Principles of Distributed Systems

inft-3507

Dr. J.Burns

ADA University

Autumn 2025

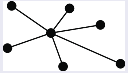
Section 1: Introduction

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

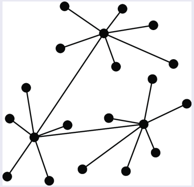
From networked systems to distributed systems

Distributed versus Decentralized

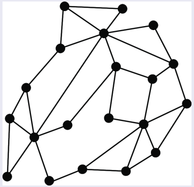
What many people state



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

Theoretical Definitions

- *Decentralized computing* - a networked computer system in which processes and resources are *necessarily* spread across multiple computers.
- *Distributed computing* - is a networked computer system in which processes and resources are *sufficiently* spread across multiple computers.

Modern ("Cloud") Definitions

- *Decentralized computing* - independent nodes operating without a central authority, making autonomous decisions
- *Distributed computing* - multiple interconnected nodes working collaboratively to solve a task, coordinated by a central system

- Both use multiple nodes but differ in control structure, coordination, and application focus, balancing autonomy versus efficiency
- A *node* is a compute resource with some local storage eg, a process, a container, a virtual machine, a dedicated server, a cluster of servers

Perspectives on distributed systems

Distributed systems are complex: take persepctives

- Architecture: common organizations
- Process: what kind of processes, and their relationships
- Communication: facilities for exchanging data
- Coordination: application-independent algorithms
- Naming: how do you identify resources?
- Consistency and replication: performance requires of data, which need to be the same
- Fault tolerance: keep running in the presence of partial failures
- Security: ensure authorized access to resources

Design goals

What do we want to achieve?

Overall design goals

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

Sharing resources

Canonical examples

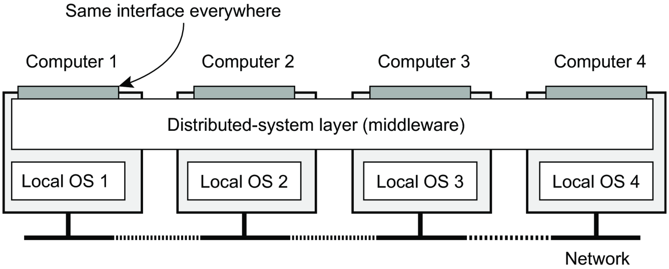
- Cloud-based shared storage and files
- Peer-to-peer assisted multimedia streaming
- Shared mail services (think of outsourced mail systems)
- Shared Web hosting (think of content distribution networks)

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

Distribution transparency

Types

Transparency	Description
Access	Hide differences in data representation and how an object is accessed
Location	Hide where an object is located
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 several independent users
Failure	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
- Systems should support portability of applications
- Systems should be easily extensible

Dependability

Basics

A component provides services to clients. To provide services, the component may require the services from other components ⇒ 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
Availability	Readiness for usage
Reliability	Continuity of service delivery
Safety	Very low probability of catastrophes
Maintainability	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)

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

- *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 λ or MTTF

Availability v Reliability: Numerical Example

System 1

- Fails every 10 minutes
- Repair time = 1 second
- *Availability* ≈ 99.83%
- *Reliability over 1 hour* ≈ almost zero

System 2

- Fails every 6 months
- Repair time = 6 hours
- *Availability* ≈ 99.83%
- *Reliability over 1 hour* ≈ almost 1

Same availability. Completely different reliability.

Terminology

Failure, error, fault

Term	Description	Example
Failure	A component is not living up to its specifications	Crashed program
Error	Part of a component that can lead to a failure	Programming bug
Fault	Cause of an error	Sloppy programmer

Terminology

Handling faults

Term	Description	Example
Fault prevention	Prevent the occurrence of a fault	Don’t hire sloppy programmers
Fault tolerance	Build a component and make it mask the occurrence of a fault	Build each component by two independent programmers
Fault removal	Reduce the presence, number, or seriousness of a fault	Get rid of sloppy programmers
Fault forecasting	Estimate current presence, future incidence, and consequences of faults	Estimate how a recruiter is doing when it comes to hiring sloppy programmers

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

Asymmetric cryptosystem

Distinguish a public key $PK(data)$ and a private (secret) key $SK(data)$.

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

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*:

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

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.

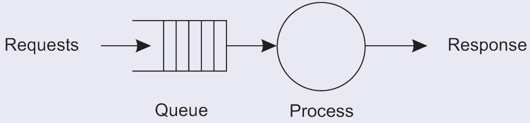
Size scalability

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: λ
- Processing capacity service: μ 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

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

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:

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

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

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 μ.

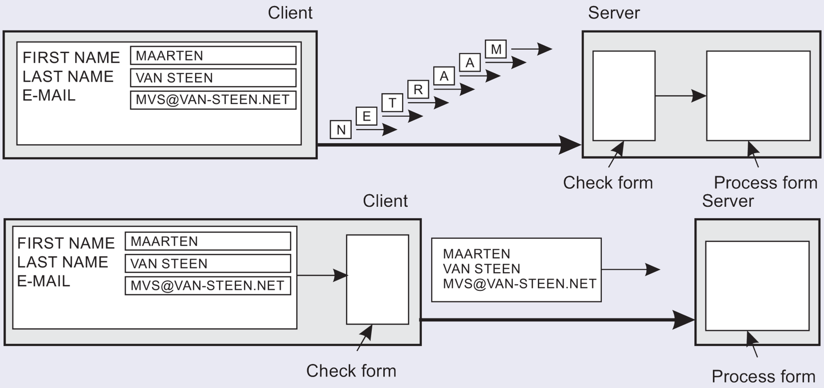
Techniques for scaling

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.

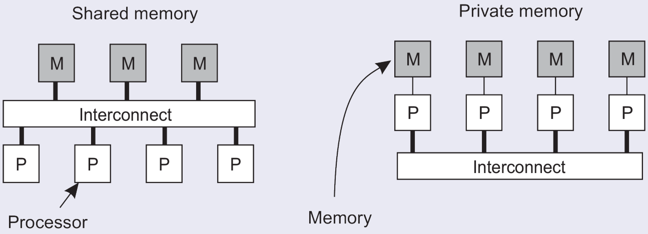
A simple classification of distributed systems

Parallel computing

Observation

High-performance distributed computing started with parallel computing

Multiprocessor and multicore versus multicomputer



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.

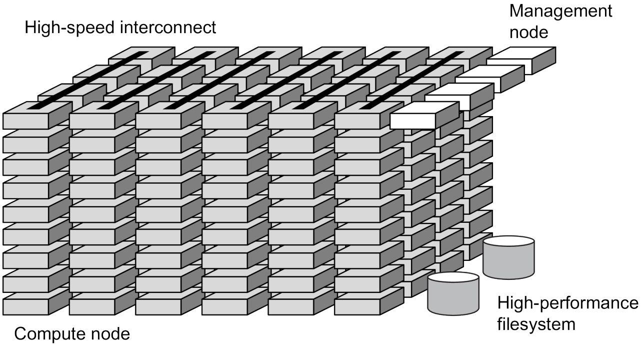
Problem

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