# Sistemas Distribuídos

## Remote Objects

Eurico Pedrosa

António Rui Borges

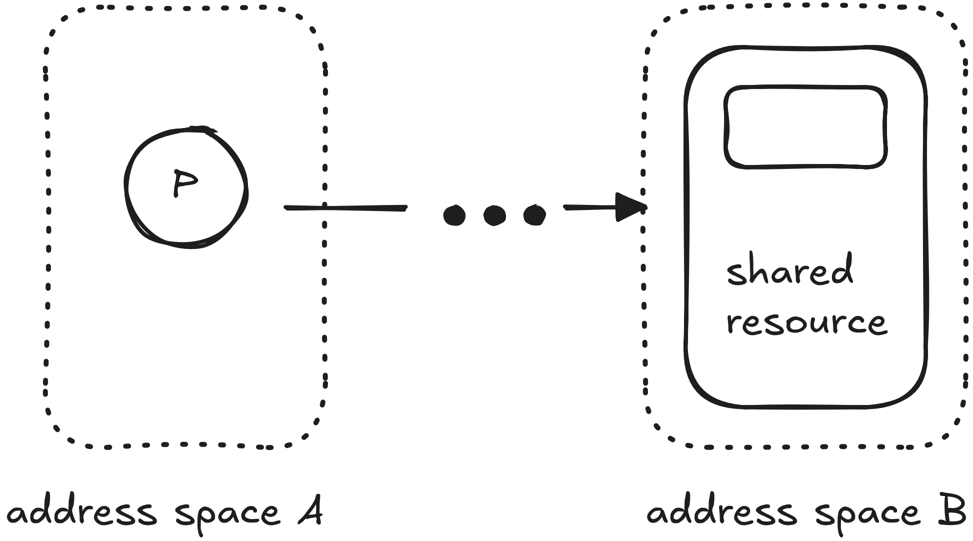
Universidade de Aveiro - DETI

2025-03-26

# Remote Procedure Call (RPC)

# Addressing Space Separation






In a **Remote Procedure Call (RPC)**, the **calling process** and the **shared resource** reside in **different addressing spaces**, typically on different machines. This architectural separation leads to the following requirements:

# Addressing Space Separation



**In Addressing Space A (Client Side):**

- A **reference to the shared resource** must be:
  - **Obtained before invocation**.
  - This reference is usually represented as a **stub** or **proxy**, which encapsulates the communication logic.
  - The stub must be **configured with the server's public address and port** to enable remote interaction.

# Addressing Space Separation



**In Addressing Space B (Server Side):**

- A **reference to the shared resource** must be:
  - **Generated** on the server.
  - **Made available** to other systems (typically clients).
  - This is usually done by **binding the reference** to a **known public address and port**, or by **registering it with a naming service**.

# Addressing Space Separation



## Key Implication:

- Because there is **no shared memory**, **all interactions** (including method invocation and parameter passing) must occur via **explicit message exchanges**.

# Remote vs Local Procedure Calls



While **Remote Procedure Calls (RPCs)** are designed to resemble **local procedure calls**, there are important distinctions that must be considered when building distributed systems

# Remote vs Local Procedure Calls



## 1. Possibility of Failure

- A remote call **may fail even if the application code is correct** if:
  - The **remote shared resource** not being instantiated or temporarily unavailable.
  - **Communication infrastructure** issues (e.g., network failures, timeouts, unreachable server).
- Unlike local calls, **remote calls depend on external systems**, introducing **inherent unreliability**.

# Remote vs Local Procedure Calls



### 2. Pass-by-Value Requirement

- Since the caller and the callee reside in **different address spaces**, all:
  - **Procedure parameters**
  - **Return values (if any)** must be **passed by value**.
- This necessitates:
  - **Marshaling** (serialization) at the source.
  - **Unmarshaling** (deserialization) at the destination.
- Parameter types and structure must be **fully defined and serialized** for transmission.

# Remote vs Local Procedure Calls



**3. Higher Execution Latency**

- **Remote procedures are slower than local ones** because:
  - A **communication mechanism** (e.g., socket messaging) is involved.
  - **Request and response messages** must be exchanged.
- This introduces **non-negligible delays** due to network latency and message processing overhead.

# Naming



In a **distributed system that uses RPC**, a **naming service** is essential to allow applications to discover and connect to remote shared resources in a **transparent and dynamic way**.

# Naming



### Purpose of the Naming Service

- Acts like a **dynamic telephone directory**:
  - Maps a **publicly known name** of a shared resource
  - To its **location and access details** (e.g., network address, port)
- Enables **network transparency** for application programmers:
  - Only the **naming service address** and the **resource name** are needed to establish remote communication.
  - No need to hardcode or manually manage IP addresses and ports.

# Naming



## Remote Reference Contents

The **remote reference** obtained via the naming service typically includes:

- **Internet address** of the host platform.
- **Port number** used for communication.
- **Procedure signatures** (methods that can be invoked remotely).
- **Type information** describing the **parameters and return values** (e.g., the names of interface or class files used for marshaling/unmarshaling).

# Naming



### Benefits for the Programmer

- The programmer can **treat remote resources as local** (network transparency).
- The complexity of **resource discovery and binding** is delegated to the **naming service**.
- Facilitates **scalability and modularity** by decoupling service name from physical location.

# Architecture of RPC

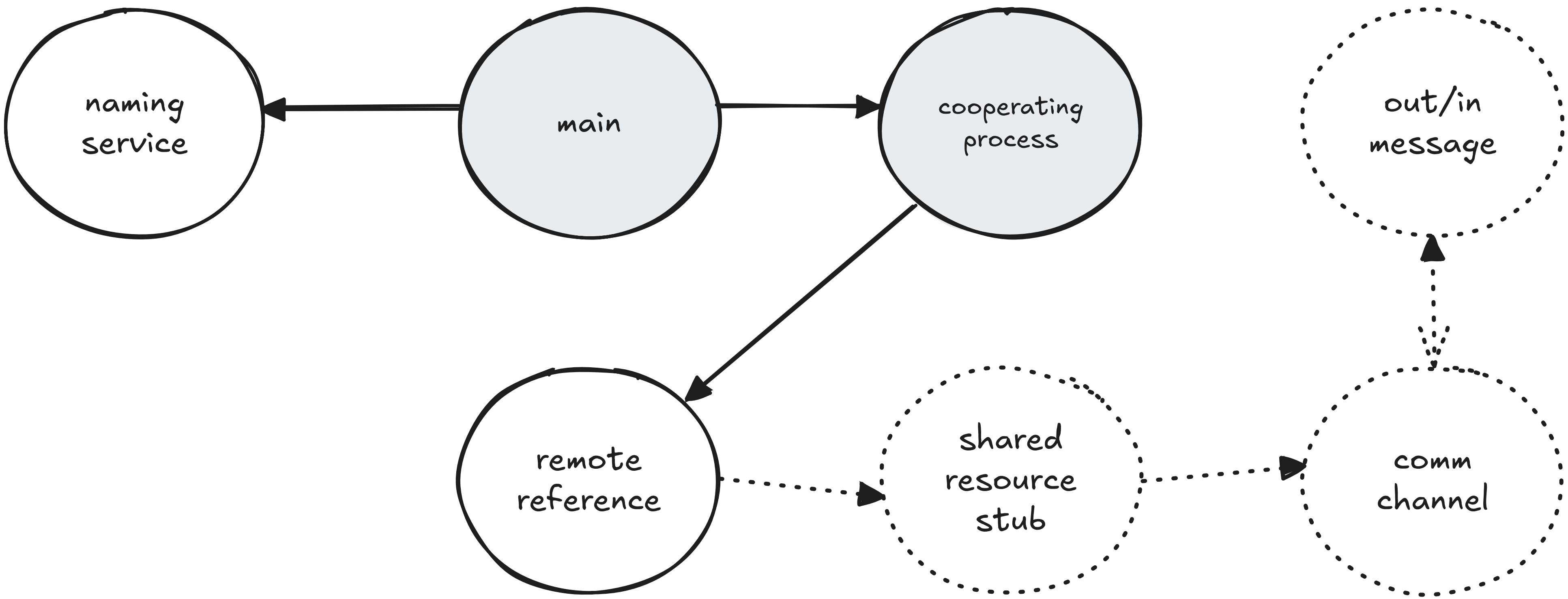
# Code Responsibility



In an **RPC environment**, the development effort is **divided** between the **application programmer** and the **underlying system (runtime environment or middleware)**:

# Code Responsibility

## Addressing Space A



# Code Responsibility



## Code Written by the Application Programmer

represented by continuous lines in diagrams

- Includes logic for:
  - **Main threads** (e.g., servers and clients)
  - **Shared resources**
  - **Service proxy agents**
  - **Communication channel wrappers** (if customized)
- The programmer must also:
  - Provide the **signature of all remotely invocable procedures**.
  - Define the **data types** used for **procedure parameters and return values** (e.g., implementing `Serializable` in Java).

# Code Responsibility



## Code Automatically Generated by the Environment

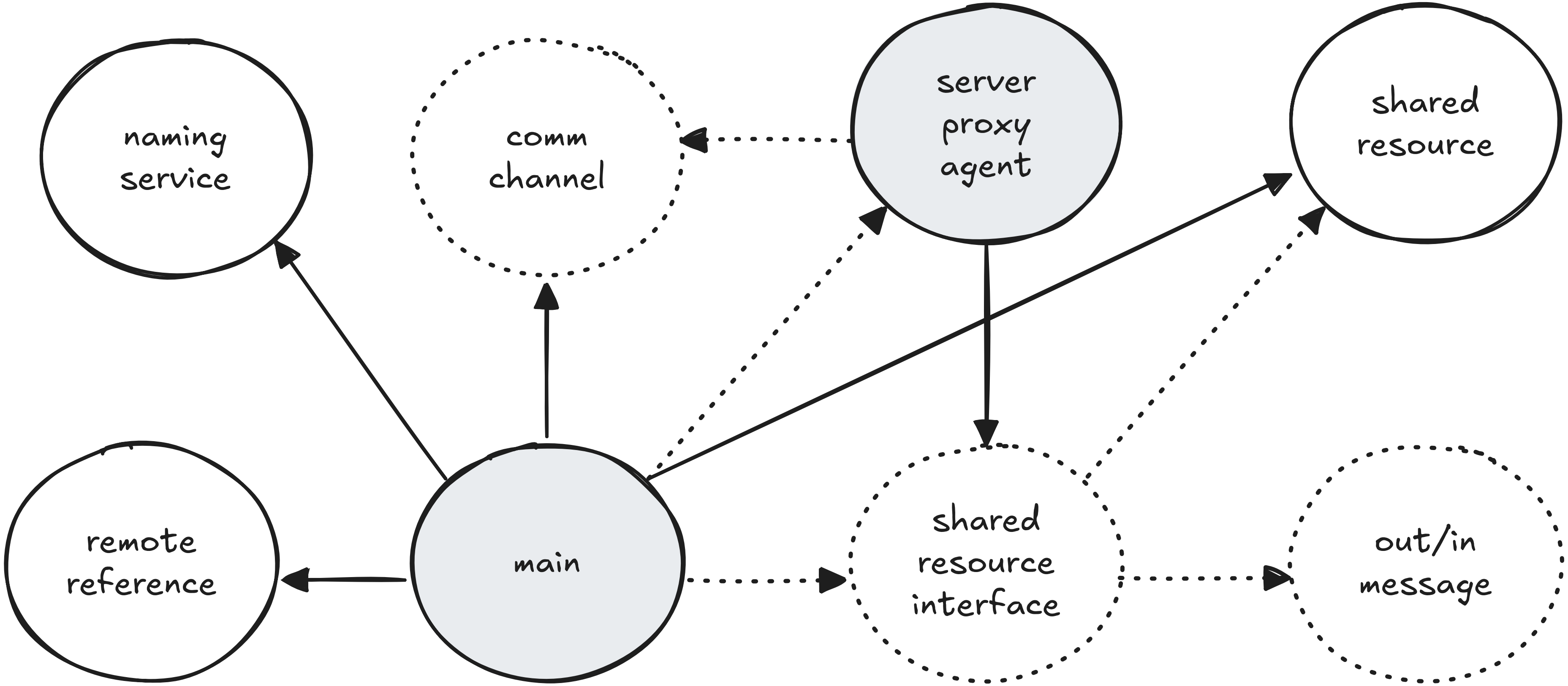
represented by dashed lines in diagrams

- Includes the creation of:
  - **Stubs (client-side proxies)**
  - **Skeletons or dispatchers (server-side handlers)**
  - **Marshaling and unmarshaling code** for message formatting
- This code is generated **transparently** by the middleware (e.g., RMI, gRPC) based on the **interface definitions and type descriptors** provided by the programmer.

# Code Responsibility



## Addressing Space B



# Code Responsibility



## Code Written by the Application Programmer

represented by continuous lines in diagrams

- Must define:
  - The **application logic** (client, server, main thread, shared resource, proxy agents)
  - The **interface of the shared resource**, including:
    - The **signatures of all remote procedures**
    - The **data types** used as procedure parameters and return values (e.g., classes implementing `Serializable`)

# Code Responsibility



## Code Automatically Generated by the Environment

represented by dashed lines in diagrams

- Is **automatically generated and transparent** to the programmer.
- Includes:
  - **Stubs** (client-side proxies for method calls)
  - **Skeletons** (server-side dispatchers that invoke real object methods)
  - **Marshaling/unmarshaling logic** for converting data to/from byte streams

# Code Responsibility



## The Skeleton

- The **collection of entities represented by dashed lines** is called the **skeleton**.
- It runs **independently of the application's main thread**.
- **Execution Model:**
  - When a **remote reference is generated**, the **base thread** associated with it is **instantiated and started**.
  - This allows the **skeleton to handle remote invocations** without blocking or interfering with the main logic of the application.

# Code Migration

**Code migration** is a valuable feature in distributed systems that allows parts of a running application to be **moved between processing nodes** at runtime. This capability introduces **flexibility, performance optimization**, and **resilience**.

# Benefits of Code Migration



## Load Balancing and Performance Optimization

- Some **processing nodes have more computation power** or specialized resources.
- Migration enables **dynamic relocation** of code to those nodes to maximize efficiency.
- This is especially beneficial in **heterogeneous systems** where not all nodes are alike.

# Benefits of Code Migration

## Fault Tolerance and System Resilience

- If a **node fails or is about to fail**, the system can:
  - **Detect the malfunction**
  - **Reassign the affected software components** to healthy nodes
- This helps **prevent application crashes** and **maintain service continuity**.

# Benefits of Code Migration

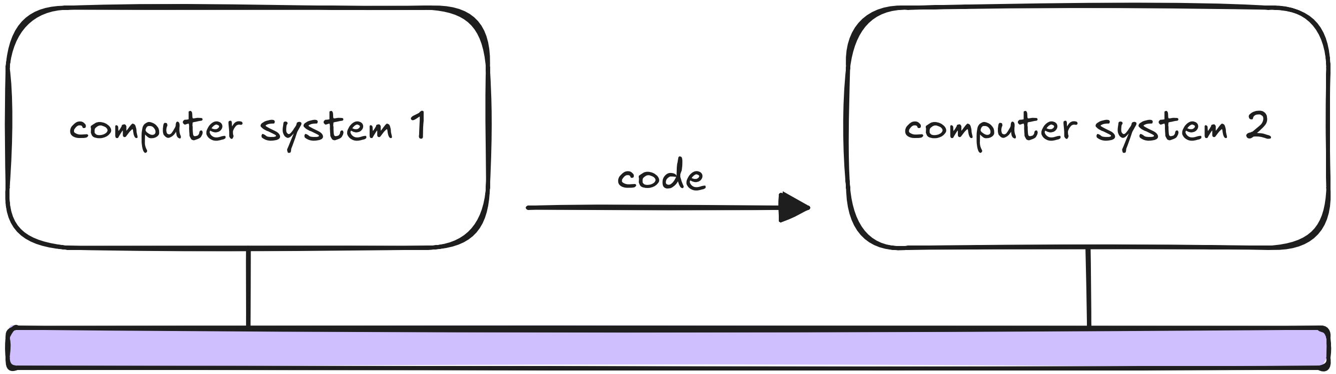


### Mechanisms Involved in Code Migration

- To support code migration, the system must handle:
  - **Code state serialization and transfer**
  - **Rebinding of dependencies** (e.g., references, resources, environment)
  - **Resumption of execution** in a consistent state

# Form of Code






When implementing **code migration** in a distributed system, an important design decision is **what form the migrating code should take**. The chosen form directly impacts **portability, compatibility, and execution strategy**.

# Form of Code



## Executable Code

- **Description**: Compiled machine code ready for direct execution.
- **Pros**:
  - **Fast execution** – no compilation or interpretation needed.
  - Minimal runtime overhead.
- **Cons**:
  - Requires **similar hardware and operating systems** on the source and destination nodes.
  - **Limited portability**.

# Form of Code



## Source Code

- **Description**: The original human-readable code is transferred and compiled on the destination node.
- **Pros**:
  - **Highly portable** – no assumptions about platform similarity.
  - Can adapt to **heterogeneous systems**.
- **Cons**:
  - Requires a **compiler or interpreter** on the destination node.
  - **Compilation time** adds overhead.
  - Potential **security risks** (e.g., executing unverified code).

# Form of Code



## Intermediate Code

- **Description**: Platform-independent bytecode interpreted or just-in-time compiled at runtime.
- **Typical Example**: Java bytecode run on the **Java Virtual Machine**.
- **Pros**:
  - **Good balance** between portability and efficiency.
  - Execution environments (e.g., JVM, .NET CLR) handle most compatibility concerns.
- **Cons**:
  - Requires an **interpreter or virtual machine** on the destination.
  - May introduce some **performance overhead** compared to native execution.

# Security Concerns



While **code migration** adds significant **flexibility and resilience** to distributed systems, it also introduces **serious security risks**, particularly when **code is received from untrusted or external sources**.

# Security Concerns



## Security Challenge

- When **code is transferred to a new node**, it may:
  - Attempt to **access local resources** (files, memory, network).
  - Contain **malicious logic** that compromises the system.
- It is essential to **ensure the integrity, confidentiality, and availability** of the host system's resources.

# Security Concerns



## Typical Solution: Security Manager

- A **dedicated component**, commonly known as the **security manager**, is introduced in the migration system.
- **Responsibilities of the Security Manager**:
  - **Monitor all resource access requests** from the incoming code.
  - **Decide**, on a **case-by-case basis**, whether access should be **allowed or denied**.
  - Enforce **security policies** defined by the system administrator or platform.

# Security Concerns



## Common Approaches

- **Java's SecurityManager** (legacy concept) monitors actions like file I/O, network access, and reflection.
- **Sandboxing** and **capability-based systems** restrict the operations that incoming code can perform.
- **Digital signing and verification** ensure that only **trusted, authenticated code** is executed.

# Suggested Reading

- M. van Steen and A.S. Tanenbaum, Distributed Systems, 4th ed., distributed-systems.net, 2023.