

Message Passing Systems - Introduction

Alessio Bechini Dept. of Information Engineering, Univ. of Pisa

alessio.bechini@unipi.it

Outline

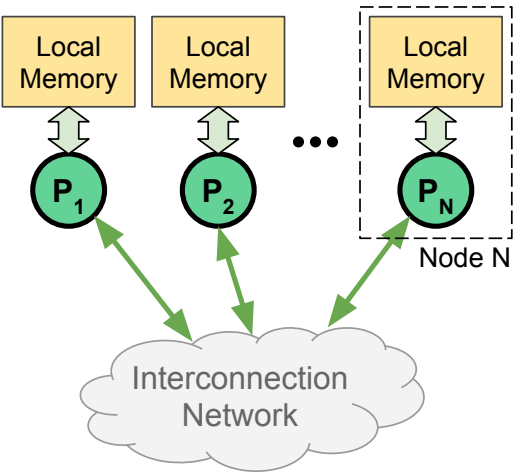
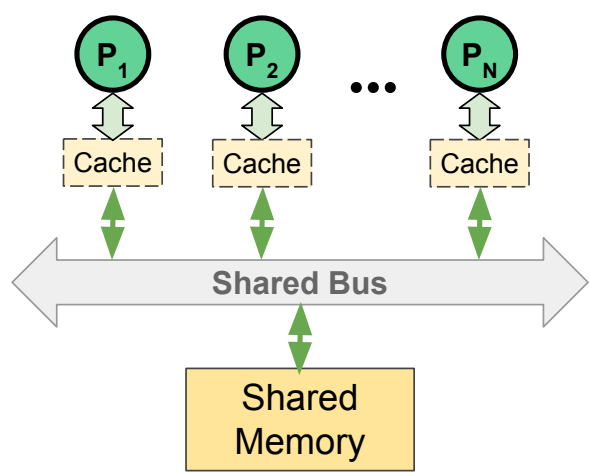
Introduction to models and constructs to design/develop distributed systems.

- Models for Distributed Systems
- Addressing of Processes
- Semantics for Send/Receive

Models for Distributed Systems

© A. Bechini 2023

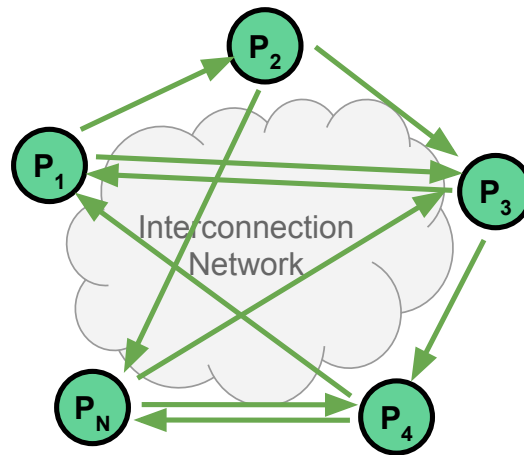
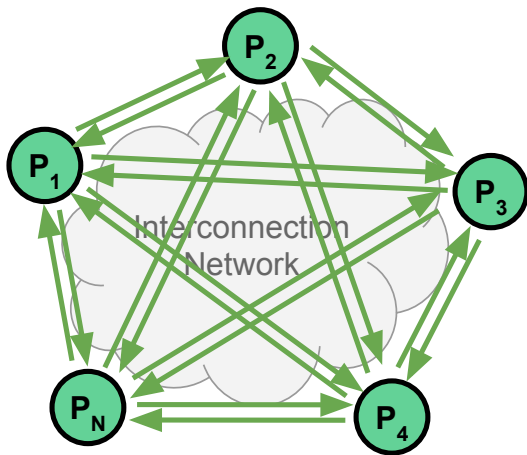
From Shared Memory to Msg Passing



Not only in distributed settings!

© A. Bechini 2023

Interconnections: Channels



Distributed Systems
≠
Networking!!!

Interconnection: fully vs partially connected.

Channels: *mono*- vs *bi*-directional, FIFO vs non-FIFO (msg ordering)

Types of Models

We can work with different type of models, to characterize different aspects of the target systems at different levels of abstraction:

physical - representation of the underlying hardware elements of a distributed system, abstracting away details of the computer and networking technologies.

architectural - system structure in terms of components and their relationships (eg client-server, peer-to-peer) - use of middleware.

fundamental (abstract) - accounts for the fundamental properties of the basic system characteristics, as well as its potential failures.

Fundamental Models

Contain only essential ingredients to understand and reason about some aspects of a system's *behavior*. The purpose is:

- To make explicit all the **relevant assumptions** about the modelled system.
- To make generalizations on what it is possible or impossible, under such assumptions.

Generalizations: general-purpose algorithms, or desirable, guaranteed properties. Guarantees depend on logical analysis and, possibly, mathematical proof. Basic behavioral aspects are: Interaction, failure, security.

How to Pass Information?

Idea: make use of *messages*. Content, or “Payload:” *what* to pass from the local memory of one process to the local memory of another.

Additional info to be possibly provided:

- Further semantics, with message *types* or attached *tags* (how to deal with a received message? What does it mean?)
- Metadata (what kind of stuff in the payload? How to interpret it?)
- Who is the sender?
- ...

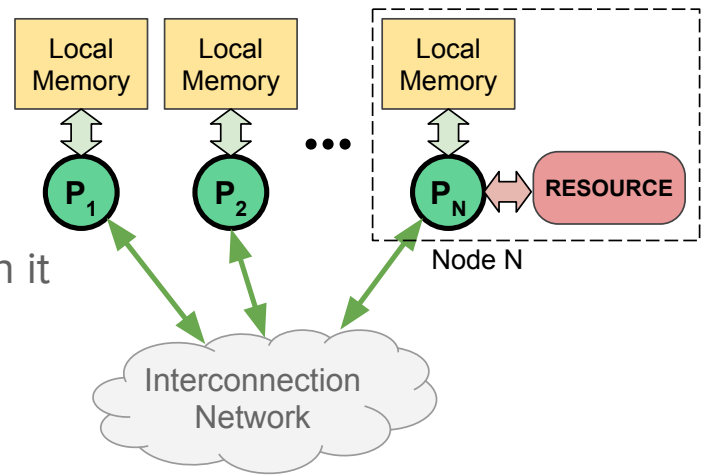
Primitives for message exchange: **send** and **receive** - addresses!

Resource Management

In a message passing system, a resource can be paired with a process able to interact with it.

If the resource is intended to be shared across processes, the process paired with it has to behave as its *manager*.

Any access to the resource have to be mediated by communication with the relative manager.



Process Addressing

Problem: Process/Node Addressing

A process in a distributed application has to be uniquely identified *for the purposes of the application*.

Moreover, it has also to be *found*, so messages can be delivered. The use of communication infrastructure (either concrete or abstract) requires an *addressing system*.

Often, specific IDs are sufficient as logic addresses, but under the hood they need to be somehow mapped onto the real addresses used by the underlying communication network.

© A. Bechini 2023

A. Bechini - UniPi

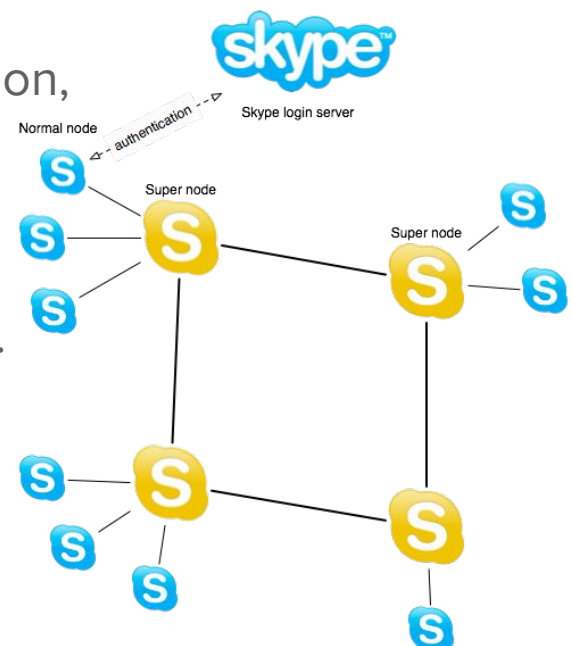
Overlay Networks

A practical approach consists in the definition, at the application level, of a communication network *on top of a real one*.

The result is known as an Overlay network.

Examples over the Internet:

- A client-server system
- Peer-to-peer applications, like Skype



© A. Bechini 2023

A. Bechini - UniPi

Explicit vs Implicit Addressing

So far we have discussed the problem of identifying one specific destination or source.

Indicate
what you want,
not who will do it!

In transmitting information, *implicit addressing* can be sometimes used instead.

E.g., a certain “service” can be provided as destination, without indicating the actual process that will carry out the job.

Distributed systems of this kind asks for adequate, complex communication support, and will be discussed later.

Addresses, Endpoints, and Ports

Different messages may be sent to one single process for many different reasons.

To facilitate their handling, multiple *communication endpoints* can be associated with one process/node.

A communication endpoint can be specified by the process/node address, + another identifier known as *port*.

Ports are a means to further specify the semantics of a message, making it easier its handling.

Getting More Complicated

Further challenges in addressing:

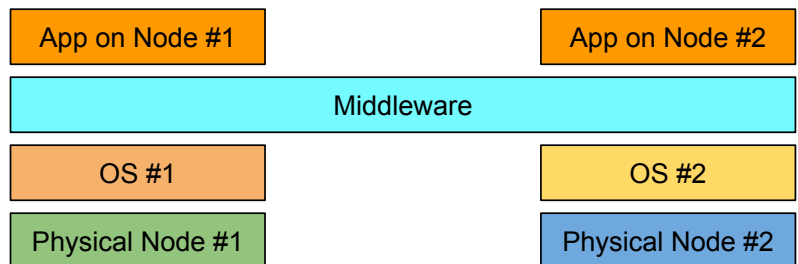
- Considering Broadcast
- Considering Multicast
- Considering dynamic membership in process groups
- ...

Introducing Middleware

The implementation of communication constructs typically hides networking details: this can be done using **Middleware**

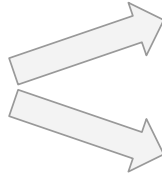
i.e. dedicated software on top of the operating system
that provides the required communication services.

Middleware can enable
interoperability
across applications
running on different OSs.



Middleware: How?

Possible ways to provide
Communication Services:



Libraries



*Runtime
Support*

Using (standard) middleware components is typically preferable to writing custom code, because it is usually the most cost-effective solution to support integration of software parts - especially distributed ones.

Semantics of Send/Receive

Abstract Models: Synch vs Asynch

Different perspectives, depending on how operations in different processes are *thought* to be executed with respect to one another.

Two basic types of abstract models:

- **Synchronous** - processes take steps simultaneously, i.e., the execution proceeds in synchronous rounds.
Simple model to reason about, but not realistic
- **Asynchronous** - no assumption on relative timings; unbounded delays.

Basic characteristic: ***lack of a global clock***

Semantics for Send/Receive

Possible behaviours of send/receive primitives:

- **Blocking** - at execution, the process possibly blocks to wait for the operation to be completed
- **Non-blocking** - the operation is just “fired” and the process keeps on executing; it requires *buffering* of messages.

Language primitives:
Smallest
units of processing

The implementation of such primitives typically requires the use of *sending and receiving queues* (a.k.a. “*mailboxes*”), i.e. buffers to host *out-going* and *in-coming* messages.

Blocking vs Non-Blocking Send

- **Blocking** - the message is sent towards destination, and the process can resume execution only upon getting sure that the message has been received. *It contains an implicit handshake.*
- **Non-blocking** - the message is sent towards destination, and the sender immediately goes on with the following instructions.

What does it mean “message sent” ?



Placed in the outgoing buffer/queue?

Or that the buffer has been flushed, and can be used again?

Blocking vs Non-Blocking Receive

- **Blocking** - the process waits for an incoming message to be available on an endpoint (possibly an incoming queue).
- **Non-blocking** - the process checks if an incoming message is available, and possibly reads it. Returns indication of success/failure.

Pay attention: the use of specific semantics in blocking/non-blocking pairs may lead to deadlock!

Early proposed construct: *Guarded command*

<guard> → <instruction> where the guard part is

< a condition plus a blocking receive operation.>

Example: Scheme for Bounded Buffer

Idea (non-blocking send, blocking receive, addressing with ports):
ask for permit to insert an item, and get an item when it is available.

