Sistemas Distribuídos

Message Passing

Eurico Pedrosa

António Rui Borges

Universidade de Aveiro - DETI

2025-03-19

Communication Systems

Performance

Factors Influencing the Performance of a Communication System:

- **Latency** – The delay that is experienced between the execution of a send operation and the commencement of data reception. This can be visualized as the transmission of an empty message.
- **Data Transfer Rate** – The speed at which data is transmitted between the sender and the receiver.
- **Bandwidth** – The system throughput, defined as the volume of message traffic that is processed per unit of time.

The total **message transmission time** is determined by the following equation:

$$\text{Transmission Time} = \text{Latency} + \frac{\text{Message Length}}{\text{Data Transfer Rate}}$$

QoS

- **Quality of Service (QoS)** defines the system's ability to meet **deadline constraints** for **transmitting and processing continuous data flows**.
- To ensure smooth operation:
  - **Latency** must stay **below** a defined **upper limit**.
  - **Bandwidth** must stay **above** a defined **lower limit**.

Reliability and Error Handling

- Modern **communication systems** are **highly reliable**.
- **Failures** are more often caused by **software errors** (on the sender or receiver side) rather than **network issues**.
- **Error detection and correction** is **delegated to applications**, following the **end-to-end argument**.

Abstraction for Application Programmers

- The **communication system** should be **integrated and abstract**, **hiding** the complexity of **underlying physical networks**.
- **Network software** is organized in a **hierarchy of layers** to provide a structured approach.
- Each **layer** presents an **interface** to the **layer above**, describing the **communication system logically**.

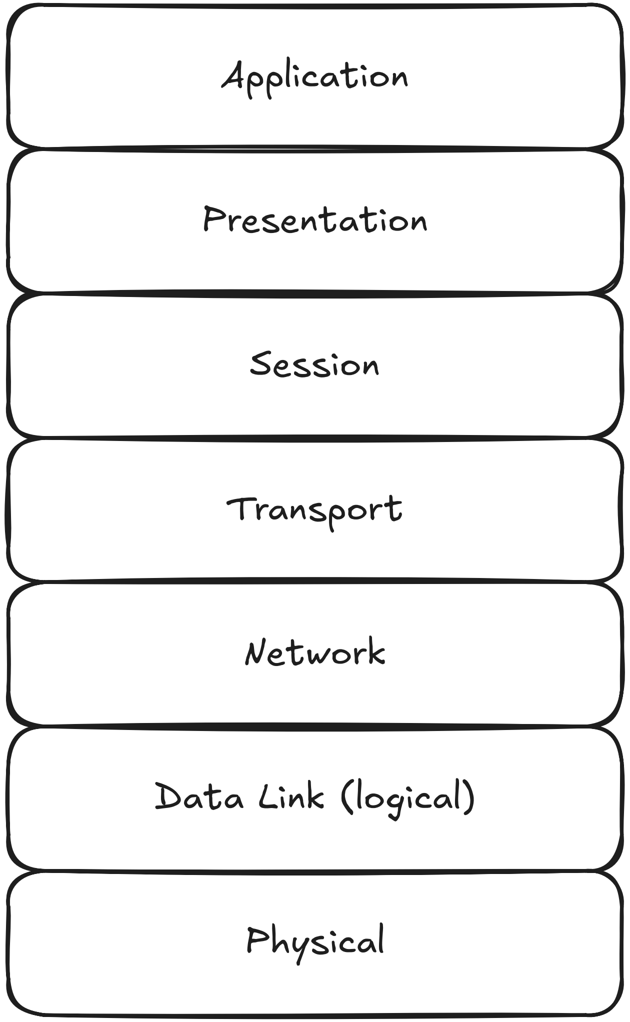
Layered Communication and Data Flow

- A **layer** is implemented as a **software module** in every **networked computer system**.
- Each module **appears** to communicate **directly** with its counterpart on another system.
- However, **direct transmission** between layers on different systems **does not occur**.
- Instead, each layer **communicates locally** through **procedure calls** to adjacent layers.

Data Encapsulation and Transformation

- **Sending side:**
  - Each layer **(except the topmost)** receives **data from the layer above**.
  - It **encapsulates** the data in a **new format** before passing it **to the layer below**.
- **Receiving side:**
  - Data is **processed in reverse**, with each layer **removing encapsulation** and passing it **upward**.

OSI Model



1. **Application Layer**
   - **Examples**: HTTP (HyperText Transfer Protocol), FTP (File Transfer Protocol), SMTP (Simple Mail Transfer Protocol), DNS (Domain Name System)
2. **Presentation Layer**
   - **Examples**: TLS (Transport Layer Security), SSL (Secure Sockets Layer), JPEG, MPEG, ASCII, XML, JSON
3. **Session Layer**
   - **Examples**: TCP (Transmission Control Protocol - error detection and retransmission), ARQ (Automatic Repeat reQuest), FEC (Forward Error Correction), CRC (Cyclic Redundancy Check)
4. **Transport Layer**
   - **Examples**: TCP (Transmission Control Protocol), UDP (User Datagram Protocol), SCTP (Stream Control Transmission Protocol)
5. **Network Layer**
   - **Examples**: IP (Internet Protocol), ICMP (Internet Control Message Protocol), RIP (Routing Information Protocol), OSPF (Open Shortest Path First)
6. **Data Link Layer**
   - **Examples**: Ethernet (IEEE 802.3), Wi-Fi (IEEE 802.11), PPP (Point-to-Point Protocol), MAC (Media Access Control)
7. **Physical Layer**
   - **Examples**: RS-232 (Serial Communication), USB (Universal Serial Bus), Fiber Optic Standards, Ethernet Physical Layer (IEEE 802.3), Bluetooth

Programming Interface

Middleware and Sockets

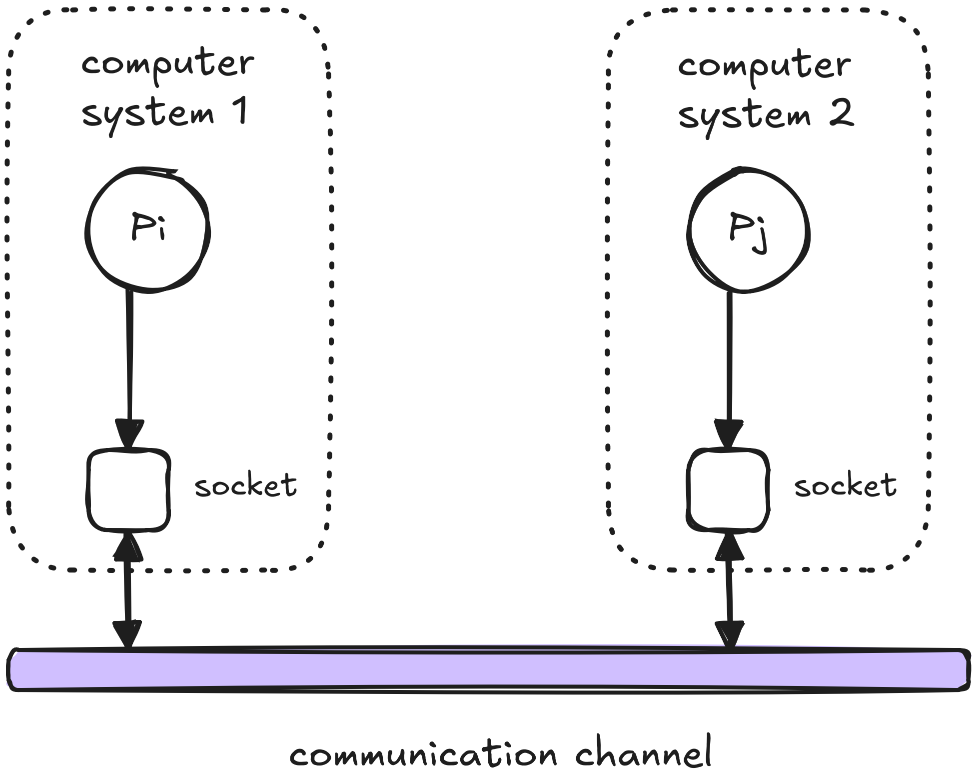
Role of Middleware

- Middleware provides an **abstraction layer** for **communication between processes** that do **not share an address space**.
- It offers a communication device called a **socket** or **end-point of communication**.

Socket Characteristics

- A **socket** is uniquely identified by:
  - An **IP address** – Identifies the **computer system**.
  - A **port number** – Specifies the **communication channel's endpoint** within the system.

Middleware and Sockets



Middleware and Sockets

Transmission Control Protocol (TCP)

- **Connection-oriented**: A **virtual communication channel** must be established before data exchange.
- **Bidirectional**: Once connected, **data flows in both directions** between endpoints.
- **Asymmetric**: Designed for the **client-server model**, where each endpoint has a **distinct role**.

User Datagram Protocol (UDP)

- **Connectionless**: No **virtual communication channel** is required before sending data.
- **Unidirectional**: Designed for the **transmission of a single message** from one endpoint to another.
- **Symmetric**: No predefined **roles** for the endpoints, both act equivalently.

TCP Protocol

Types of Sockets in TCP Communication

1. **Listening Socket**
   - **Created by the server** to **wait for incoming connection requests**.
   - Operates in **passive mode**, listening for client connections.
   - **Accepts** a request and spawns a new communication socket.

2. **Communication Socket**
   - **Created by the client** when it **initiates a connection** to the server.
   - **Created by the server** after accepting a connection request, establishing a **virtual communication channel**.
   - Enables **bidirectional data exchange** between client and server.

TCP Client-Server Communication Flow

Client Side

1. **Instantiate Communication Socket**
2. **Connect to Server** (using **server's public address**)
3. **Open Input & Output Streams**
4. **Write Request** to the server
5. **Read Reply** from the server
6. **Close Streams & Communication Socket**

Server Side

1. **Instantiate Listening Socket** (binds to **server's public address**)
2. **Continuously Listen** for client connection requests
3. **When a request arrives:**
   - **Instantiate Communication Socket** for client
   - **Create and Start a Service Proxy Agent**
4. **Within Service Proxy Agent:**
   - **Open Input & Output Streams**
   - **Read Request from Client**
   - **Execute Local Processing**
   - **Write Reply to Client**
   - **Close Streams & Communication Socket**

UDP Protocol

UDP Communication and Socket Types

- **UDP Uses a Single Type of Socket**
  - Unlike **TCP**, **UDP** does **not require a connection** before data exchange.
  - **Messages (datagram packets)** are **sent directly** from the sender to the receiver.

1. **Receiving Socket**
   - **Instantiated by the receiver** at a **specific port**.
   - **Listens** for incoming packets from **multiple sources**.
2. **Sending Socket**
   - **Instantiated by the sender** to **transmit packets**.
   - Can send messages to **multiple destination addresses**.

UDP Communication Flow

Source Side

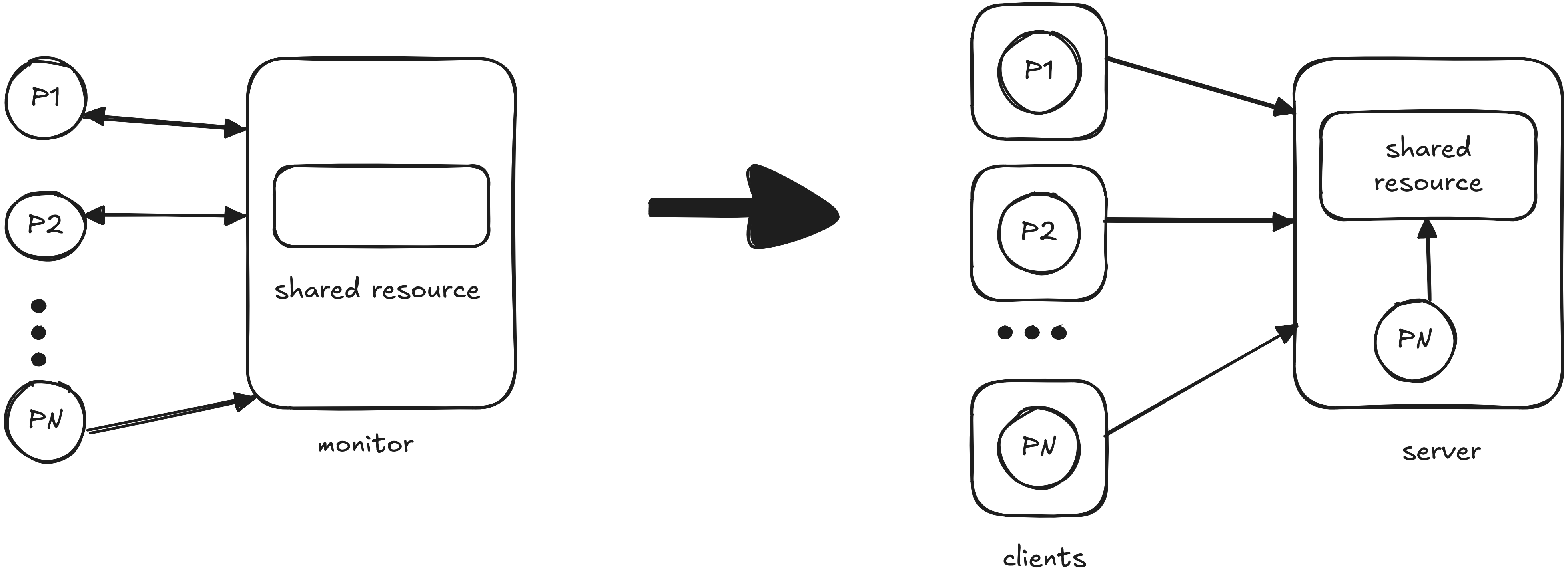
1. **Instantiate Send Socket**
2. **Convert Message to Byte Array**
3. **Instantiate Data Packet** (containing the byte array and destination address)
4. **Send Data Packet** to the receiver

Destination Side

1. **Instantiate Receive Socket** (binds to a public address/port)
2. **Receive Data Packet** from the network
3. **Convert Byte Array to Message**
4. **Process the Received Message**

Transformation Principles

Changes in Concurrent Code for Distributed Execution



Changes in Concurrent Code for Distributed Execution

Key Requirements for Transforming a Concurrent System to a Distributed System:

- **Minimal modifications** should be applied to the **interaction mechanism** among entities.
- The **cooperating processes and shared resource** reside in **different computer systems**, meaning **no shared address space**.

Changes in Concurrent Code for Distributed Execution

Implications for Remote Method Invocation:

1. **Method Invocation via Message Exchange**
   - A **request message** is sent for method invocation.
   - A **response message** is sent back with the return value.

2. **Message Content**
   - Must include **method parameters and return values**.
   - Must also carry **caller process attributes** relevant to execution or modified during execution.

3. **Pass-by-Value Requirement**
   - All parameters must be **passed by value** since **direct memory sharing is not possible**.

Message Transmission

Message Representation and Interpretation

- Messages are **transmitted through communication channels**.
- At the **lowest level**, a message is **represented as an array of bytes**.
- Since the **client and server are separate programs**, the **receiver** must correctly **interpret** the byte array.

Ensuring Proper Message Interpretation

- A message must contain:
  - **Parameter values**
  - **Parameter types**
  - **Data structure information**

Message Transmission

Marshaling and Unmarshaling

- **Marshaling**: The process of **converting parameters and data** into a structured message for transmission.
- **Unmarshaling**: The process of **extracting parameter values** from a received byte array.

Marshaling in Java

- Java **automatically handles marshaling/unmarshaling**.
- Programmers only need to **define message data types** as implementing the **Serializable** interface.

Message Transmission


```java
import java.io.Serializable;

public class Message implements Serializable {
    private static final long serialVersionUID = <long literal>;

    // Definition of message parameters
    private int id;
    private String content;
    private CustomDataType customData; // Reference type

    // Constructor for message instantiation
    public Message(int id, String content, CustomDataType customData) {
        this.id = id;
        this.content = content;
        this.customData = customData;
    }

    // Public methods for retrieving message parameter values
    public int getId() { return id; }
    public String getContent() { return content; }
    public CustomDataType getCustomData() { return customData; }
}
```

Message Transmission

Rules for Serialization in Java

- **All message parameters must be serializable** to enable marshaling.
- **If a parameter is a reference type**, it must also **implement `Serializable`**.
- **Recursive rule**: If a reference type contains **other reference types**, they must also be **serializable** until only **primitive data types** remain.

```java
import java.io.Serializable;

public class CustomDataType implements Serializable {
    private static final long serialVersionUID = 1L;
    private double value;

    public CustomDataType(double value) {
        this.value = value;
    }

    public double getValue() {
        return value;
    }
}
```

Client Architecture

Changes in the Main Thread Execution

- Previously, the **main thread instantiated both**:
  - **Cooperating processes**
  - **Shared resource**
- Now, the **shared resource is located in a different address space** and **cannot be instantiated locally**.

Client Architecture

Remote Reference (Stub) for the Shared Resource

- A **remote reference** (called a **stub**) is instantiated **instead of the actual shared resource**.
- **Stub instantiation parameters**:
  - **Internet address** of the server hosting the shared resource.
  - **Listening port number** of the server.
- Other **instantiation values** must now be **sent via a method invocation on the stub**.

Responsibilities of the Stub

- **Intercept method calls** on the shared resource.
- **Convert method calls** into **message exchanges** with the remote server.
- **Send request messages** to the server.
- **Receive and return responses** to the calling process.

Client Architecture

Data Type for the Main Thread

Most of the **existing code remains unchanged**, except for the following modifications:

- **Stub Instantiation**
  - The **stub of the shared resource** is instantiated **instead of the shared resource itself**.
- **Passing Additional Initialization Parameters**
  - Any **extra values** required for **instantiating the shared resource** are now passed **through a new method on the stub**.
- **Server Shutdown Handling**
  - If **server shutdown** is required at the **end of operations**, a **shutdown method** must be invoked on the **stub**.

Data Type for Cooperating Processes

The **code structure remains mostly unchanged**, with the following modifications:

- **Reference to the Stub**
  - Instead of passing a **reference to the shared resource**, a **reference to the stub of the shared resource** is passed upon instantiation.

Definition of the Stub for the Shared Resource

A **new data type** must be created for the **stub of the shared resource**. This **stub** acts as a **proxy** for remote method invocation and follows a **structured sequence of operations**.

Client Architecture

Operations in the Stub

For each **method invocation**, the following steps are performed:

1. **Open a Communication Channel**
   - A connection to the **server** is established (e.g., using **TCP sockets**).
2. **Create an Outgoing Message**
   - Construct a message containing:
     - **Method identification**
     - **Method parameters**
     - **Caller process attributes** (relevant to method execution)
3. **Send the Outgoing Message**
   - Transmit the **service request** to the server.

Client Architecture

4. **Receive and Validate Incoming Message**
   - Wait for the **server's response**.
   - Validate the **reply message** for correctness.
5. **Update Caller Process Attributes**
   - Modify **affected attributes** based on **method execution results**.
6. **Close the Communication Channel**
   - Terminate the connection once the interaction is complete.
7. **Return Method Results**
   - Provide the **final result** to the caller.

Client Architecture

Data Type for the Communication Channel

- **A new data type must be created** to encapsulate **socket operations.**
- Its key role is to **simplify network communication** by handling:
  - **Connection setup**
  - **Message transmission**
  - **Message reception**
  - **Connection termination**

Data Type for Messages

- **A new message data type must be created** to define the structure of exchanged messages.
- There are **two possible design choices**:
  1. **Single Message Data Type** – One data type that covers **all communication cases**.
  2. **Multiple Message Types** – Different message data types, each tailored for **specific communication scenarios**.

Server Architecture

Base Thread (Main Server Thread)

- The **shared resource is passive** and must be **instantiated by the base thread**.
- A **communication channel** (socket) is created to **listen for service requests** on a **public address**.
- When a **service request** arrives:
  1. A **service proxy agent thread** is instantiated.
  2. The base thread **resumes listening** for new requests.
  3. This enables **server replication**, allowing multiple client requests to be handled concurrently.

Server Architecture

Service Proxy Agent (Handles Individual Requests)

- **Receives the incoming message**.
- **Decodes** the message and **extracts process attributes** from the client.
- **Sets itself as a client clone** by incorporating the required attributes.
- **Invokes the corresponding method** on the shared resource.
- **Creates an outgoing message** with the response data.
- **Sends the response** back to the client.
- **Closes the communication channel** and **terminates itself**.

Data Type for the Main Thread (Server Base Thread)

- **New and must be created** but remains **mostly invariant** across servers.
- **Modifications required:**
  - **Public address** for service requests must be **specific to each server.**
  - **Shared resource and its interface** must be instantiated **based on the server's requirements.**

Data Type for the Service Proxy Agent Thread

- **New and must be created**, but remains **largely uniform** across servers.
- **Modifications required:**
  - To allow it to act as a **clone of multiple client classes**, it must:
    - **Implement interfaces** for each client class.
    - Provide methods for **setting and getting relevant client attributes**.

Server Architecture

Data Type for the Interface to the Shared Resource

- **New and must be created**, but remains **invariant** across servers.
- **Internal Structure:**
  - Contains **only one public method**:
    - `processAndReply()` – Handles **decoding, processing, and replying** to service requests.
  - **Internal Operation Steps:**
    1. **Incoming Message Validation:**
       - Ensures correctness.
       - Incorporates **client attributes** when needed.
    2. **Processing and Response Generation:**
       - Invokes the appropriate **shared resource method**.
       - Creates an **outgoing message (reply)** for the client.

Server Architecture

Data Type for the Shared Resource

- **Remains mostly unchanged**, with a minor modification:
  - **References to cooperating processes** must be **updated** to use the **service proxy agent data type** instead.

Data Type for the Communication Channel

- **New and must be created** to **encapsulate socket operations**.
- **Key features:**
  - Handles **socket creation and management**.
  - Supports **sending and receiving messages** between the server and clients.
  - Manages **connection lifecycle** (opening, maintaining, and closing connections).

Server Architecture

Data Type for Messages

- **The same data type is used on both the client and server sides.**
- Ensures **consistent message structure** for:
  - **Serialization (marshaling/unmarshaling)**
  - **Parameter passing**
  - **Response formatting**

Mixed Architecture

Servers Acting as Both Clients and Servers

Key Concept

- In **distributed applications**, multiple servers are often involved.
- Some servers **provide services** while also **requesting services from other servers**.
- This results in **servers functioning as both clients and servers simultaneously**.

Mixed Architecture

Implementation Approach

- **No conceptual complexity arises**—both roles can be **merged seamlessly**.
- The system follows a **mixed architecture**, where:
  - Each **server contains both client and server functionalities**.
  - The **server role** handles **incoming requests**.
  - The **client role** sends **requests to other servers** when needed.

Mixed Architecture

Structural Adjustments

1. **Each server instantiates:**
   - A **listening socket** for handling client requests.
   - A **communication module** for making outgoing requests to other servers.
2. **Message formats remain consistent** across both roles.
3. **Concurrency management** is required to handle **simultaneous client and server operations**.