universidade de aveiro

deti

departamento de eletrónica, telecomunicações e informática

Distributed Systems

Introduction to Distributed Systems

Eurico Pedrosa <efp@ua.pt>

2^{nd} semester - $2025'26$

Just because it is possible to build distributed systems does not necessarily mean that it is a good idea.

— Steen and Tanenbaum 2023

Networked Computer System

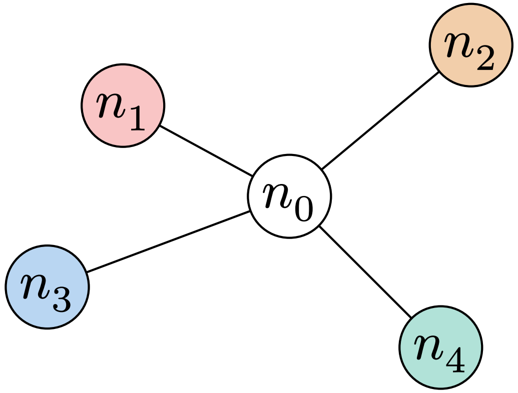
A **Networked Computer System** is:

- a **collection of autonomous computers**
- **interconnected by a communication network**(wired, wireless, or both)
- that can **exchange data and access services remotely**
- with **no shared memory and no global clock**
- often **highly dynamic**, as nodes may join or leave and network conditions change

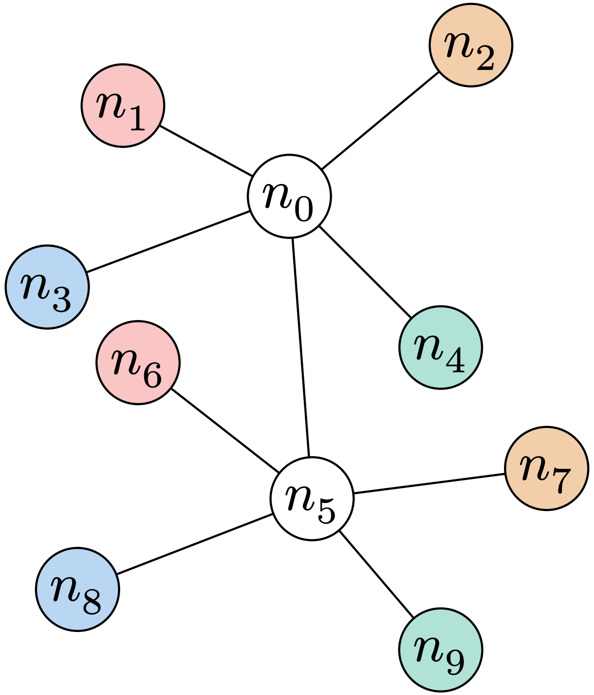
Such systems may range from **a few devices to millions of machines**, and form the **foundation on which *distributed* and *decentralized* systems are built**.

Decentralized vs Distributed Systems

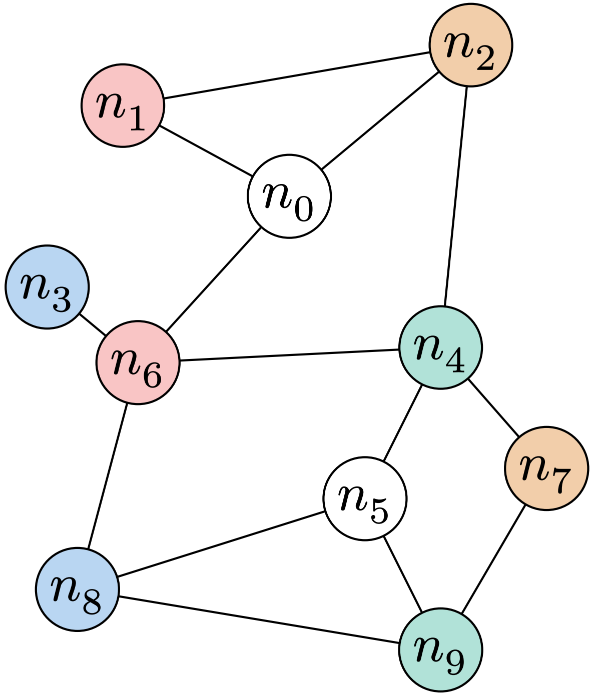
Centralized



Decentralized



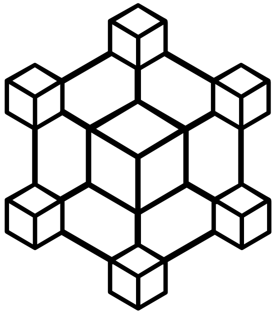
Distributed



These organizations are *NOT* that meaningful

Decentralized vs Distributed Systems

- A *decentralized system* is a networked computer system in which proce spread across multiple computers.
  - It **integrates** existing networked systems to share services and resources



Example: Distributed Ledger (blockchain)

- **necessarily** spread across participants due to lack of trust
- transactions validated by the participants

Decentralized vs Distributed Systems

- A *distributed system* is a networked computer system in which processes and resources are **sufficiently** spread across multiple computer
  - It **expands** systems with additional computers to improve scalability, availability, and reach

Example: Google Mail

- single point of entry – `{mail,imap,smpt}.gmail.com`
- **sufficiently** spread across millions computers
  - a computers does not process all e-email

Decentralized vs Distributed Systems

- **Centralized solutions are often simpler**
  - Centralized systems are usually easier to design and manage
- **Decentralization introduces inherent complexity**
  - Spreading components introduces hidden dependencies.
- **Partial failures are unavoidable**
  - Components fail independently and must be detected and handled
- **Systems are highly dynamic**
  - Nodes and resources can join or leave at any time.
- **Necessity vs sufficiency**
  - Decentralization may be required; distribution should be minimal

Design Goals of a Distributed System

- A **distributed system** should:
  - **Make resources easily accessible**
    - users access shared files or services from anywhere
  - **Hide distribution, be transparent**
    - applications use remote resources as if they were local
  - **Be open**
    - components interoperate via standard interfaces and protocols
  - **Be dependable**
    - provide reliable and available services, even in the presence of faults
  - **Be scalable**
    - the system grows by adding nodes, not by redesign

Resource Sharing

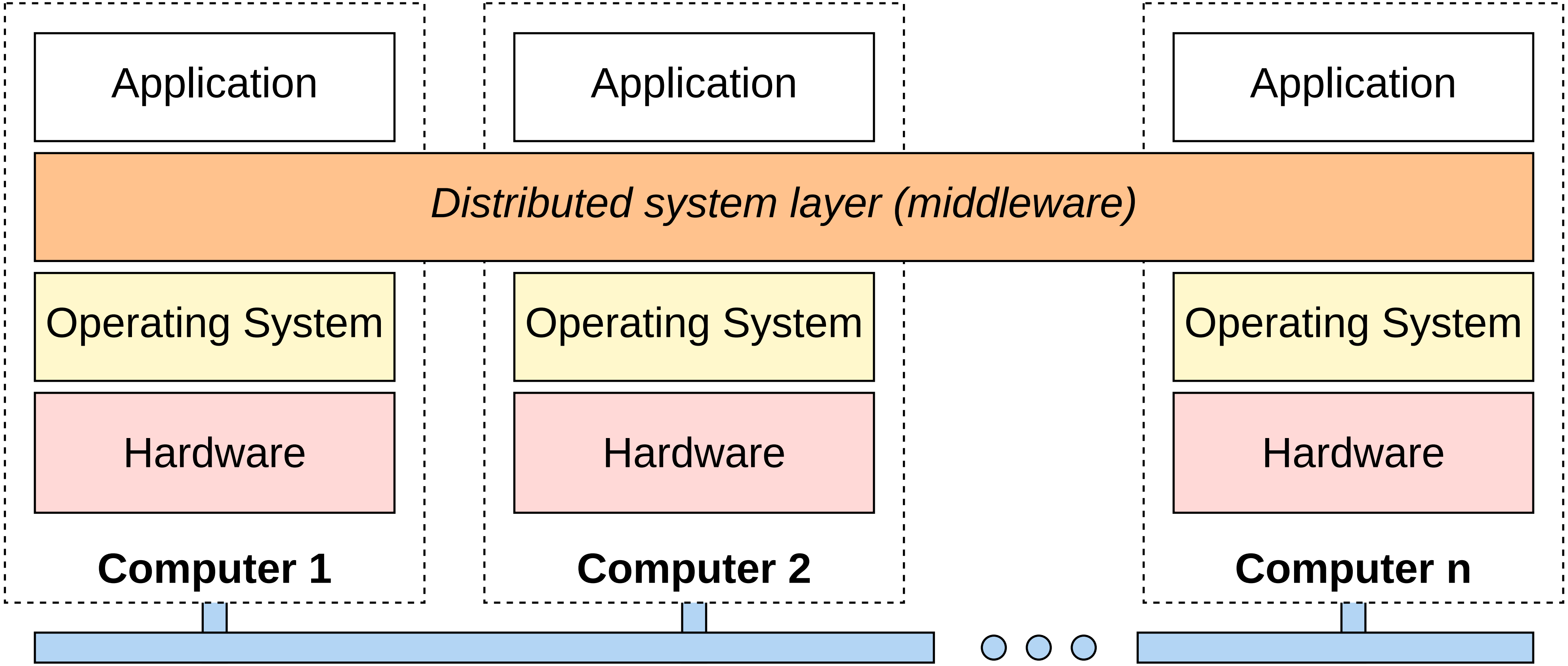
- **Primary goal of distributed systems**
  - Enable users and applications to **share resources** efficiently.
- **Resources include**
  - hardware, data, and services.
- **Sharing avoids duplication**
  - Expensive or scarce resources do not need to be locally available.
- **Network as the enabler**
  - Resources are accessed remotely via a network, often transparently.

Resource Sharing: Key Implications

- **Location transparency**
  - Users access resources without knowing where they are located
- **Concurrency is unavoidable**
  - Multiple users may access the same resource simultaneously
- **Access control is required**
  - Not all users or applications should have the same rights
- **Basis for higher-level services**
  - File systems, databases, and web services build on shared resources

Distribution Transparency

- The system **hides the fact** that resources are distributed across multiple computers.
  - distribution transparency through a **middleware layer**
- Users/apps interact with the system **as if it were a single, non-distributed system**.
- Full transparency is **impossible**
  - always involves **trade-offs**, especially with performance and scalability.


Application
Application
Application
Distributed system layer (middleware)
Operating System
Operating System
Operating System
Hardware
Hardware
Hardware
Computer 1
Computer 2
Computer n

Types of Distribution Transparency

Transparency	Description
Access	Hide differences in how data is represented and accessed
Location	Hide where an object is located
Relocation	Hide that an object may move while it is being used
Migration	Hide that an object may move to another location
Replication	Hide that an object is replicated
Concurrency	Hide that an object may be shared by multiple independent users
Failure	Hide the failure and recovery of an object

Openness in Distributed Systems

An **open distributed system** is a system that:

- Offers components that can **easily be used by or integrated into other systems**
- Is itself often built from **components originating elsewhere**

Openness enables:

- Integration of **heterogeneous components**
- Cooperation between **independently developed systems**
- Evolution of systems over time without redesigning everything

Openness in Distributed Systems

Three Core Concepts

1. **Interoperability**
   - Ability of **independently developed systems or components** to:
     - Co-exist
     - Work together
   - Achieved by relying on **common interface specifications**
     - Interface Definition Languages (IDLs) specify:
       - Function names
       - Parameters and return types
       - Exceptions
     - IDLs usually specify **syntax**, not full **semantics**

2. **Composability**
   - Open systems allow building applications by **assembling components**
   - Components may come from:
     - Different vendors
     - Different development teams

3. **Extensibility**
   - Easy to:
     - Add new components
     - Replace existing ones
   - Without affecting unrelated components
   - Examples:
     - Browser plug-ins, Website extensions (e.g., CMS plug-ins)

Openness in Distributed Systems

Separating Policy from Mechanism

- **Mechanism**: what the system can do
  - e.g. web caching
- **Policy**: how it should behave
  - e.g user can decide what to cache and for how long
- **Openness** requires:
  - Mechanisms to be fixed
  - Policies to be adaptable or replaceable

Openness in Practice

Reality Check

- Perfect openness is **hard to achieve**
- Integration often requires:
  - Significant effort
  - Workarounds

Open Source as an Extreme Form

- Full access to implementation
- High transparency
- Not always the best engineering solution?!

Openness and Middleware

- **Middleware** helps achieve openness by:
  - Hiding heterogeneity
  - Providing uniform interfaces
- Design patterns:
  - **Wrappers / Adapters**
  - **Interceptors**

Dependability in Distributed Systems

Dependability is a **fundamental design goal** of distributed systems

- It addresses the fact that:
  - Distributed systems are **subject to partial failures**
    - Components may fail independently
    - Operates over unreliable networks
- In a **partial failure**
  - Others continue to operate
  - The system does not completely stop
- **Failures** should be **masked**
  - And recover if possible
  - Usually refered to as being **fault tolerant**

Dependability in Distributed Systems

A **dependable system** is a system that provides:

- **Availability**
  - It remains accessible and ready to provide its services when requested
- **Reliability**
  - It continues to operate correctly over time, producing correct results
- **Safety**
  - It avoids reaching incorrect or harmful states, even when failures occur
- **Maintainability**
  - It can be repaired, adapted, or updated efficiently when problems occur

Dependability in Distributed Systems

Faults, Errors, Failures

- **Fault**: the cause of an error
  - e.g. A **communication link becomes unreliable** (messages can be lost or delayed)
- **Error**: a system state that may lead to failure
  - e.g. Due to message loss, a **remote procedure call does not receive a reply**, leaving the client in an incorrect state (unsure whether the operation executed)
- **Failure**: deviation from the system's specification
  - e.g. The client **assumes the remote operation failed and retries**, causing the operation to be executed twice, violating the system's intended behavior

Dependability in Distributed Systems

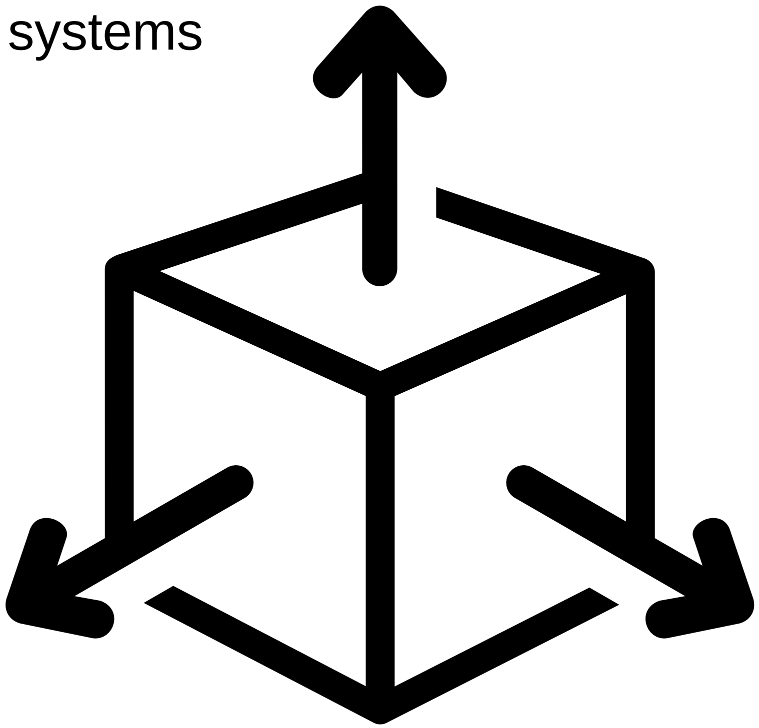
Fault Tolerance: A system is **fault tolerant** if it continues to provide services according to its specifications despite the presence of faults

- Faults are generally classified as
  - **Transient**: short-lived, disappears on its own
    - e.g. a message lost due to temporary network congestion
  - **Intermittent**: appears and disappears unpredictably
    - e.g. a network link that occasionally delays or loses messages
  - **Permanent**: persists until repair or replacement
    - e.g. a crashed server that stops responding

Scalability in Distributed Systems

Scalability is a **fundamental design goal** of distributed systems

- It concerns the system's ability to:
  - Cope with growth
  - Without unacceptable loss of performance
  - Or major redesign
- A system is scalable if:
  - Its performance remains acceptable
  - When the system size increases
  - By **adding resources**, not redesigning the system



Dimensions of Scalability

Scalability problems arise along three axes:

1. **Size scalability**
   - Supports more users and resources without noticeable performance loss
2. **Geographical scalability**
   - Remains usable despite large physical distances and communication delays
3. **Administrative scalability**
   - Remains manageable across multiple independent organizations

Dimensions of Scalability: Size

- As the **number of requests increases**, a server (or group of servers) becomes **limited** by:
  - **Computational capacity**
    - e.g. A centralized server cannot process all incoming requests fast enough
  - **Storage capacity (I/O throughput)**
    - e.g. A file or database server becomes overloaded by concurrent reads and writes
  - **Network capacity**
    - e.g. Network links to a centralized service become saturated as requests increase
- Can be addressed by
  - **Scaling up**: upgrade CPU, add more memory, better nerwork interface
  - **Scaling out**: deploy more machines

Dimensions of Scalability: Geographical

- System components are spread across
  - Wide-area networks
  - Large physical distances
- Main challenges:
  - Networks reliability
  - Latency, Bandwidth limitations

Geographical Scalability is Hard

Communication latency and bandwidth limitations become unavoidable over large distances and cannot be fully hidden

Dimensions of Scalability: Geographical

Hiding communication latencies

- Avoid blocking on remote communication.
- Overlap communication with useful computation
- Use asynchronous interaction instead of waiting for replies

Distribution

- Split services and data across multiple locations
- Place components closer to users to reduce long-distance communication

Dimensions of Scalability: Geographical

Replication

- Maintain multiple copies of data or services at different locations.
- Reduces access latency and improves availability for remote users.

Caching

- Store frequently accessed data closer to where it is used.
- Reduces repeated long-distance communication.

Dimensions of Scalability: Administrative

- The system spans:
  - Multiple organizations
  - Multiple administrative domains
- Challenges include:
  - Trust
  - Policy differences
  - Management autonomy

Administrative scalability is achieved by decentralizing control, respecting organizational autonomy, and relying on standard interfaces

Dimensions of Scalability: Administrative

Decentralization of control

- Avoid global centralized control components.
- Each administrative domain retains autonomy over its own resources.

Separation of administrative domains

- Organizations with local
  - Policies
  - Management
  - Security decisions

Dimensions of Scalability: Administrative

Use of standard interfaces and protocols

- Interaction between domains relies on:
  - Well-defined interfaces
  - Agreed-upon standards
- Avoids tight coupling between organizations.

Limited global assumptions

- Do not assume:
  - Uniform security policies
  - Global trust, Centralized management
- Accept heterogeneity as a design constraint.

Classification of Distributed Systems

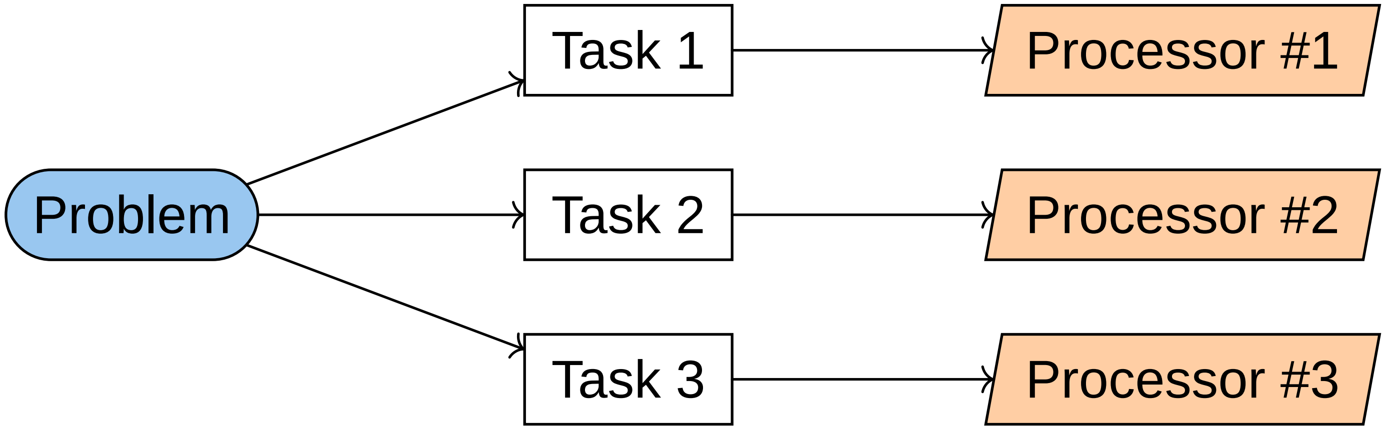
Distributed systems can be broadly classified into three categories:

- **High-performance distributed computing systems**
  - Combine multiple computers to execute compute-intensive tasks efficiently
  - e.g., clusters, grids, clouds
- **Distributed information systems**
  - Integrate applications and databases to support organizational information processing
  - e.g. transaction processing, enterprise systems
- **Pervasive systems**
  - Composed of many small, often mobile components operating in dynamic environments

High-Performance Distributed Computing Systems

High-performance distributed computing systems are designed to **execute compute-intensive applications efficiently** by combining the processing power of multiple computers.

They originate from **parallel computing** and focus primarily on performance rather than transparency or ease of use.



High-Performance Distributed Computing Systems

Cluster Computing

- A **cluster** is a collection of computers:
  - Usually **homogeneous**
  - Connected by a **high-speed local network**
- The system is managed as a **single computing resource**

Key characteristics

- Nodes run similar hardware and operating systems
- Centralized management and scheduling
- Optimized for performance and low latency

High-Performance Distributed Computing Systems

Grid Computing

- **Grid computing** connects resources from:
  - Different organizations and administrative domains
- These resources form a **virtual organization**

Key characteristics

- Highly **heterogeneous**:
  - Hardware, Operating systems, Networks
- Strong focus on:
  - Resource sharing, Access control
  - Coordination across domains

High-Performance Distributed Computing Systems

Cluster vs. Grid (Key Contrast)

- **Clusters**
  - Single administrative domain
  - Homogeneous resources
  - Easier management
- **Grids**
  - Multiple administrative domains
  - Heterogeneous resources
  - Emphasis on federation and policy

Distributed Information Systems

Distributed information systems arise in organizations that need to **integrate many networked applications and databases**.

Their main goal is **information processing and integration**, rather than raw computational performance.

Distributed Information Systems

Enterprise Application Integration (EAI)

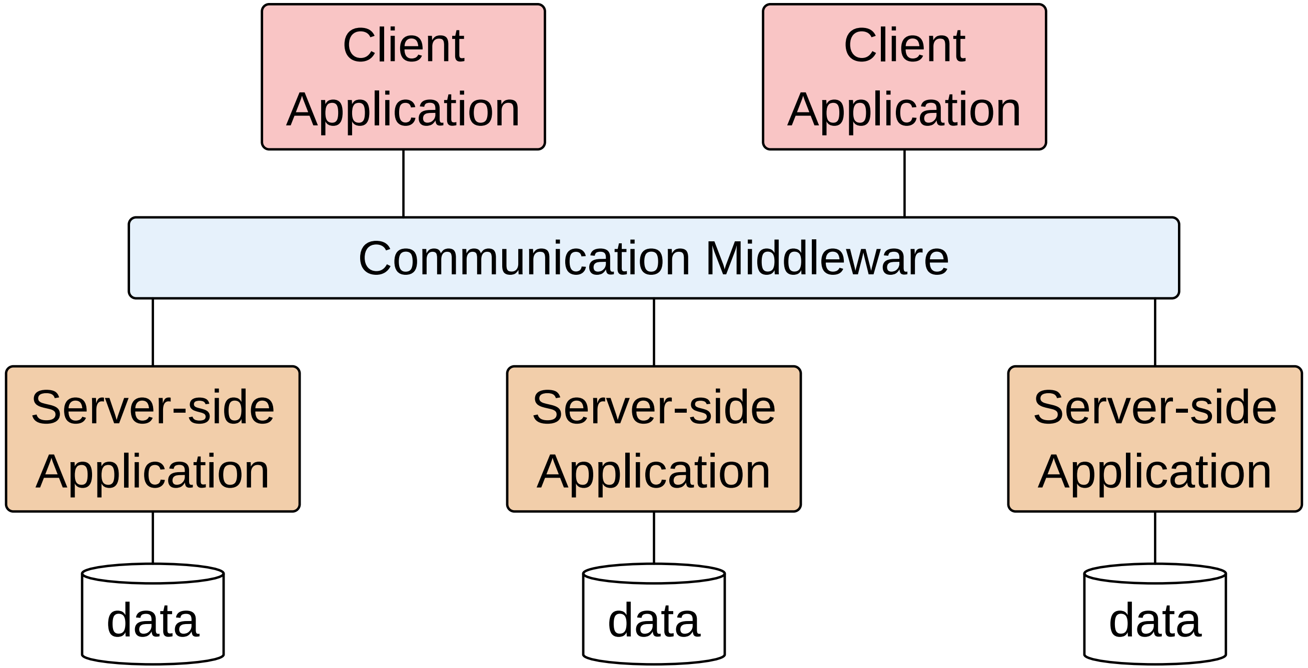
- Applications communicate **directly with each other**
- Integration happens at the **application level**, not only through databases
- Emerged as applications:
  - Became more complex
  - Were decomposed into independent components

Key aspects

- Interoperability between applications
- Communication through middleware
- Avoids tight coupling between systems

Distributed Information Systems

Middleware in Enterprise Application Integration (EAI)



Distributed Information Systems

Types of communication middleware support interaction between distributed components:

- **RPC**: invokes remote operations as if they were local calls.
- **RMI**: object-oriented version of RPC for remote method calls.
- **RPC/RMI limitations**:
  - both sides must be running
  - tight coupling and location awareness
- **Message-Oriented Middleware (MOM)**:
  - components exchange messages via logical channels
  - supports decoupled publish–subscribe communication

Distributed Information Systems

Distributed Transaction Processing

- Applications are structured as:
  - Clients issuing requests
  - Servers executing operations, often involving databases
- Multiple operations may be combined into a **single distributed transaction**

Key idea

- A transaction guarantees that:
  - **All operations execute**, or
  - **None of them do**
- Transactions adhere to the **ACID** properties

Pervasive Systems

- **Pervasive systems** consist of a **large number of small devices**
- Devices are often:
  - **Embedded**
  - **Mobile**
  - **Resource-constrained**
- The goal is to **blend computation naturally into the environment**
- Examples of devices include:
  - Sensors and actuators
  - Smartphones and wearable devices

Pervasive Systems

Key characteristics

- Highly **dynamic**:
  - Devices may frequently join and leave
- Often **context-aware**:
  - React to location, environment, or user activity
- Limited resources: (CPU, memory, energy, and network capacity)

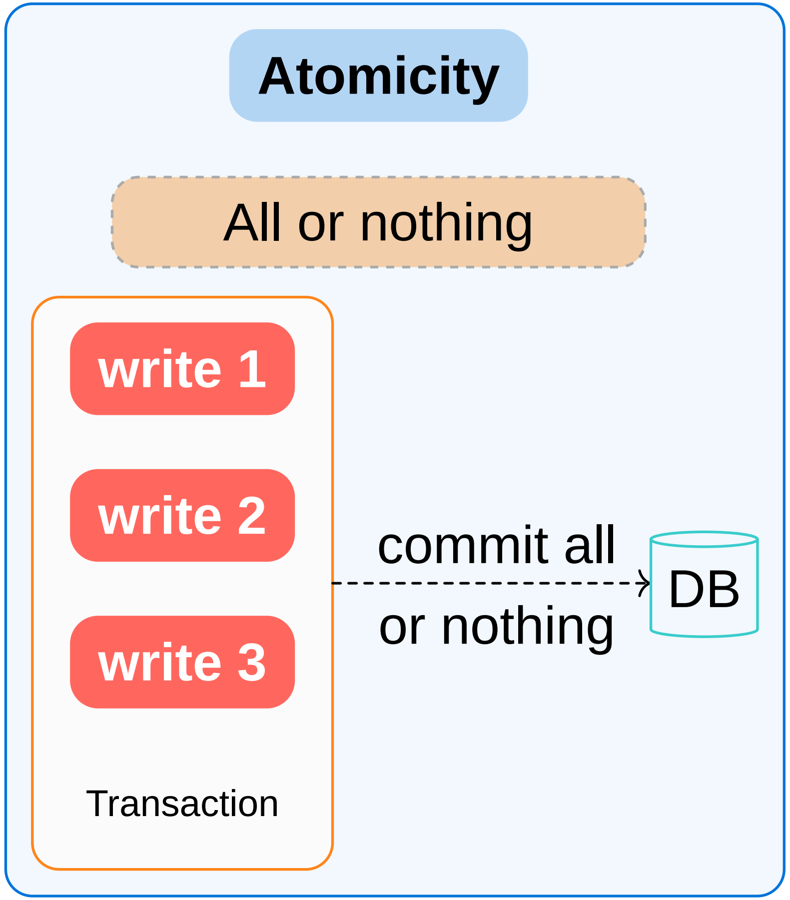
Main challenges

- **Scalability** with many devices
- **Unreliable communication**
- **Autonomous operation** with little or no centralized control

ACID Properties

Atomicity

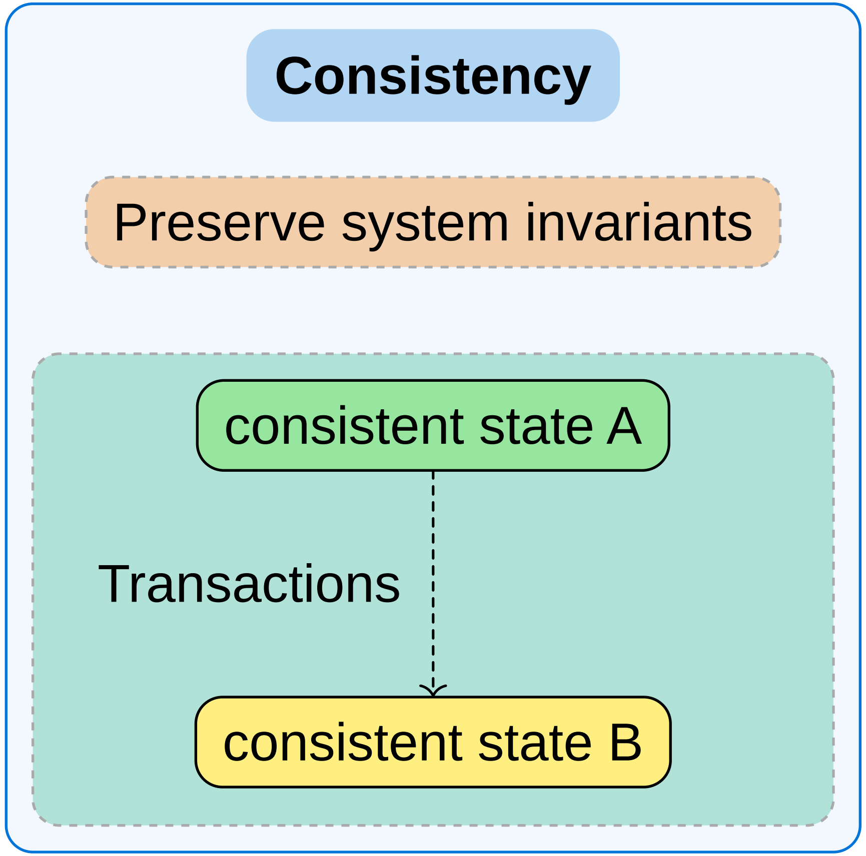
- All writes in a transaction are executed as a single unit
- Writes cannot be split into smaller parts
- If a fault occurs, all writes are rolled back
- Atomicity means “all or nothing”



ACID Properties

Consistency

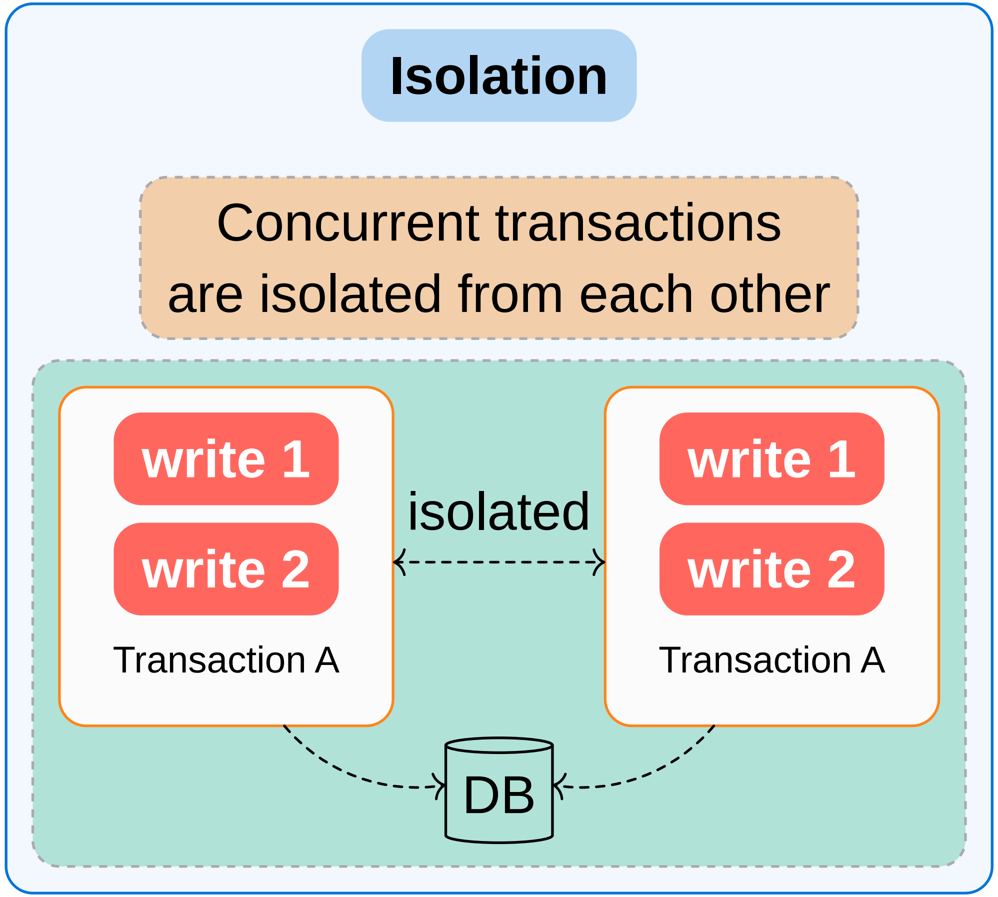
- Ensures that database invariants are preserved
  - e.g. account balances must not be negative
  - e.g. foreign keys must reference existing records
- Any data written by a transaction must satisfy all defined rules
- Transactions move the database from one valid state to another



ACID Properties

Isolation

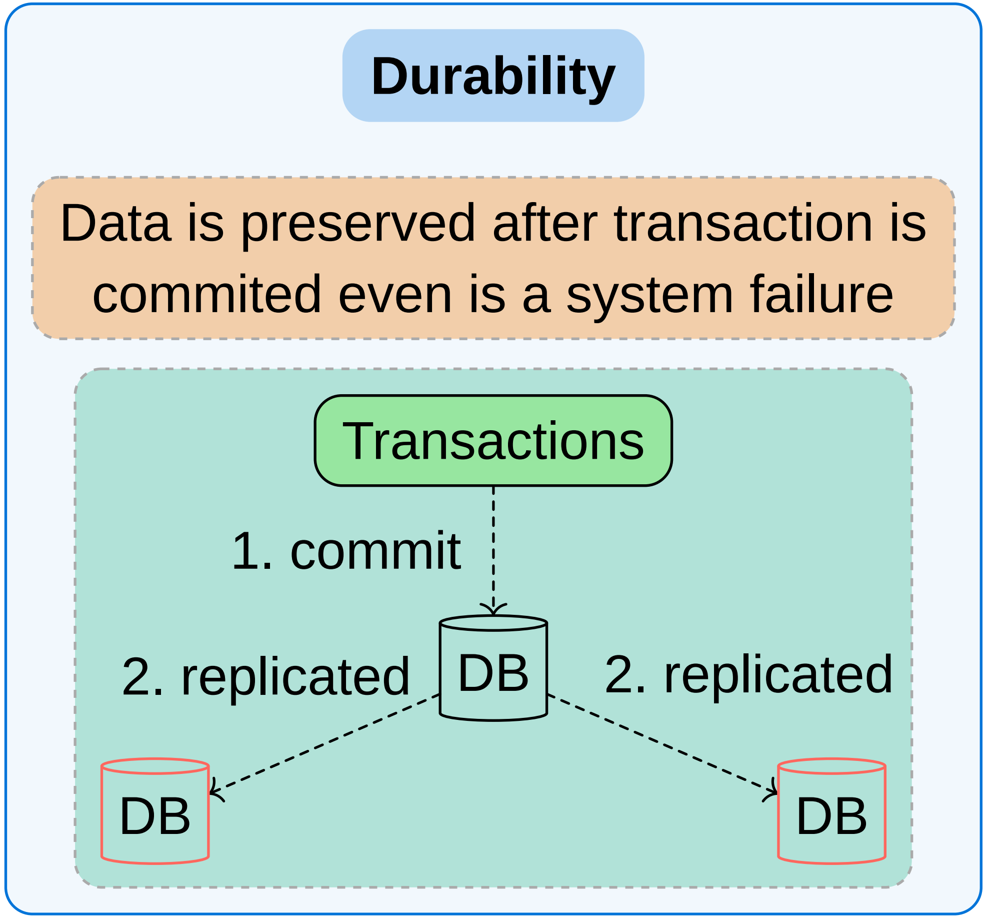
- Concurrent transactions do not interfere with each other
  - e.g. two updates to the same record do not see partial results
- **Serializability**: strongest isolation level
  - transactions behave as if executed one after another
- **Weak isolation**: used in practice for efficiency
  - transactions may see intermediate results



ACID Properties

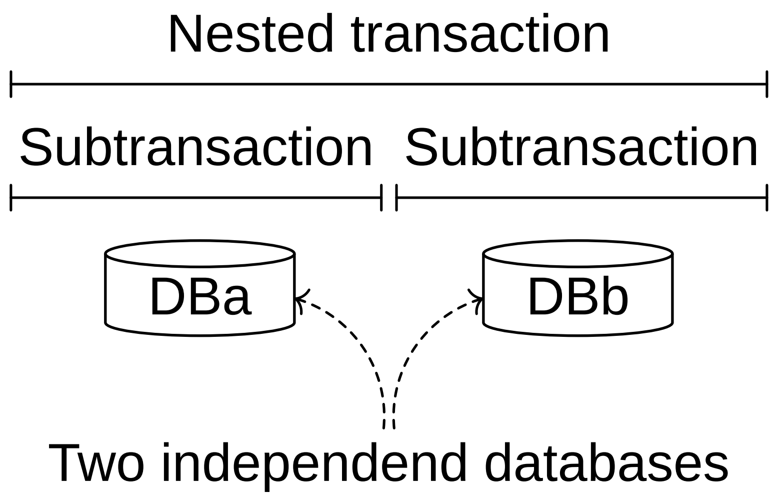
Durability

- Once a transaction commits, its data is not lost
- Data remains stored even after system failures
- In distributed systems, durability is achieved through replication



Nested Transactions

- A **nested transaction** is a transaction composed of multiple **subtransactions**
- The **top-level transaction** may:
  - Create subtransactions
  - Execute them **in parallel**
  - Run them on **different machines**
- Subtransactions can themselves create further subtransactions



Nested Transactions

Semantics

- Each (sub)transaction operates on a **private view of the data**
- If a **subtransaction commits**, its results become visible **only to its parent**
- If a **parent transaction aborts**, **all committed subtransactions are undone**
- **Durability applies only to top-level transactions**

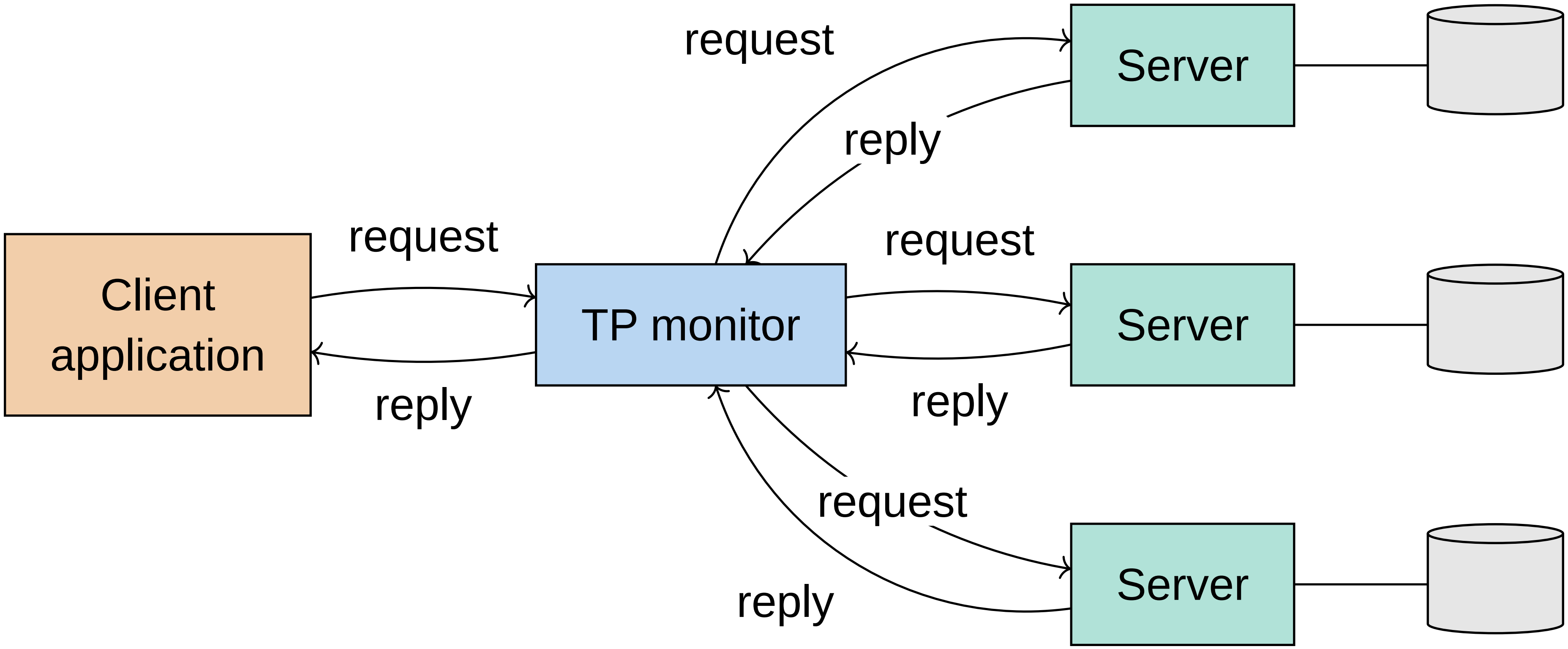
Why they matter

- Provide a natural way to **distribute a transaction across multiple machines**
- Allow logical decomposition of complex tasks
  - e.g., travel booking, flight + hotel + rental car

Transaction-Processing Monitor (TP Monitor)

- A **TP monitor** is middleware that supports the execution of **distributed (nested) transactions**
- It allows an application to:
  - Access **multiple servers or databases**
  - Using a **single transactional programming model**
- And it
  - Coordinates **subtransactions**
  - Ensures that the **collection of subtransactions satisfies the ACID properties**
  - Handles transaction coordination using a **distributed commit protocol**
- Applications do **not** need to implement transaction coordination themselves

Transaction-Processing Monitor (TP Monitor)



Pitfalls in Distributed systems

Designers often (wrongly) assume that:

- The network is **reliable**
- The network is **secure**
- Communication latency is **zero**
- Bandwidth is **infinite**
- The topology does **not change**
- There is **one administrator**
- Transport cost is **zero**
- The network is **homogeneous**

Distributed systems fail when they are designed based on assumptions that do not hold in practice.

Resources

- M. van Steen and A.S. Tanenbaum, Distributed Systems, 4th ed., distributed-systems.net, 2023.