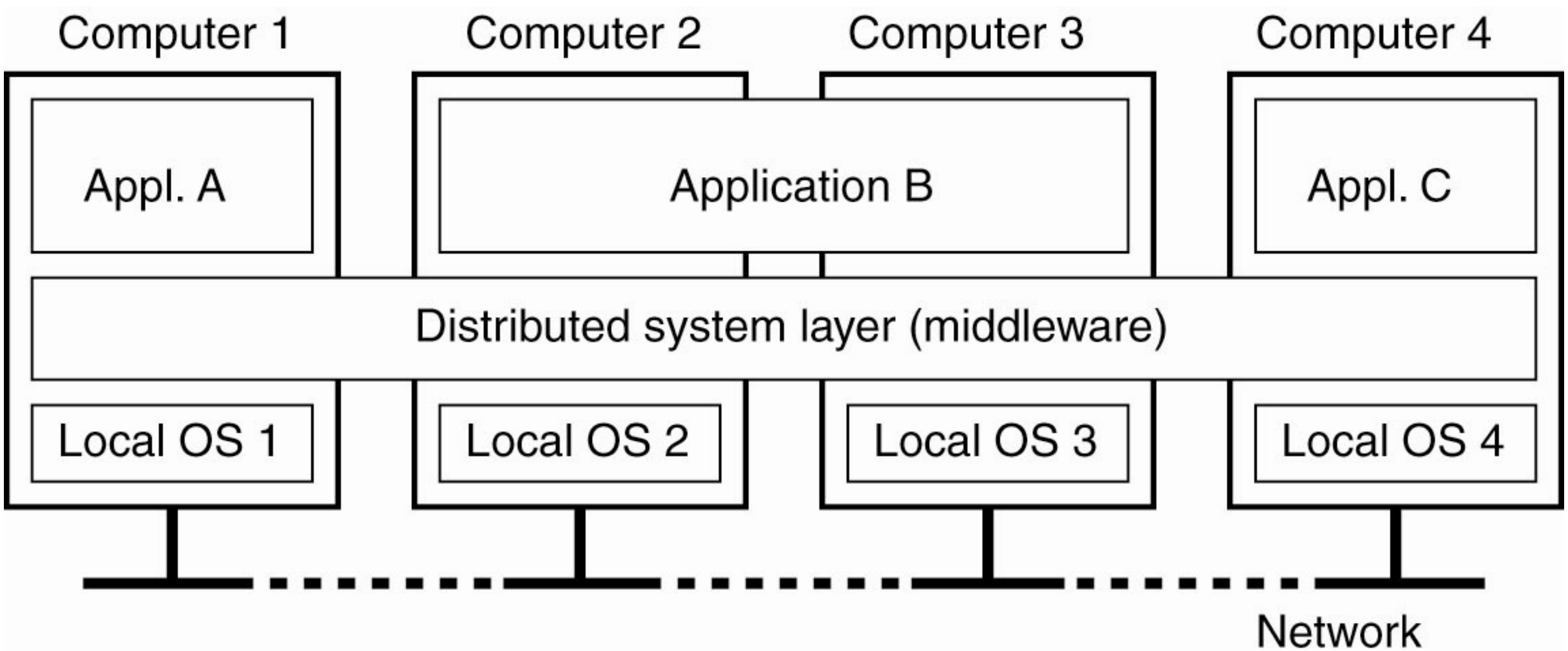
- It is one in which the failure of a computer you didn't even know existed can render your own computer unusable
  - [Lamport87]
- This apparently humorous definition entails the basic aspects of a distributed system:
  - The transparency of distribution ("a computer you didn't even know existed")
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



Definition of a distributed system

- Consequences of the definition:

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

Definition of a distributed system

- Consequences of the definition:

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

READING REPORT

[Hebert13] Hebert, F., *Fallacies of Distributed Computing*, Learn You Some Erlang for Great Good! Chapter 26: Distribunomicon, No Starch Press, pp. 445-451, January 2013

Definition of a distributed system

- Why do we build distributed systems?
  - Functional: computers have different functionality
    - e.g. clients vs. servers
  - Performance: work divided among computers
    - e.g. P2P, distributed computing systems (SETI@HOME)
  - Fault tolerance: e.g. replication
  - Accommodate a natural geographic distribution
    - e.g. weather stations, remote resources
  - Resource sharing: e.g. printer in a LAN
  - Economical: multiple low-cost small computers instead of an expensive big one

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 require a global view:

Challenges of distributed systems

a) Global time (we will cover this in Lesson 3)
   - 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 (out of the scope of this course)
   - Each process is independent
   - Perfect clock synchronization is not feasible

   i. 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 require coordination:

   a) Mutual exclusion (we will cover this in Lesson 4)
      - Processes coordinate their accesses to some shared resource (critical section) using solely message passing

   b) Leader election (we will cover this in Lesson 4)
      - 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 (we will cover this in Lesson 4)

- Agree on which messages a group of processes deliver 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 (out of the scope of this course)

- 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 in 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 the time intervals in 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

$\Rightarrow$ 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 primary
     - 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 <u>failure</u> is any deviation of the observed behavior of a system from its specifications
     - On a single machine, they often affect the **entire** system
     - On distributed systems, failures are **partial**
       - Only some components fail
   - A **fault tolerant system** is able to meet its specifications even with <u>partial failures</u>
     - Requires techniques for detecting, masking, tolerating, and recovering from 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 *Fail-stop crash* *Fail-silent crash*	Process halts and remains halted Other processes can reliably detect the failure Other processes may not be able to detect the failure
Omission failure *Receive omission* *Send omission*	Sent message never arrives at the other end Process fails to receive incoming messages Process fails to send outgoing messages
Timing failure	Process response lies outside a specified time interval
Response failure *Value failure* *State transition failure*	Process response is incorrect The value of the response is wrong The process deviates from the correct flow of control
Arbitrary (Byzantine) failure	Process may arbitrarily omit intended steps or take unintended steps, which exhibit differently to different observers

Fault tolerance

- Deal with failures using **redundancy**
  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/software components or data items
     - More details in Lesson 5
  3. Time redundancy
     - Repeat an operation several times
       - e.g. retry a remote operation until a reply is received
     - More details in Lesson 2

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 nodes and lines. Collecting 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. Inefficient multipoint communication in WAN

C. Administrative scalability
   1. The various domains have to protect themselves against malicious attacks from each other
      - Conversely, 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 (JavaScript)
      • 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. file and web caches (in browsers and proxies)
   - Having multiple replicas leads to <u>inconsistencies</u>
     - 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 Lesson 5

* Caching: decision is taken by the client of a resource (not the owner)

Scaling techniques

4. Decentralized algorithms
   - There is not any central node that coordinates the actions of the others or has complete information about the system state
   - Nodes cooperate with one another using only its local data to run the algorithm incrementally
   - Algorithm can make progress despite the failure of some nodes
   - No assumption about the presence of a global clock or the speed of processes/communication
   - e.g. routing in P2P systems (details in Lesson 9)

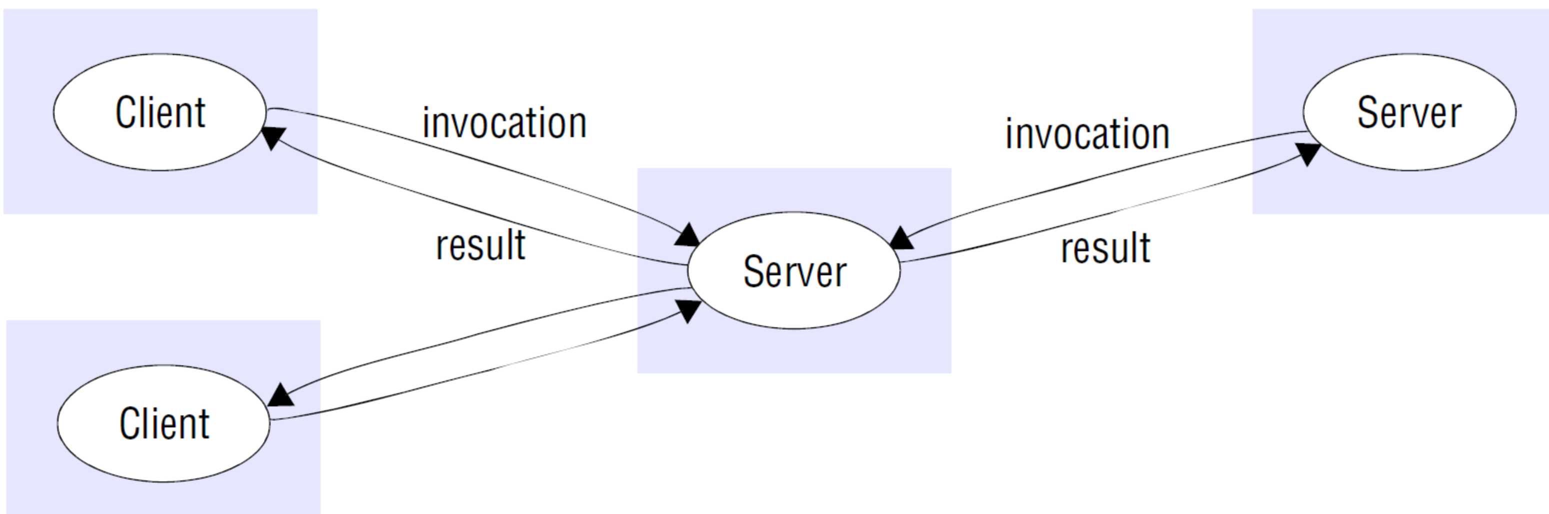
Contents

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

System architectures

A. Client-server architectures

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



Client-server architectures

- Clients
  - Middleware 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
  - Middleware runs processes that wait for incoming service requests at specific end points (i.e. ports)
    - Assign fixed ports to well-known services (FTP, SSH, …)
    - Dynamically assign ports to services: clients must look up the end points to find the specific servers

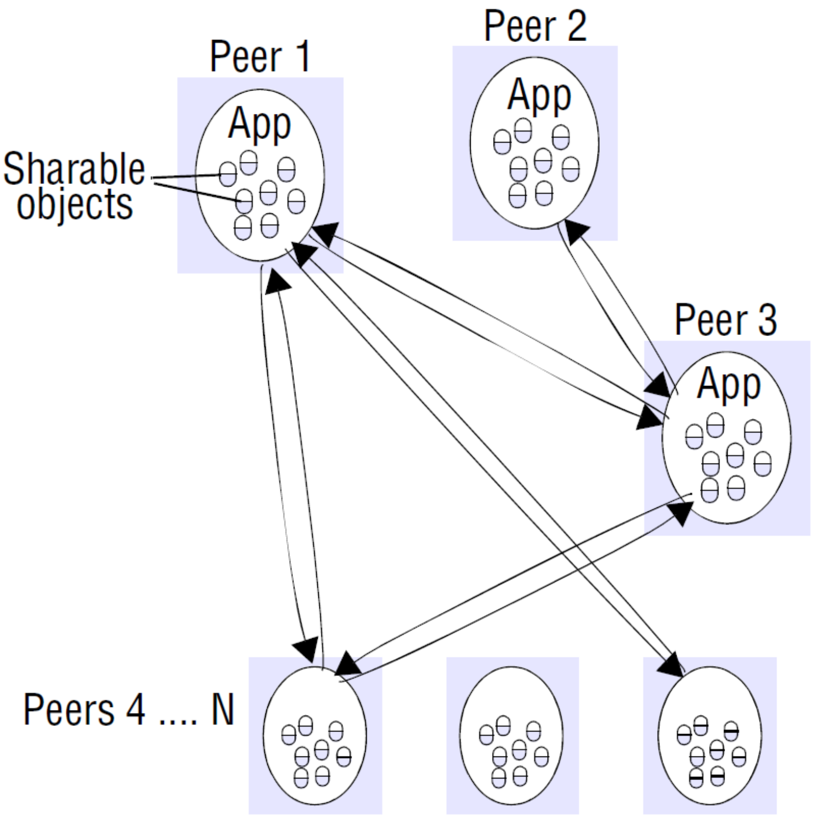
Client-server architectures

- 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's 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

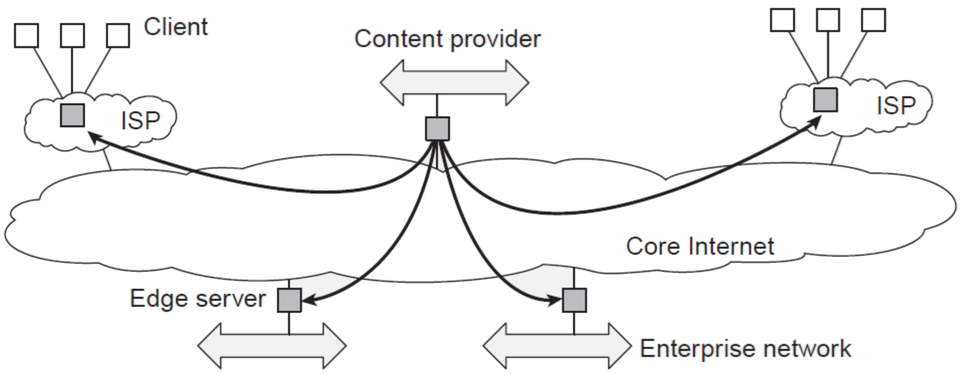
1. Structured Peer-to-Peer architectures
   - Topology constructed deterministically
     - Nodes are organized following a specific distributed data structure and they are responsible for data items based only on their ID
     - Typically Distributed Hash Tables (DHT) (e.g. Chord)
2. Unstructured Peer-to-Peer 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)

➢ We will cover them in Lesson 9

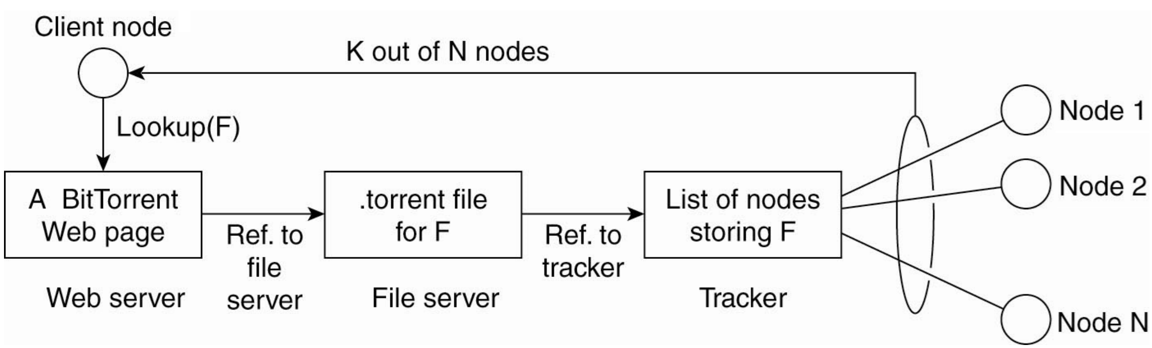
System architectures

C. Hybrid architectures

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



e.g. edge/fog systems
e.g. CDN (content distribution networks)
we will cover them in Lessons 8 and 11



e.g. collaborative systems (BitTorrent)
we will cover them in Lesson 9

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
- Further details:
  - [Tanenbaum]: chapters 1, 2.3, 2.4, 3.3, and 3.4
  - [Coulouris]: chapters 1 and 2