

UNIT 9

Distributed Operating System

(4 Hrs) — Operating System (CACS251)

9.1 Introduction

A **distributed system** is a collection of independent computers (nodes) that appears to its users as a **single coherent system**. The nodes are connected by a network and have **no shared memory and no common clock**; they communicate by exchanging messages. A **distributed operating system (DOS)** manages the resources of all the machines so that users see one unified system — the location of processes and files is hidden (**transparency**).

Centralized system	Distributed system
One powerful computer (mainframe) with terminals	Many computers connected by a network
Single point of failure	Failure of one node does not stop the system
Scaling requires replacing the machine	Scale by adding more nodes
Shared memory and one clock	No shared memory; separate clocks

Goals of a distributed system

- **Resource sharing** — printers, files, databases and computing power are shared across the network.
- **Transparency** — access, location, migration, replication, concurrency and failure transparency.
- **Openness** — well-defined interfaces and standards allow components to be added or replaced.
- **Scalability** — works well as users, resources and geographic size grow.
- **Reliability, availability and performance** through replication and parallelism.

9.2 Advantages of Distributed Systems over Centralized Systems

Item	Advantage
Economics	Microprocessors offer a better price/performance ratio than mainframes (a collection of cheap CPUs can outperform one costly mainframe).

Item	Advantage
Speed	A distributed system may have more total computing power than any single mainframe (parallel execution).
Inherent distribution	Some applications are naturally distributed — a bank with many branches, air-traffic control, chain stores.
Reliability	If one machine crashes the system as a whole can continue with a modest loss of performance.
Incremental growth	Computing power can be added in small increments as the need grows.

9.3 Advantages of Distributed Systems over Independent PCs

Item	Advantage
Data sharing	Users can share a common database (e.g., an airline reservation system) instead of keeping separate copies.
Device sharing	Expensive peripherals (colour laser printers, tape libraries, large disks) are shared.
Communication	Person-to-person communication is easier — e-mail, file transfer, video conferencing.
Flexibility	Workload is spread over the available machines in the most efficient way.

9.4 Disadvantages of Distributed Systems

Problem	Explanation
Software	Little software exists for distributed systems; designing, implementing and using it is harder than for a centralized system.
Networking	The network can lose messages, become saturated, or be a single point of failure; adds latency.
Security	Easy data sharing also means easy access to secret data; more attack points; authentication across machines is difficult.
Complexity	Partial failures, concurrency, consistency and clock issues make the system harder to reason about and debug.
Cost of communication	Message passing is far slower than local memory access.

9.5 Hardware and Software Concepts

9.5.1 Hardware Concepts — Flynn's taxonomy

Class	Meaning	Example
SISD	Single instruction stream, single data stream	Traditional uniprocessor computer
SIMD	Single instruction, multiple data	Vector / array processors, GPUs
MISD	Multiple instruction, single data	Rare / theoretical
MIMD	Multiple instruction, multiple data — independent computers each with its own program counter and data	All distributed systems and multiprocessors

MIMD machines are further divided by memory: **multiprocessors** (shared memory — one virtual address space seen by all CPUs) and **multicomputers** (private memory — each machine has its own memory). Both can be organised as **bus-based** (a single network/backplane, e.g., a LAN — limited scalability) or **switched** (point-to-point wiring such as grids, hypercubes — scalable). Multicomputers may be *tightly coupled* (short delays, high data rate — parallel machines) or *loosely coupled* (a LAN of workstations).

9.5.2 Software Concepts

System	Description	Transparency / Scalability
Network Operating System (NOS)	Loosely coupled software on loosely coupled hardware. Each machine runs its own OS; users log into a specific machine and use remote commands (rlogin, ftp, NFS mounts).	Low transparency — users know the machines exist
True Distributed OS	Tightly coupled software on loosely coupled hardware; one <i>single-system image</i> — a virtual uniprocessor. The OS decides where processes run and where files are stored.	High transparency; hard to build
Middleware-based system	A layer above local OSes (CORBA, Web services, RPC frameworks) hides heterogeneity and offers distributed services.	Good scalability and openness — used in practice
Multiprocessor time-sharing OS	One copy of the OS on a shared-memory multiprocessor; single run queue.	Very high transparency; limited scalability

9.6 Communication in Distributed Systems

Because there is no shared memory, all communication uses **message passing** over the network. Layered protocol models (OSI/TCP-IP) define the format of messages. The **client-server model** is the most common: a client sends a request message to a server, which does the work and returns a reply. It is often implemented with a simple **connectionless** request/reply protocol.

9.6.1 Message Passing

The two basic primitives are **send(destination, message)** and **receive(source, message)**. Design issues:

Issue	Options
Blocking vs non-blocking	Blocking (synchronous): the sender is blocked until the message is received/acknowledged. Non-blocking (asynchronous): the sender continues immediately and the message is copied to a buffer.
Buffered vs unbuffered	Unbuffered: the receiver must already be waiting (<i>receive</i> posted) or the message is lost; buffered: messages wait in a mailbox/port until read.
Reliable vs unreliable	Reliability via acknowledgements, time-outs and retransmission; duplicates removed using sequence numbers.
Addressing	Direct (process ID), or indirect via mailboxes/ports.
Group communication	Unicast, multicast, broadcast.

9.6.2 Remote Procedure Call (RPC)

Birrell and Nelson (1984): let a program **call a procedure located on another machine** exactly as if it were local. When process A on machine 1 calls a procedure on machine 2, A is suspended and execution continues on machine 2; information travels in parameters and comes back in the result. No message passing is visible to the programmer — RPC hides the network (**access transparency**).

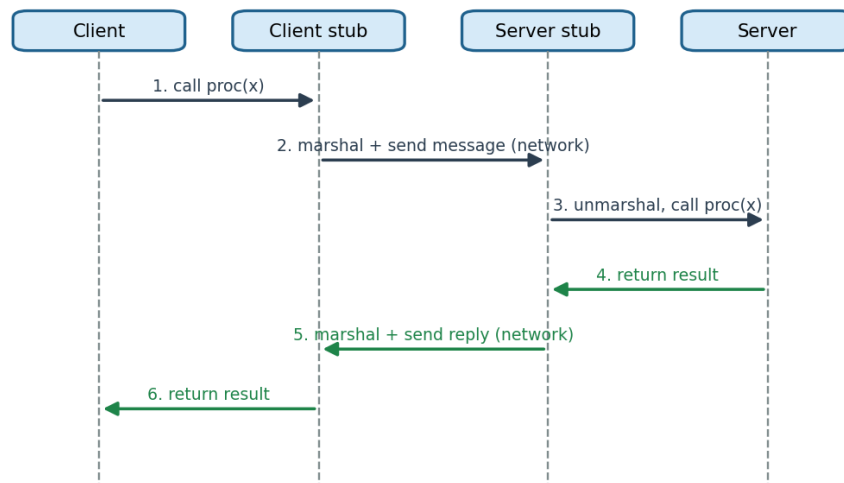


Figure 9.1 — Steps of a remote procedure call using stubs

Steps of an RPC

- **1.** The client calls the **client stub** as an ordinary local procedure.
- **2.** The client stub **marshals** (packs) the parameters into a message and makes a system call to send it.
- **3.** The OS sends the message over the network to the server machine.
- **4.** The server OS passes the message to the **server stub**, which **unmarshals** the parameters.
- **5.** The server stub calls the actual procedure on the **server**, which does the work and returns the result.
- **6.** The server stub marshals the result, the reply is sent back, the client stub unmarshals it and returns to the client as if from a local call.

Problems in RPC

- **Parameter passing:** pointers and references are meaningless on another machine (pass by copy/restore); different data representations (big vs little endian, ASCII vs EBCDIC) require a standard format (XDR, ASN.1).
- **Binding:** the client must locate the server — through a **name/directory server** or port mapper.
- **Failure semantics:** lost request, lost reply, server crash, client crash (orphans). Semantics may be *at-least-once*, *at-most-once* or *exactly-once* (hardest).
- **Not fully transparent:** global variables and latency; a remote call is about 1000 × slower than a local one.
- Examples: Sun RPC/ONC RPC, Java RMI, gRPC, XML-RPC and SOAP.

9.7 Processes in Distributed Systems

Threads are especially valuable in distributed systems: a multithreaded client can hide network latency by working on other requests while waiting for a reply (e.g., a browser fetching several images concurrently), and a multithreaded server can handle many clients at once. **Server organisation:** *dispatcher/worker model* (one dispatcher thread accepts requests and hands them to worker threads), *team model* and *pipeline model*.

Processor allocation and process migration

In a distributed system there are many CPUs; deciding **which process runs on which machine** is **processor allocation**. Models: workstation model (each user's local machine), **processor pool** model (a rack of CPUs allocated on demand) and hybrid. Algorithms may be *deterministic or heuristic, centralized or distributed, optimal or suboptimal, sender-initiated or receiver-initiated*. **Load balancing** can involve **process migration** — moving a running process from an overloaded machine to an idle one (state transferred and re-created).

Code migration and Virtualisation

Moving *code* instead of data can improve performance (send a query program to the database server) and flexibility. Virtual machines and containers are used to host distributed processes and isolate them.

9.8 Clock Synchronization

In a single computer all processes see the same clock. In a distributed system every machine has its own **physical clock** (a quartz crystal) which **drifts** at a slightly different rate, so times diverge (**clock skew**). Yet many algorithms — distributed make, transaction ordering, file time-stamps, security tickets — require agreement on time.

9.8.1 Physical Clock Synchronisation

Clocks are kept within a maximum drift rate ρ . If two clocks drift in opposite directions, after Δt they differ by up to $2\rho\Delta t$; to guarantee a maximum skew δ the clocks must be resynchronised at least every $\delta / 2\rho$ seconds. Reference time comes from **UTC** (via shortwave radio, GPS or NTP servers).

Algorithm	How it works	Type
Cristian's algorithm	A client asks a time server (with a UTC receiver) for the time. The client sets its clock to $T_{\text{server}} + (T_1 - T_0)/2$, where T_0 and T_1 are the times the request was sent and the reply received (half the round-trip delay is added). Repeated measurements reduce error.	Passive time server; centralized
Berkeley algorithm	A time daemon polls every machine for its time, computes the average (ignoring outliers), and tells each machine the amount by which to adjust (not the absolute time) — no UTC source needed; keeps machines mutually consistent.	Active time server; centralized
Network Time Protocol (NTP)	Hierarchical (strata) of servers on the Internet; clients synchronise with several servers using statistical filtering; accuracy of a few milliseconds.	Distributed, Internet standard
Averaging algorithms	Each machine broadcasts its time at regular intervals; each averages the values from all others (decentralised).	Decentralized

Example — Cristian's algorithm

A client sends a request at its local time $T_0 = 10:00:00.000$ and receives the reply at $T_1 = 10:00:00.040$. The reply says the server time is $10:00:05.020$.

Round-trip = 40 ms $\rightarrow$ one-way $\approx$ 20 ms. Client sets the clock to $10:00:05.020 + 0.020 = 10:00:05.040$.

Example — Berkeley algorithm

The time daemon's clock reads 3:00. Machine A reads 3:25 and machine B reads 2:50.

Average = $(3:00 + 3:25 + 2:50)/3 = 3:05$. Daemon tells itself to advance by +5, machine A to **slow by 20** (3:25 $\rightarrow$ 3:05), machine B to advance by +15 (2:50 $\rightarrow$ 3:05).

9.8.2 Logical Clocks — Lamport's Algorithm

Lamport (1978) observed that processes need to agree on the **order of events**, not on the absolute time. He defined the **happens-before** relation ($\rightarrow$): (1) if a and b are events in the same process and a occurs before b, then $a \rightarrow b$; (2) if a is the sending of a message and b is its receipt, then $a \rightarrow b$; (3) if $a \rightarrow b$ and $b \rightarrow c$ then $a \rightarrow c$. Two events not related by $\rightarrow$ are **concurrent**.

Each process keeps a **logical clock counter C** incremented between successive events. Every message carries the sender's timestamp. **Rule:** when a message arrives whose timestamp is greater than the receiver's clock, the receiver sets its clock to **(timestamp + 1)**. This guarantees: if $a \rightarrow b$ then $C(a) < C(b)$. Ties between processes can be broken using process numbers to get a total order.

Example — Lamport timestamps

Process P1 sends a message *m* at logical time 6 (its clock = 6). Process P2's clock currently reads 3 when *m* arrives.

Since $6 > 3$, P2 sets its clock to $6 + 1 = 7$ on receipt. Had P2's clock been 9, it would simply continue ($9 \rightarrow 10$ for the receive event).

Vector clocks

Lamport clocks cannot tell whether two events are concurrent. **Vector clocks** keep an *array* of counters, one per process; comparing vectors shows causality: $V(a) < V(b)$ means *a* happened before *b*, and if neither is smaller the events are concurrent.

Key Takeaways

- ✓ A **distributed system** = independent computers appearing as one; no shared memory or clock; communication by messages.
- ✓ Advantages over centralized: price/performance, speed, reliability, incremental growth; over independent PCs: data/device sharing, communication, flexibility. Disadvantages: software, network, security, complexity.
- ✓ Hardware: MIMD multiprocessors (shared memory) and multicomputers; software: network OS, true distributed OS, middleware.
- ✓ **RPC** hides message passing using **client and server stubs** and marshalling of parameters.
- ✓ **Clock synchronisation**: Cristian (time server), Berkeley (average), NTP; **Lamport's logical clocks** order events with the happens-before relation.

— End of Unit 9 —