

# Principles of Distributed Systems

inft-3507

Dr. J.Burns

**ADA University**

Autumn 2025

## **Section 1: Introduction**

*This content is based on the following public resources: <https://www.distributed-systems.net/index.php/books/ds4/>*

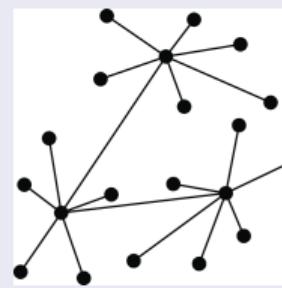
From networked systems to distributed systems

# Distributed versus Decentralized

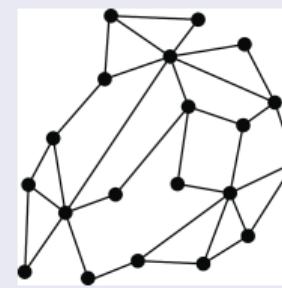
What many people state



Centralized



Decentralized



Distributed

When does a decentralized system become distributed?

- Adding 1 link between two nodes in a decentralized system?
- Adding 2 links between two other nodes?
- In general: adding  $k > 0$  links....?

# Alternative approach

## Distributed system

- A distributed system consists of multiple computers (nodes) that work together to appear as a single system to the user.
- **Focus:** performance, scalability, fault tolerance
- **Control** may still be centralized

## Decentralized system

- A decentralized system distributes control and decision-making authority across multiple independent nodes rather than relying on a central authority.
- **Focus:** autonomy, resilience to control, trust minimization
- **Control:** No single node has ultimate authority

# Control & Authority (The Main Difference)

## Distributed

Multiple nodes *compute together*, but control can still be centralized:

- Leader / coordinator often exists
- Decisions may flow from one organization or service

## Decentralized

Multiple nodes *share control*:

- No single node (or org) has ultimate authority
- Governance/decision-making spread across participants

## Key takeaway

A system can be **distributed yet centralized**; a **decentralized** system must distribute *control*.

# Architecture & Coordination

## Typical distributed coordination

- Client–server, master–worker, leader-based replication
- Specialized roles (leader, replicas, workers)

## Typical decentralized coordination

- Peer-to-peer style participation
- Collective coordination (e.g., voting/consensus-like mechanisms)

## Intuition

Distributed → “many machines, one system”

Decentralized → “many owners, shared authority”

# Fault Tolerance & Failure Modes

## Distributed systems

- Handle **node failures** (crashes, partitions)
- But may have **single point of control**

## Decentralized systems

- Handle node failures *and* control failures
- Aim to avoid **single authority failure**

## Practical distinction

Distributed: resilience to **hardware/service failure**

Decentralized: resilience to **control/censorship/authority failure**

# Trust & Security Model

## Distributed (often)

- Assumes trusted administrators / central policy
- Security enforced via access control and ops processes

## Decentralized (often)

- Assumes some participants may be malicious
- Uses mechanisms to reduce reliance on trust in any one node

## Rule of thumb

More decentralization usually means a **stronger adversarial model** (and more design complexity).

# Performance & Efficiency Trade-offs

Distributed systems tend to optimize

- Latency, throughput, operational efficiency
- Coordination cost kept low via leaders / centralized control

Decentralized systems tend to trade performance for

- Independence, robustness, shared governance
- Higher coordination overhead (more parties must agree)

Trade-off

Decentralization often increases cost of coordination  $\Rightarrow$  lower peak performance.

# Summary Comparison

## Quick comparison

- **Goal:** Distributed → scalability/performance    vs    Decentralized → autonomy/resilience
- **Control:** Distributed → may be centralized    vs    Decentralized → shared control
- **Trust:** Distributed → trusted operators    vs    Decentralized → trust-minimized
- **Complexity:** Distributed → moderate    vs    Decentralized → higher

# Design goals

# What do we want to achieve?

## Overall design goals

- Support sharing of resources
- Distribution transparency
- Openness
- Scalability

# Sharing resources

## Canonical examples

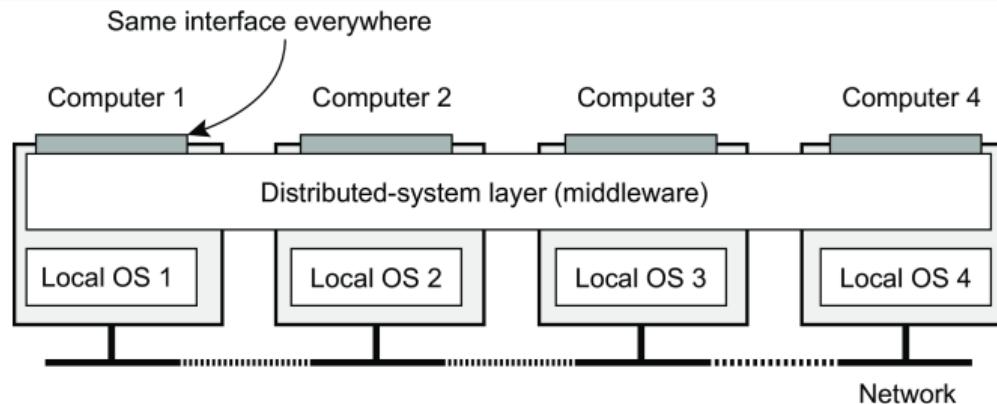
- Cloud-based shared storage and files
- Peer-to-peer assisted multimedia streaming
- Shared mail services (think of outsourced mail systems)
- Shared Web hosting (think of content distribution networks)

## Observation

*“The network is the computer”*

(John Gage, Sun Microsystems)

# Distribution transparency



## What is transparency?

*The phenomenon by which a distributed system attempts to **hide** the fact that its processes and resources are physically distributed across multiple computers, possibly **separated by large distances**.*

## Observation

Distribution transparency is handled through many different techniques in a layer between applications and operating systems: a **middleware layer**

# Distribution transparency

## Types

| Transparency       | Description   |
|--------------------|---|
| <b>Access</b>      | Hide differences in data representation and how an object is accessed |
| <b>Location</b>    | Hide where an object is located                                       |
| <b>Migration</b>   | Hide that an object may move to another location                      |
| <b>Replication</b> | Hide that an object is replicated                                     |
| <b>Concurrency</b> | Hide that an object may be shared by several independent users        |
| <b>Failure</b>     | Hide the failure and recovery of an object                            |

# Openness of distributed systems

## Open distributed system

A system that offers components that can easily be used by, or integrated into other systems. An open distributed system itself will often consist of components that originate from elsewhere.

## What are we talking about?

Be able to interact with services from other open systems, irrespective of the underlying environment:

- Systems should conform to well-defined interfaces
- Systems should easily interoperate
- This means publicly documented protocols and APIs
- Vendor-neutral standards
- Interoperability and extensibility

# Dependability

## Basics

A component provides services to clients. To provide services, the component may require the services from other components  $\Rightarrow$  a component may depend on some other component.

## Specifically

A component  $C$  depends on  $C^*$  if the correctness of  $C$ 's behavior depends on the correctness of  $C^*$ 's behavior. (Components are processes or channels.)

# Dependability

Requirements related to dependability

| Requirement            | Description                              |
|------------------------|--|
| <b>Availability</b>    | Readiness for usage                      |
| <b>Reliability</b>     | Continuity of service delivery           |
| <b>Safety</b>          | Very low probability of catastrophes     |
| <b>Maintainability</b> | How easy can a failed system be repaired |

# Traditional reliability - MTTF/MTTR

The following *traditional* metrics were derived from shop-floor machine reliability modelling over many years.

## Traditional metrics

- **Mean Time To Failure (MTTF)**: The average time until a component fails.
- **Mean Time To Repair (MTTR)**: The average time needed to repair a component.
- **Mean Time Between Failures (MTBF)**: Simply  $MTTF + MTTR$ .

You will often see *MTTF*, *MTTR*, *MTBF* used for modelling system reliability over time.

# Reliability v Availability

- *Availability* is about how much of the time the system is usable, allowing multiple failures and repairs.
- *Availability* is measured in *Uptime percentage* or *SLA "nines"* (eg, 5 9s)

$$A = \lim_{T \rightarrow \infty} \frac{\text{time system is up in } [0, T]}{T}$$

- *Reliability*: time to first failure
- *Reliability* is about whether failure occurs at all during an interval.

$$R(t) = \Pr\{\text{system survives without failure for time } t\}$$

- *Reliability* is measured Failure rate  $\lambda$  or MTTF

# Availability v Reliability: Numerical Example

## System 1

- Fails every 10 minutes
- Repair time = 1 second
- Availability  $\approx 99.83\%$
- Reliability over 1 hour  $\approx$  almost zero

## System 2

- Fails every 6 months
- Repair time = 6 hours
- Availability  $\approx 99.83\%$
- Reliability over 1 hour  $\approx$  almost 1

**Conclusion:** Two systems, with the same *availability*, but completely different *reliability*.

# A Note on Security

## Observation

A distributed system that is not secure, is not dependable

## What we need

- **Confidentiality:** information is disclosed only to authorized parties
- **Integrity:** Ensure that alterations to assets of a system can be made only in an authorized way

## Authorization, Authentication, Trust

- **Authentication:** verifying the correctness of a claimed identity
- **Authorization:** does an identified entity have proper access rights?
- **Trust:** one entity can be assured that another will perform particular actions according to a specific expectation

# Security mechanisms

## Symmetric cryptosystem

With encryption key  $E_K(\text{data})$  and decryption key  $D_K(\text{data})$ :

$\text{if } \text{data} = D_K(E_K(\text{data})) \text{ then } D_K = E_K$ . Note: encryption and decryption key are the same and should be kept **secret**.

## Asymmetric cryptosystem

Distinguish a **public key**  $PK(\text{data})$  and a **private (secret) key**  $SK(\text{data})$ .

- Encrypt message from *Alice* to *Bob*:  $\text{data} = \underbrace{SK_{\text{bob}}(\overbrace{PK_{\text{bob}}(\text{data})}^{\text{Sent by Alice}})}_{\text{Action by Bob}}$
- Sign message for *Bob* by *Alice*:  $[\text{data}, \underbrace{data \stackrel{?}{=} PK_{\text{alice}}(SK_{\text{alice}}(\text{data}))}_{\text{Check by Bob}}] = \underbrace{[\text{data}, SK_{\text{alice}}(\text{data})]}_{\text{Sent by Alice}}$

# Security mechanisms

## Secure hashing

In practice, we use **secure hash functions**:  $H(data)$  returns a **fixed-length string**.

- Any change from  $data$  to  $data^*$  will lead to a completely different string  $H(data^*)$ .
- Given a hash value, it is computationally impossible to find a  $data$  with  $h = H(data)$

## Practical digital signatures

Sign message for *Bob* by *Alice*:

$$[data, \underbrace{H(data) \stackrel{?}{=} PK_{alice}(sgn)}_{\text{Check by Bob}}] = [\underbrace{data, H, sgn = SK_{alice}(H(data))}_{\text{Sent by Alice}}]$$

# Scale in distributed systems

## Observation

Many developers of modern distributed systems easily use the adjective “scalable” without making clear **why** their system actually scales.

## At least three components

- Number of users or processes (**size scalability**)
- Maximum distance between nodes (**geographical scalability**)
- Number of administrative domains (**administrative scalability**)

## Observation

Most systems account only, to a certain extent, for size scalability. Often a solution: multiple powerful servers operating independently in parallel. Today, the challenge still lies in geographical and administrative scalability.

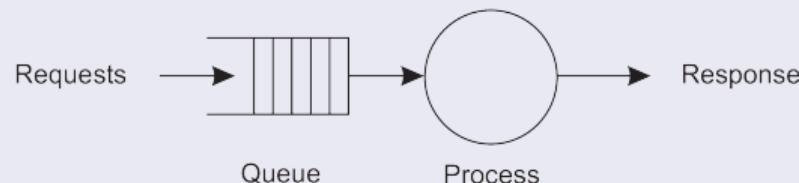
# Size scalability

## Root causes for scalability problems with centralized solutions

- The computational capacity, limited by the CPUs
- The storage capacity, including the transfer rate between CPUs and disks
- The network between the user and the centralized service

# Performance models

A centralized service can be modeled as a simple queuing system



## Assumptions and notations

- The queue has infinite capacity  $\Rightarrow$  arrival rate of requests is not influenced by current queue length or what is being processed.
- Arrival rate requests:  $\lambda$
- Processing capacity service:  $\mu$  requests per second

# Performance models

Utilization  $U$  of a service is the fraction of time that it is busy

$$U = \frac{\lambda}{\mu}$$

Average number of requests in the system

$$\bar{N} = \frac{U}{1 - U}$$

Average throughput

$$X = \frac{\lambda}{\mu} \cdot \mu = \lambda$$

via the Principle of Equilibrium (or conservation)

# Performance models

Response time: total time take to process a request after submission

From Little's Law:

$$\bar{N} = X \cdot R \Rightarrow R = \frac{\bar{N}}{X}$$

$$\Rightarrow R = \frac{1}{\mu \cdot (1 - U)}$$

## Observations

- If  $U$  is small, response-to-service time is close to 1: a request is immediately processed
- If  $U$  goes up to 1, the system comes to a grinding halt.

Solution: increase  $\mu$ .

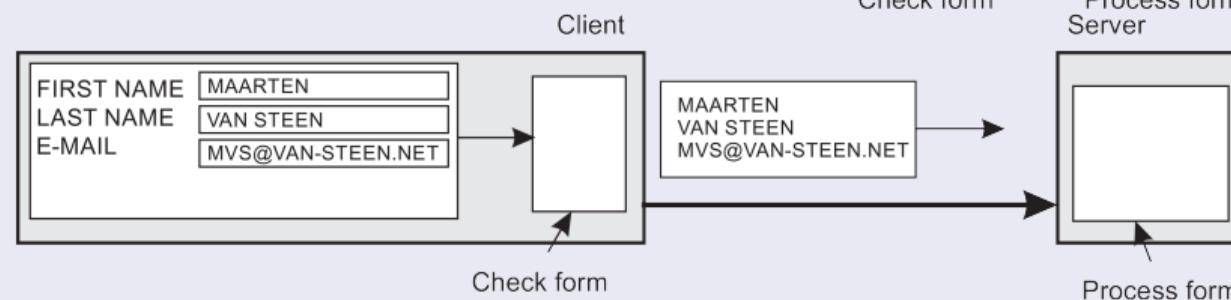
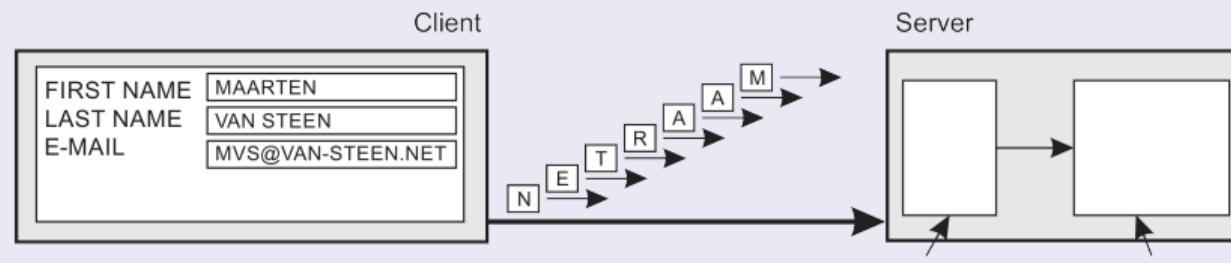
# Techniques for scaling

## Hide communication latencies

- Make use of [asynchronous communication](#)
- Have separate handler for incoming response
- Problem: not every application fits this model

# Techniques for scaling

Facilitate solution by moving computations to client



# Techniques for scaling

## Partition data and computations across multiple machines

- Move computations to clients (Java/ECMA script)
- Decentralized naming services (DNS)
- Decentralized information systems (WWW)

# Techniques for scaling

Replication and caching: Make copies of data available at different machines

- Replicated file servers and databases
- Mirrored Websites
- Web caches (in browsers and proxies)
- File caching (at server and client)

# Scaling: The problem with replication

Applying replication is easy, except for one thing

- Having multiple copies (cached or replicated), leads to **inconsistencies**: modifying one copy makes that copy different from the rest.
- Always keeping copies consistent and in a general way requires **global synchronization** on each modification.
- Global synchronization precludes large-scale solutions.

## Observation

If we can tolerate inconsistencies, we may reduce the need for global synchronization, but tolerating inconsistencies is application dependent.

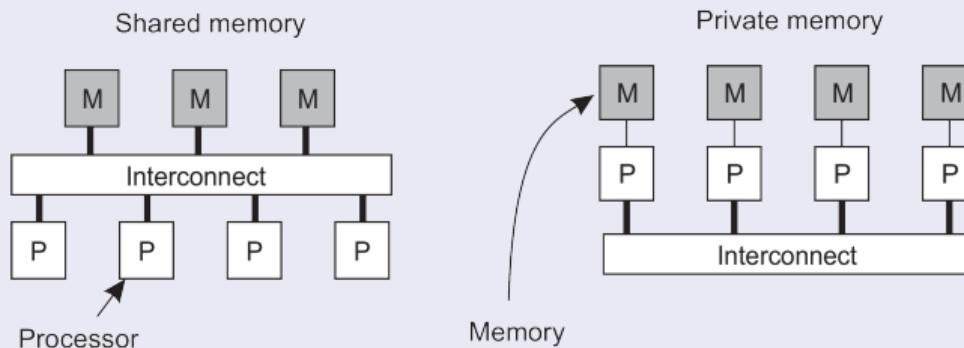
# A simple classification of distributed systems

# Parallel computing

## Observation

High-performance distributed computing started with parallel computing

## Multiprocessor and multicore versus multicompiler



# Distributed shared memory systems

## Observation

Multiprocessors are relatively easy to program in comparison to multicomputers, yet have problems when increasing the number of processors (or cores). **Solution:** Try to implement a **shared-memory model** on top of a multicomputer.

## Example through virtual-memory techniques

Map all main-memory pages (from different processors) into one **single virtual address space**. If a process at processor  $A$  addresses a page  $P$  located at processor  $B$ , the OS at  $A$  **traps and fetches**  $P$  from  $B$ , just as it would if  $P$  had been located on local disk.

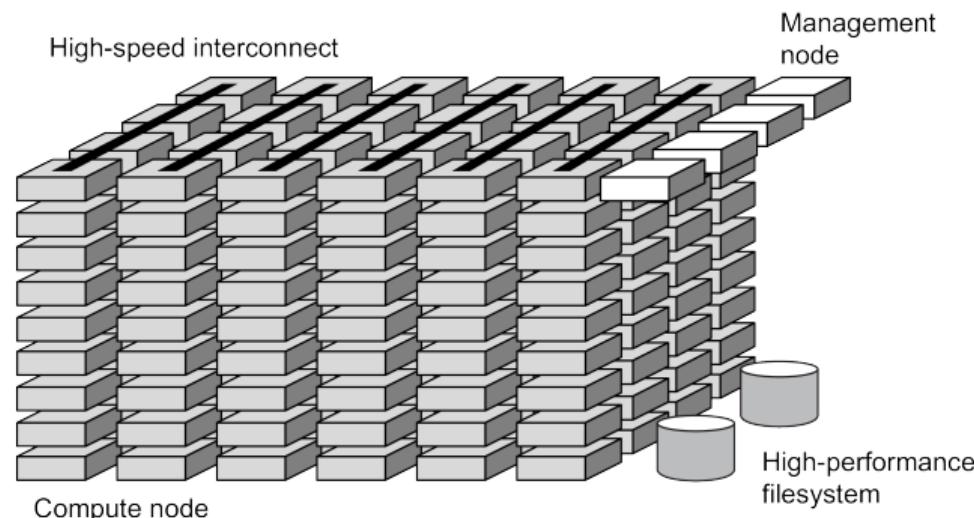
## Problem

Performance of distributed shared memory could never compete with that of multiprocessors, and failed to meet the expectations of programmers. It has been widely abandoned by now.

# Cluster computing

Essentially a group of high-end systems connected through a LAN

- Homogeneous: same OS, near-identical hardware
- Single, or tightly coupled managing node(s)



# Summary

# Summary and Conclusions

We have discussed some important principles in Distributed Systems, namely:

- Centralized, Decentralized and Distributed Types
- Support sharing of resources
- Distribution transparency
- Openness and Security
- Performance and Scalability