

UNIT 1

Introduction

Distributed Systems (CACS352) — BCA III/VI

⚠ Please note

*These notes were prepared with AI assistance and **may contain errors or omissions**. Please cross-check definitions, algorithms and numerical examples against your prescribed textbook (e.g., Tanenbaum & van Steen, *Distributed Systems*), your class notes and your instructor before relying on them for exams.*

1.1 Characteristics of Distributed Systems

A **distributed system** is a collection of autonomous computing elements (nodes) that appears to its users as a single coherent system. The nodes may be hardware devices or software processes; they cooperate only by exchanging messages over a network. Two ideas are central: (1) nodes are *autonomous*, each with its own clock and its own view of the state, and (2) users believe they are dealing with one system — this illusion is the job of **middleware**.

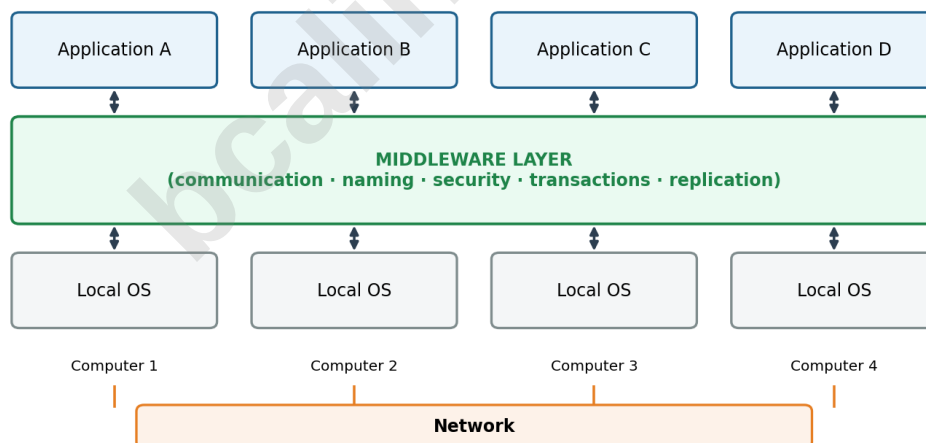


Figure 1.1 — Middleware as a layer between applications and local operating systems.

Main characteristics

- **Resource sharing** — hardware (printers, disks, CPUs), data and services are shared across the network.
- **Concurrency** — many components execute at the same time and compete for shared resources.

- **No global clock** — each node has its own clock; ordering of events must be established by message exchange (see Unit 6).
- **Independent failures** — a node or a link can fail while the rest keeps running; failures may be partial.
- **Heterogeneity** — different hardware, operating systems, networks and programming languages coexist.
- **Openness and scalability** — the system can be extended and can grow in size without degrading.
- **Message-passing communication** — there is no shared memory; all coordination uses messages with unpredictable delay.
- **Transparency** — complexity of distribution is hidden from users and application programmers.

Types of transparency

Transparency	What is hidden
Access	Differences in data representation and how a resource is accessed (e.g., byte order, file-system naming).
Location	Where a resource is physically located (e.g., a URL does not reveal the server's city).
Relocation	That a resource may be moved to another location while in use.
Migration	That a resource or process may move to another location (also called mobility).
Replication	That several copies of a resource exist.
Concurrency	That a resource may be shared by several competing users.
Failure	The failure and recovery of a resource.

Note: Some books merge migration into relocation. Full transparency is neither always possible nor desirable — the speed of light and the trade-off between consistency and availability set limits (a user far from the server will always see more latency than a nearby one).

1.2 Design Goals

1. Supporting resource sharing

The main reason to build a distributed system is to make it easy to access and share remote resources — files, printers, computing power, data and services (e.g., cloud storage, collaborative editing, peer-to-peer file sharing). The cost of sharing is added security and coordination complexity.

2. Making distribution transparent

A distributed system should hide the fact that processes and resources are physically distributed (see the table above). Transparency has a price in performance; for example, hiding a network failure completely may mean retrying for a long time, which harms responsiveness.

3. Being open

An **open** distributed system offers components that can easily be used by, or integrated into, other systems. Openness depends on published, standardised interfaces (often written in an Interface Definition Language, IDL).

- **Interoperability** — two implementations from different vendors can work together by relying on each other's services as specified by a common standard.
- **Portability** — an application developed for system A can run unchanged on system B that implements the same interfaces.
- **Extensibility** — it is easy to add new components or replace existing ones without affecting the rest.
- **Separating policy from mechanism** — the system provides mechanisms (e.g., a caching facility) while the policy (which pages to cache, for how long) is set by the application or user.

4. Being scalable

Scalability is measured along three dimensions: **size** (more users and resources), **geography** (users and resources far apart, so latency grows) and **administration** (many organisations with conflicting policies). Common scaling problems are centralised services, centralised data and centralised algorithms — a single server is a bottleneck and a single point of failure.

Technique	Idea	Example
Hiding communication latencies	Use asynchronous communication; move computation to the client	Form validation in the browser instead of the server
Partitioning / distribution	Split a component into parts and spread them out	DNS name space split into zones; Web split across servers
Replication / caching	Keep copies close to users; improves availability and load	Web proxies, CDNs; but creates consistency problems

Example 1.1 — Availability through replication

A service runs on one server that is available 99% of the time. Two independent replicas are deployed, and the service works if at least one of them is up. Find the availability.

Solution

Probability that one server is down = $1 - 0.99 = 0.01$.

Both down at the same time = $0.01 \times 0.01 = 0.0001$.

Availability = $1 - 0.0001 = 0.9999$, i.e., **99.99%** (downtime falls from ≈ 87.6 hours/year to ≈ 53 minutes/year).

Caveat: the replicas must fail independently and must be kept consistent.

Pitfalls — the fallacies of distributed computing

Many first-time designers make false assumptions. The classic list (L. Peter Deutsch and colleagues) is:

- The network is reliable.
- Latency is zero.
- Bandwidth is infinite.
- The network is secure.
- The topology does not change.
- There is one administrator.
- Transport cost is zero.
- The network is homogeneous.

1.3 Types of Distributed Systems

A. High-performance distributed computing systems

- **Cluster computing** — a collection of similar, tightly connected computers on a fast LAN, all running the same OS and managed centrally (e.g., Beowulf clusters). Typical use: parallel scientific computing.
- **Grid computing** — resources from different organisations are federated into a virtual organisation; nodes are heterogeneous and geographically dispersed (e.g., CERN's LHC Computing Grid).

- **Cloud computing** — computing is delivered on demand as a service over the Internet with pay-per-use billing and elasticity.

Cloud model	Provides	Example
IaaS — Infrastructure as a Service	Virtual machines, storage, networks	Amazon EC2, Google Compute Engine
PaaS — Platform as a Service	Runtime and tools to deploy applications	Google App Engine, Heroku
SaaS — Software as a Service	Complete applications over the Web	Gmail, Google Docs, Office 365

B. Distributed information systems

These integrate networked applications inside organisations. The dominant form is the **transaction processing system**: a client groups several operations into a transaction with primitives such as `BEGIN_TRANSACTION`, `END_TRANSACTION`, `ABORT_TRANSACTION`, `READ` and `WRITE`. Transactions satisfy the **ACID** properties:

- **Atomicity** — all operations happen or none do.
- **Consistency** — a transaction moves the system from one valid state to another.
- **Isolation** — concurrent transactions do not interfere; the result equals some serial execution.
- **Durability** — once committed, the effects survive crashes.

Enterprise Application Integration (EAI) lets separate applications communicate directly, using Remote Procedure Call (RPC), Remote Method Invocation (RMI) or message-oriented middleware (MOM) — covered in Unit 4.

C. Pervasive systems

Pervasive (ubiquitous) systems blend into the environment: smartphones, wearables, smart homes, sensor networks and the Internet of Things (IoT). They are characterised by **mobility**, context awareness, ad hoc and spontaneous interaction, and resource-constrained devices. Examples: a health-care body-area network, a smart-farm sensor network.

1.4 Case Study: The World Wide Web

The Web began at CERN (Tim Berners-Lee, 1989–1991) as a way for physicists to share documents. It is a huge client–server distributed system in which **browsers** (clients) fetch resources from **Web servers** using the **HyperText Transfer Protocol (HTTP)**. Resources are named by **URLs** (e.g., `https://www.example.com/index.html`), written mostly in HTML and rendered by the browser.

How a page request works

- The browser resolves the host name to an IP address using DNS.
- It opens a TCP connection (TLS for HTTPS) to the server, usually on port 80 or 443.
- It sends an HTTP request, e.g., `GET /index.html`.
- The server returns a response with a status code, headers and the body (HTML).
- The browser parses the HTML and fetches embedded objects (images, CSS, scripts) with further requests.

HTTP method	Purpose
GET	Retrieve a resource
HEAD	Retrieve only the headers of a resource
POST	Submit data to be processed (e.g., a form)
PUT	Store / replace a resource at a URL
DELETE	Remove a resource

Status class	Meaning	Example
1xx	Informational	100 Continue
2xx	Success	200 OK
3xx	Redirection	301 Moved Permanently
4xx	Client error	404 Not Found
5xx	Server error	500 Internal Server Error

Distributed-system aspects of the Web

- **Statelessness** — HTTP servers keep no session state between requests, which aids scalability; state is added with cookies or server-side sessions.

- **Caching** — browser caches, proxy servers and Content Delivery Networks (CDNs) reduce latency and load; freshness is controlled by headers such as Cache-Control and Expires.
- **Replication** — popular sites use many servers behind a load balancer; DNS or CDNs direct users to a nearby copy.
- **Multi-tier architecture** — Web server, application server and database server (see Unit 2).
- **Web services** — SOAP/WSDL and REST APIs let programs, not just people, use the Web.

Note: HTTP/1.1 supports persistent connections; HTTP/2 multiplexes several requests on one connection; HTTP/3 runs over QUIC (UDP). Check which version your syllabus emphasises.

Key Takeaways

- ✓ A distributed system = autonomous nodes that appear as a single coherent system, coordinated only by messages.
- ✓ Middleware provides the common services (communication, naming, security) that make distribution transparent.
- ✓ Design goals: resource sharing, transparency, openness, scalability — each with trade-offs.
- ✓ Scaling techniques: hide latency, partition, replicate/cache (replication introduces consistency issues).
- ✓ Types: high-performance (cluster, grid, cloud), information systems (transactions, EAI), pervasive systems.
- ✓ The Web is a stateless client-server system built on URLs, HTTP and HTML, scaled through caching and replication.

— End of Unit 1 —