Computação em Larga Escala

Message Passing

Eurico Pedrosa

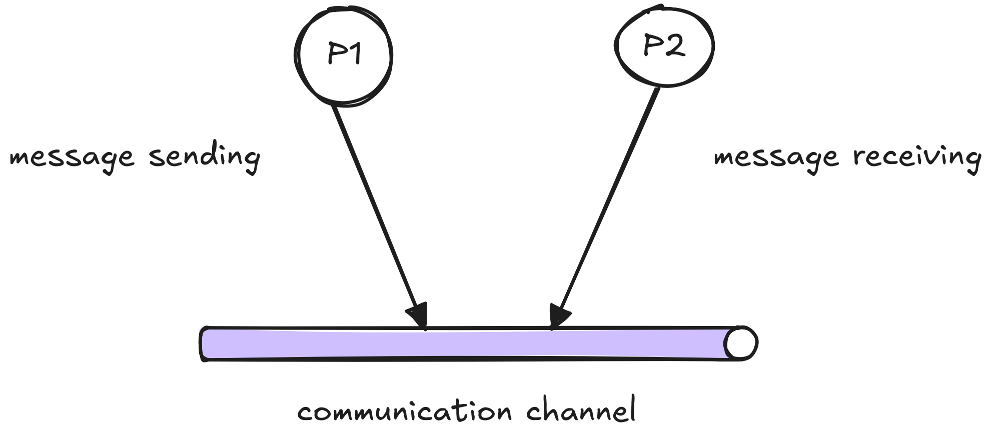
António Rui Borges

Universidade de Aveiro - DETI

2025-03-16

General Principle

Message Exchange



Message exchange is a flexible communication method that does not require a shared address space. It works consistently across single-processor, multiprocessor, and distributed environments.

- A **forwarder** P_F sends a message through a **communication channel**
- A **recipient** P_R accesses the channel and waits for the message (receiving operation)

Synchronization

Synchronization in Message Exchange

For reliable communication, **synchronization** between the **forwarder** and **recipient** is required. There are two main types:

- **Non-blocking synchronization** – Processes manage synchronization themselves.
  - **Sending**: Forwards the message and returns immediately, without confirmation of reception.
  - **Receiving**: Always returns, regardless of whether a message was received.

```
/* sending operation */
void msgSendNB (unsigned int destid, MESSAGE msg);
/* receiving operation */
void msgReceiveNB (unsigned int srcid, MESSAGE *msg, bool *msg_arrival);
```

Synchronization in Message Exchange

- **Blocking synchronization** - Message exchange ensures built-in synchronization.
  - **Sending**: Blocks until the message is received.
  - **Receiving**: Blocks until a message arrives.

```c
/* sending operation */
void msgSend (unsigned int destid, MESSAGE msg);
/* receiving operation */
void msgReceive (unsigned int srcid, MESSAGE *msg);
```

Types of Blocking Synchronization

1. **Rendezvous** – Both processes reach an exchange point before transferring the message. No intermediate storage is needed. Common in **point-to-point connections**.
2. **Remote** – The sender blocks until confirmation of reception. May involve **intermediate storage** and is typical in **shared communication channels**.

Addressing Types

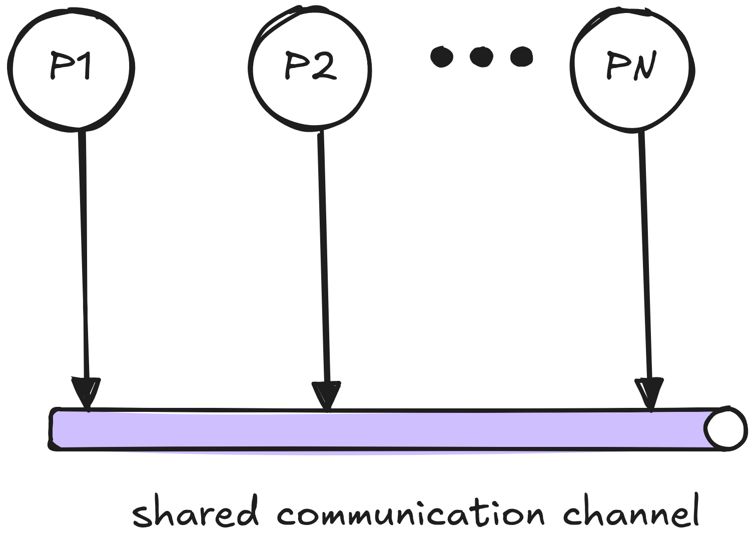
Message Addressing and Communication Channels

For message exchange, the **sender** and **receiver** must identify each other:

- **Direct addressing** – The sender explicitly references the recipient.
- **Indirect addressing** – The communication channel is specified instead.

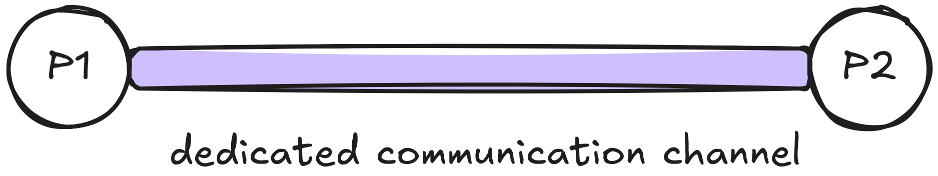
Some channels support **intermediate storage**, forming **mailboxes** that queue messages in chronological order.

Message Addressing and Communication Channels



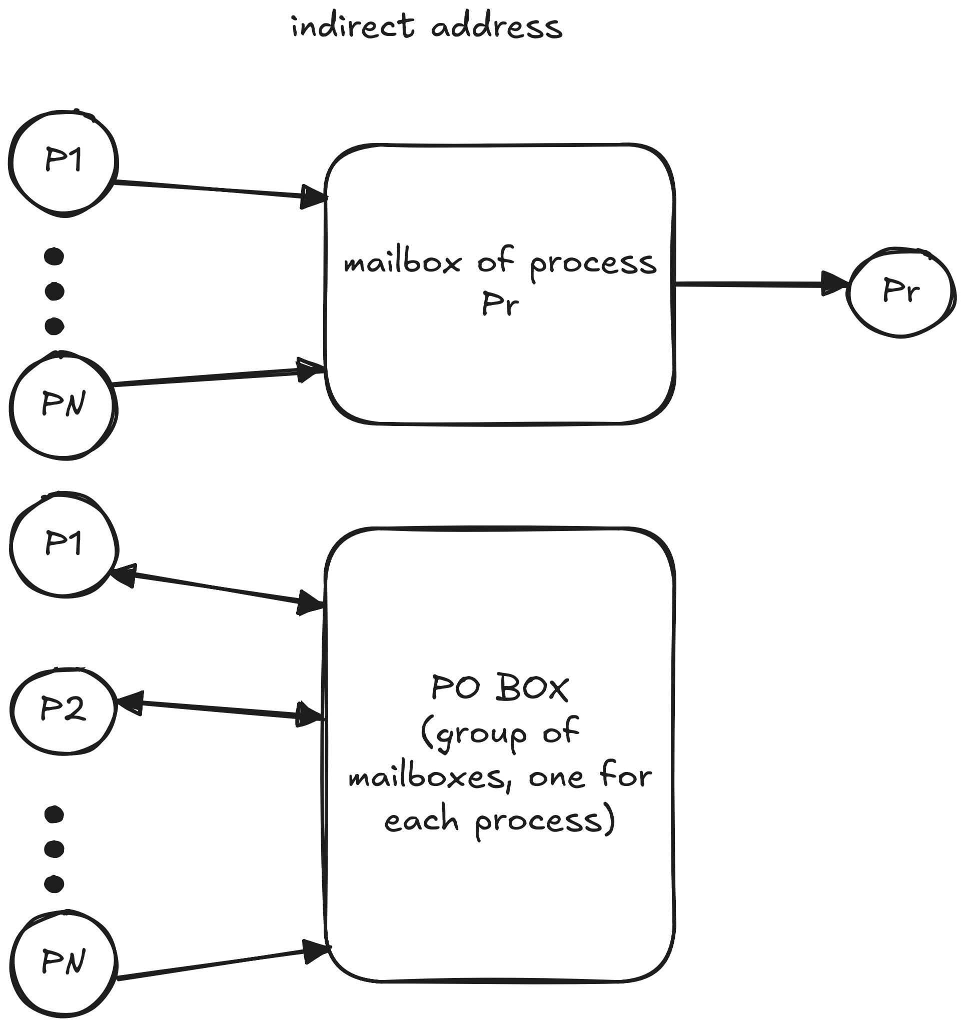
direct address - if based on process id

indirect address - if based on access port



indirect address

Message Addressing and Communication Channels



Communication Types

Communication Contexts

- **One-to-One** – Message exchange between a **forwarder** and a **recipient**.
- **One-to-Many** – Message sent to multiple recipients:
  - **Broadcast** – Sent to all processes in the application.
  - **Multicast** – Sent to a specific group of processes.

Composite Message

- **Scatter** – A **one-to-many** communication where a composite message is split into exclusive parts, each sent to a different recipient.
- **Gather** – A **many-to-one** communication where a composite message is formed by merging parts received from multiple senders.

Producer/Consumer

Producer

```
/* Mailbox-based message exchange
   - A shared mailbox (capacity: K messages) is accessible by all producers (forwarders)
and consumers (recipients). */

static unsigned int com;  // Mailbox identifier

typedef struct {
    DATA info;
} MESSAGE;  // Message structure

/* Producer Process */
void main (unsigned int p) {
    MESSAGE msg;

    while (1) {  // Infinite loop
        produceValue(&msg.info);
        msgSendNB(com, msg);  // Non-blocking send; blocks if mailbox is full
        doSomeThingElse();  // Continue other tasks
    }
}
```

Consumer

```
/* Mailbox-based message exchange
   - A shared mailbox (capacity: K messages) is accessible by all producers (forwarders)
and consumers (recipients). */

static unsigned int com;  // Mailbox identifier

typedef struct {
    DATA info;
} MESSAGE;  // Message structure

/* Consumer Process */
void main (unsigned int c) {
    MESSAGE msg;

    while (1) {  // Infinite loop
        msgReceive(com, &msg);  // Waits for a message
        consumeValue(msg.info);
        doSomeThingElse();  // Continue other tasks
    }
}
```