

UNIT 4

Communication

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.*

4.1 Foundations

Interprocess communication is at the heart of every distributed system. Because there is no shared memory, processes communicate by **message passing** over a network, according to agreed rules called **protocols**.

Layered protocols

The **OSI reference model** has seven layers: Physical, Data link, Network, Transport, Session, Presentation, Application. On the Internet a simplified stack is used (Link, IP, TCP/UDP, Application). Distributed systems add a **middleware layer** between transport and application that offers general-purpose protocols: authentication, commit protocols, RPC, locking, marshalling and multicast. Applications then rely on these instead of raw sockets.

Types of communication

Type	Meaning	Example
Transient	Message is stored only while sender and receiver are both running; discarded if it cannot be delivered	Sockets, RPC
Persistent	Message is stored by the middleware until delivered, even if the receiver is not running	E-mail, message queues
Synchronous	Sender blocks until the message is stored / received / processed	Classic RPC
Asynchronous	Sender continues immediately after handing over the message	Message queuing
Discrete	Each message is a complete unit of information	Requests, e-mail
Streaming	Continuous flow of data with timing constraints	Audio, video

Note: A sender may block at three points: until the message is copied to the local buffer, until it arrives at the receiver, or until the receiver has processed it. The stronger the guarantee, the longer the sender waits.

Marshalling and data representation

Different machines represent data differently (e.g., byte order, character sets). Before sending, parameters are converted to a common network representation — **marshalling** — and converted back at the receiver — **unmarshalling**.

Example 4.1 — Endianness

The 32-bit integer 5 is sent from a big-endian machine to a little-endian machine without conversion. What does the receiver read?

Solution

Big-endian byte sequence for 5: 00 00 00 05 (most significant byte first).

A little-endian receiver reads the first byte as the least significant: $0x05000000 = 5 \times 16^6 = \mathbf{83\ 886\ 080}$.

Hence both sides must agree on a canonical wire format (e.g., network byte order = big-endian, or XDR, Protocol Buffers).

4.2 Remote Procedure Call (RPC)

RPC (Birrell & Nelson, 1984) lets a program call a procedure that executes on another machine as if it were local, hiding all message passing. The client calls a **client stub**, which marshals parameters into a message and sends it; the **server stub** unmarshals it, calls the real procedure, and returns the result in the same way.

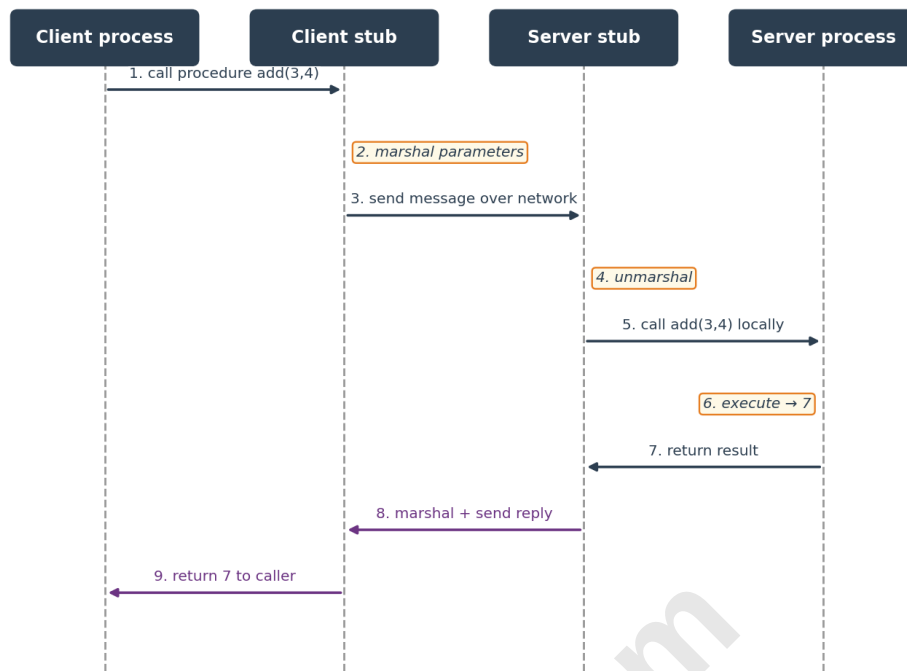


Figure 4.1 — Steps of a remote procedure call.

Parameter passing

- **Call-by-value** — values are copied into the message (easy).
- **Call-by-reference** — pointers are meaningless in another address space. Common solution: copy the referenced data into the message and copy back (*call-by-copy/restore*), or use global references to remote objects.
- **Interface Definition Language (IDL)** — procedures and types are declared in an IDL; a compiler generates stubs for client and server in the chosen languages.

Binding

Before a call, the client must **bind** to a server: the server registers its service and endpoint with a directory service or a daemon; the client looks up the server at run time (dynamic binding), which lets services move or be replicated.

Asynchronous RPC

In **asynchronous RPC** the client continues once the server acknowledges the request; the server later returns results. In **deferred synchronous RPC** the client continues, then polls or waits for the result later. **One-way RPC** needs no reply at all.

RPC failure semantics

Five things can go wrong: the client cannot locate the server, the request is lost, the server crashes after receiving it, the reply is lost, or the client crashes after sending it (leaving an **orphan** computation). Retrying makes the semantics differ:

Semantics	Behaviour	Suitable for
Maybe	No retry; call may or may not have executed	Non-critical operations
At-least-once	Client retries until a reply; operation may run more than once	Idempotent operations (e.g., read a value)
At-most-once	Duplicates are filtered with request ids; the operation runs zero or one time	Most RPC systems (e.g., Java RMI)
Exactly-once	Executed once and only once — impossible to guarantee in general with crashes	Needs transactions / idempotent design

4.3 Message-Oriented Communication

Transient messaging — sockets

The **Berkeley socket** interface provides transport-level access for TCP/UDP. Server-side primitives: **socket** (create endpoint), **bind** (attach local address), **listen** (announce willingness to accept), **accept** (block until a connection arrives). Client-side: **socket**, **connect**. Both then use **send/receive** and finally **close**.

Libraries such as **ZeroMQ** and the **Message-Passing Interface (MPI)** provide higher-level messaging on top of sockets or fast interconnects.

Persistent messaging — message-queuing systems

In **message-queuing systems** (message-oriented middleware, MOM) applications communicate by inserting messages into **queues** that are managed by **queue managers**. Sender and receiver need not run at the same time; the system guarantees eventual delivery but not when. Primitives:

Primitive	Meaning
Put	Append a message to a specified queue (non-blocking)
Get	Block until the queue is non-empty, then remove the first message
Poll	Check a queue; remove the first message if any — never blocks
Notify	Install a handler (callback) called when a message arrives

Queues are addressed by (queue manager, queue name). A **relay** forwards messages between queue managers, forming an overlay network. **Message brokers** convert message formats between applications (see Unit 2). Modern examples: IBM MQ, RabbitMQ (AMQP), Apache Kafka, MQTT.

4.4 Multicast Communication

Multicast sends data to a group of receivers. IP multicast exists but is rarely deployed across the Internet, so **application-level multicasting** is used: nodes organise into an overlay (tree or mesh) and forward data along it.

- **Tree-based** — a single path between any two nodes; efficient, but repairing failures is hard.
- **Mesh-based** — each node has several neighbours; more robust (e.g., Chord-based multicast).
- **Cost measures** — *link stress* (how often a message crosses the same physical link), *stretch* (ratio of overlay to direct delay) and *tree cost*.

Gossip-based (epidemic) data dissemination

Updates spread like a disease. A node holding an update is **infected**, one not yet updated is **susceptible**, and one that no longer spreads it is **removed**.

- **Anti-entropy** — every node periodically picks another at random and exchanges updates: push, pull or push-pull. Push-pull is fastest. Spreading to all N nodes takes $O(\log N)$ rounds.
- **Rumour spreading** — a node with a new update pushes it to random nodes and stops (loses interest) with some probability each time it contacts an already-informed node. Fast but may leave a small fraction uninformed.
- **Deletion** — spread special *death certificates*, otherwise deleted items would be re-infected.

Example 4.2 — How fast does gossip spread?

In a system of $N = 1024$ nodes using push-pull anti-entropy, roughly how many rounds are needed until all nodes have an update (approximation $\log_2 N$)?

Solution

Rounds $\approx \log_2 1024 = 10$. Because informed nodes roughly double each round: $1 \rightarrow 2 \rightarrow 4 \rightarrow \dots \rightarrow 1024$.

In practice the count is a small constant times $\log N$ because contacts are random and can overlap.

4.5 Case Study: Java RMI and MPI

Java Remote Method Invocation (RMI)

Java RMI is the object-oriented equivalent of RPC. A **remote object** implements a **remote interface** (extends `java.rmi.Remote`, every method declares `RemoteException`). The client holds a **stub** (proxy) that forwards calls to the server-side object; the RMI runtime handles serialisation. Parameters that are *Serializable* are passed by copy; remote objects are passed by reference (as stubs). Objects are located through the **RMI registry**.

```
// Remote interface
public interface Hello extends java.rmi.Remote {
    String sayHello(String name) throws java.rmi.RemoteException;
}
// Server
public class HelloImpl extends java.rmi.server.UnicastRemoteObject
    implements Hello {
    public HelloImpl() throws java.rmi.RemoteException {}
    public String sayHello(String n) { return "Hello, " + n; }
    public static void main(String[] a) throws Exception {
        java.rmi.registry.LocateRegistry.createRegistry(1099);
        java.rmi.Naming.rebind("rmi://localhost/Hello", new HelloImpl());
    }
}
// Client
Hello h = (Hello) java.rmi.Naming.lookup("rmi://localhost/Hello");
System.out.println(h.sayHello("Nepal"));
```

Java RMI provides at-most-once semantics, distributed garbage collection through leases, and dynamic class loading of stubs.

Message Passing Interface (MPI)

MPI is a standard library for message passing among processes in parallel and cluster applications (C, C++, Fortran). Processes belong to **communicators** (e.g., `MPI_COMM_WORLD`) and are identified by a **rank**. It was designed for high-performance computing, offers rich sending modes and collective operations, and is not persistent (transient communication).

Primitive	Meaning
<code>MPI_send</code>	Send a message and wait until it is copied to a local or remote buffer
<code>MPI_ssend</code>	Synchronous send — wait until the receiver has started to receive
<code>MPI_bsend</code>	Copy to a user-supplied buffer and continue
<code>MPI_isend / MPI_issend</code>	Non-blocking variants; check completion later

Primitive	Meaning
MPI_recv / MPI_irecv	Blocking / non-blocking receive
MPI_sendrecv	Send and wait for a reply in one call
MPI_Bcast, MPI_Scatter, MPI_Gather, MPI_Reduce, MPI_Barrier	Collective operations on a group

```
#include
int main(int argc, char** argv) {
    int rank, size, x;
    MPI_Init(&argc, &argv);
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
    MPI_Comm_size(MPI_COMM_WORLD, &size);
    if (rank == 0) { x = 42; MPI_Send(&x, 1, MPI_INT, 1, 0, MPI_COMM_WORLD); }
    else if (rank == 1) { MPI_Recv(&x, 1, MPI_INT, 0, 0, MPI_COMM_WORLD, MPI_STATUS_IGNORE); }
    MPI_Finalize(); return 0;
}
```

Note: The syllabus line reads 'Message Passing Interface (MN)' — this is almost certainly a typo for MPI.

Feature	RPC / RMI	Message queuing	MPI
Coupling	Tight (call-reply)	Loose	Tight, within a job
Persistence	Transient	Persistent	Transient
Typical use	Client-server services	Enterprise integration	Parallel computing

Key Takeaways

- ✓ Communication may be transient/persistent, synchronous/asynchronous and discrete/streaming.
- ✓ RPC hides message passing using stubs, marshalling and an IDL; failures give at-least-once or at-most-once semantics.
- ✓ Sockets give raw transient messaging; message-queuing systems give persistent, asynchronous, loosely coupled messaging with put/get/poll/notify.
- ✓ Multicast is done at application level using tree or mesh overlays; gossip (epidemic) protocols spread updates in $O(\log N)$ rounds.
- ✓ Java RMI = object-oriented RPC (remote interfaces, stubs, registry); MPI = rank-based message passing for parallel programs.

— End of Unit 4 —