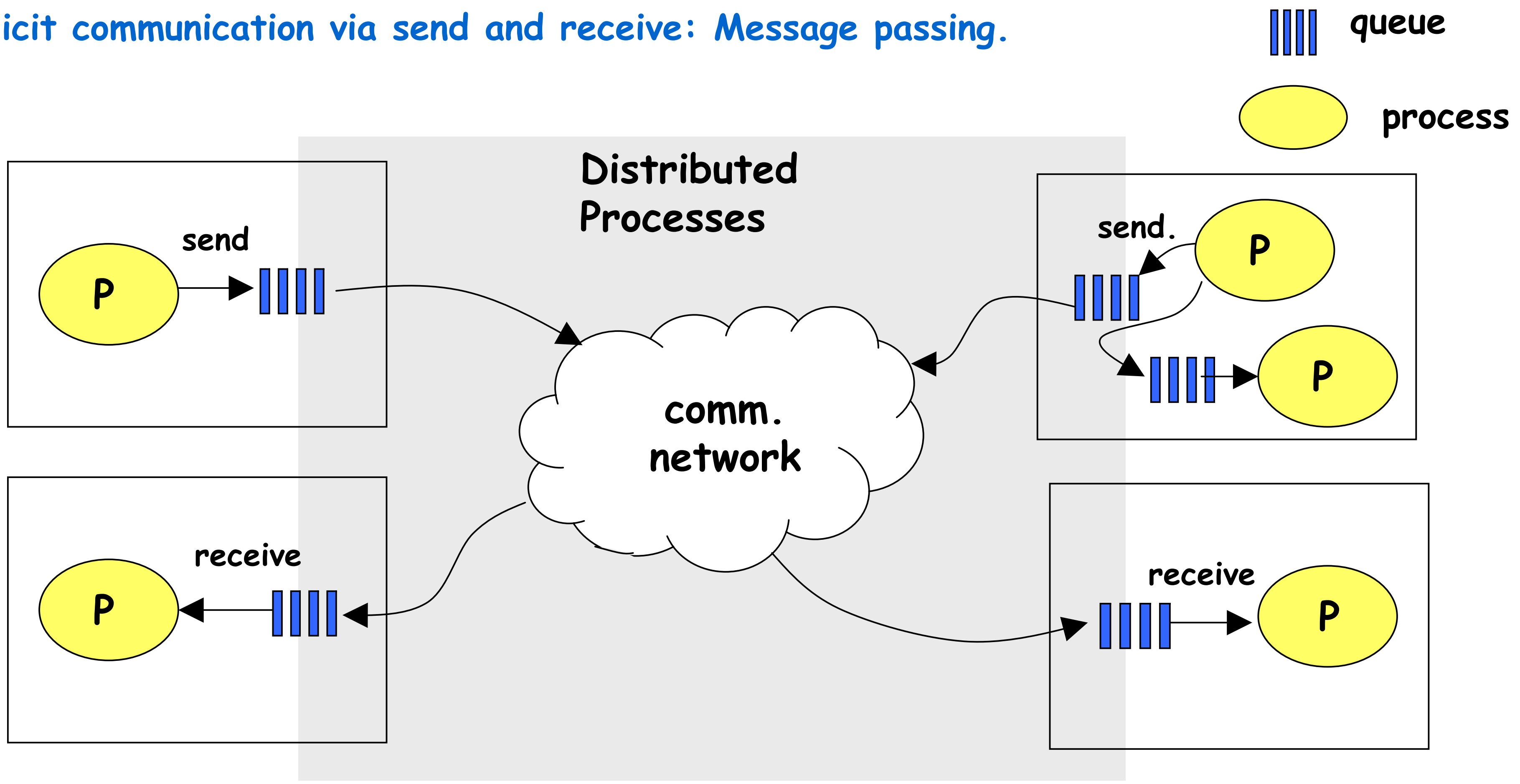
Operating Systems II

IPC
Inter Process Communication

Principles of distributed computations

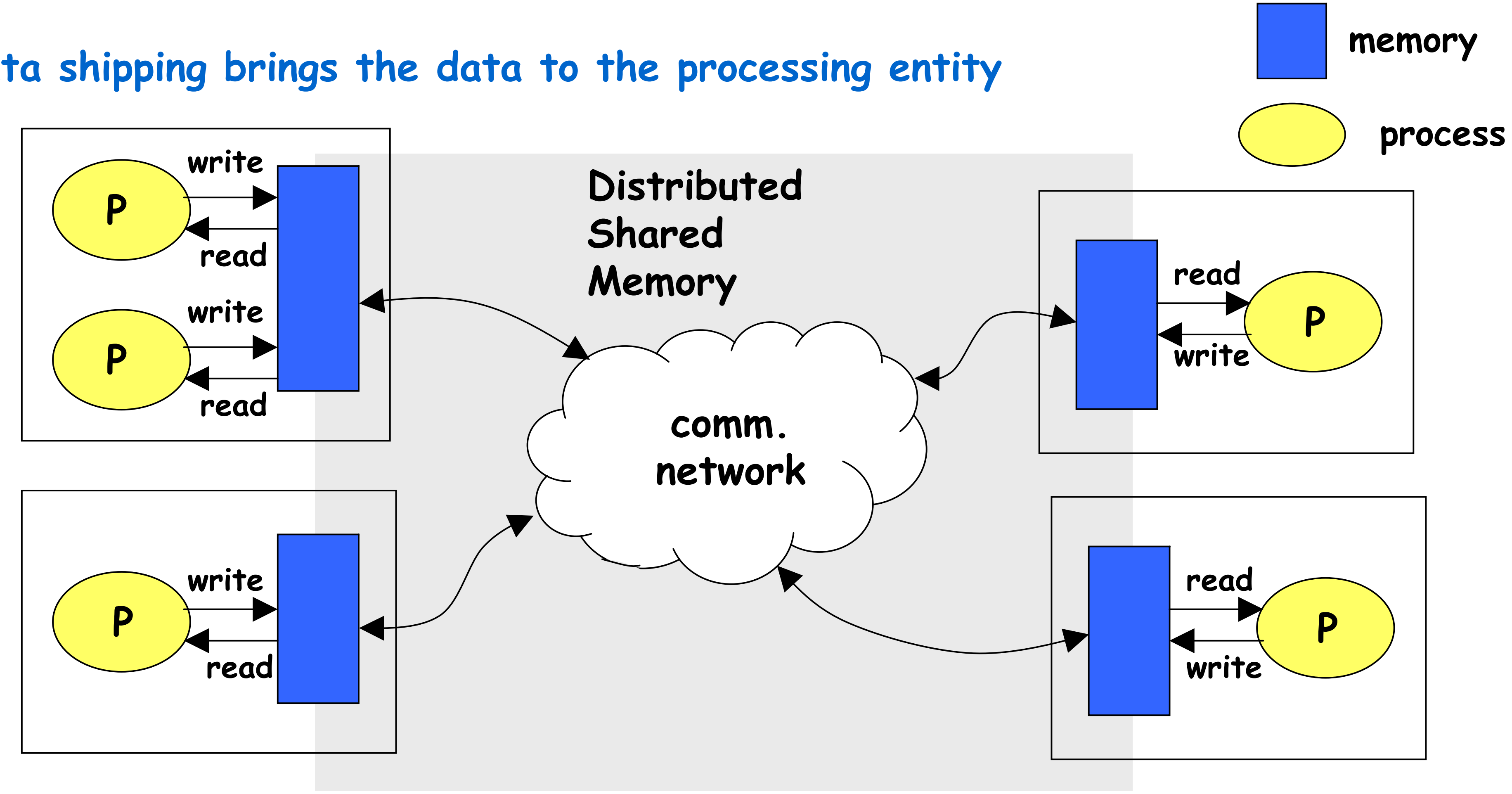
Explicit communication via send and receive: Message passing.



Problem: very low level, very general, poorly defined semantics of communication

Principles of distributed computations

Data shipping brings the data to the processing entity



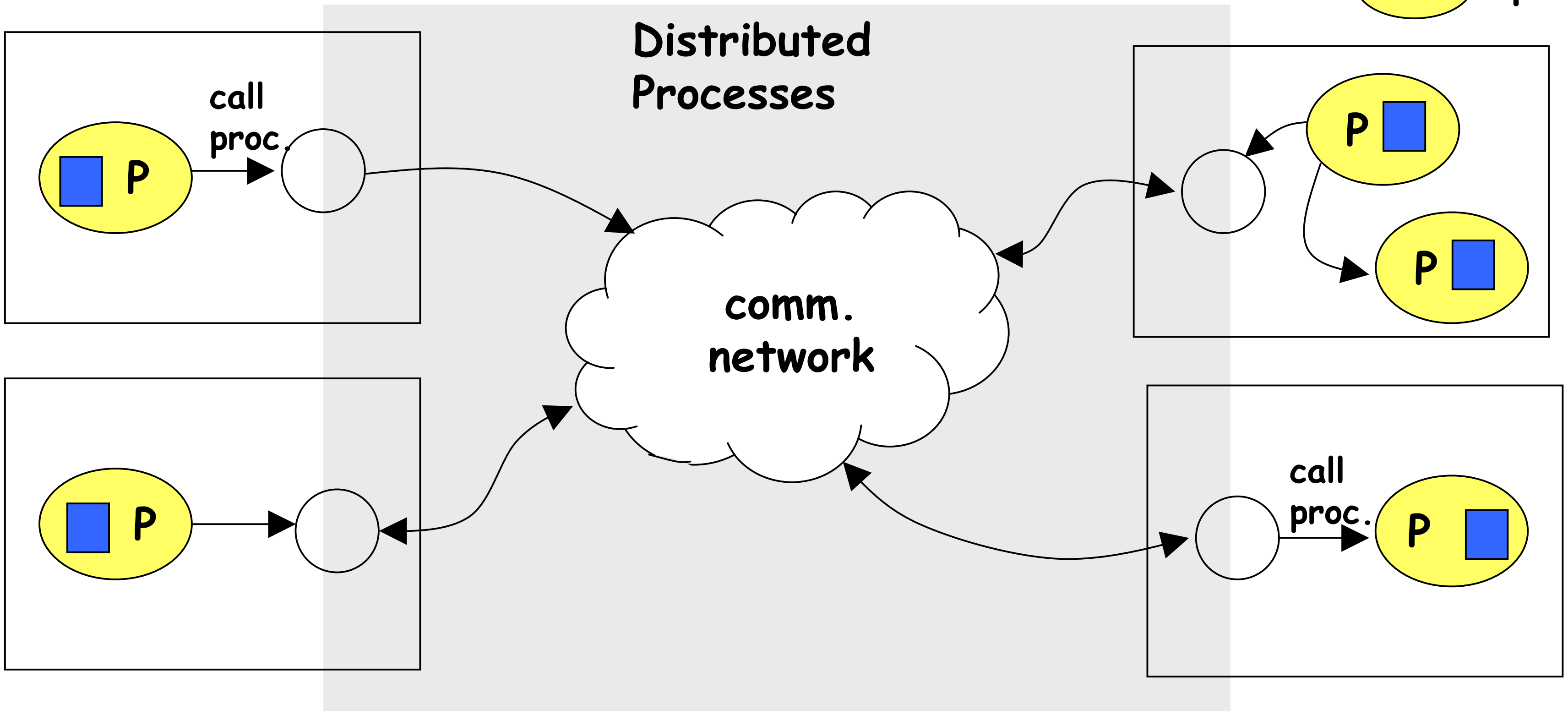
Problem: Consistency in the presence of concurrency and communcation delays

Principles of distributed computations

Function shipping initiates computations in a remote processing entity

🟦 memory

⬭ process



Problem: computation bottlenecks, more complex programming model, references.

abstractions for communication

- Distributed shared memory
- Message passing
- Remote Procedure Call
- Remote Object Invocation
- Notifications
- Publish Subscribe
- Shared data spaces

Types of interaction

explicit	Message-oriented interaction
implicit	Distributed shared memory

request/reply	Client-Server oriented interaction
producer/consumer	Peer-to-Peer interaction

abstractions for communication

Dimensions of Dependencies:

Flow coupling: Control transfer with communication

Defines whether there is a control transfer coupled with a message transfer. E.g. if the sender blocks until a message is correctly received.

Space Coupling: References nust be known

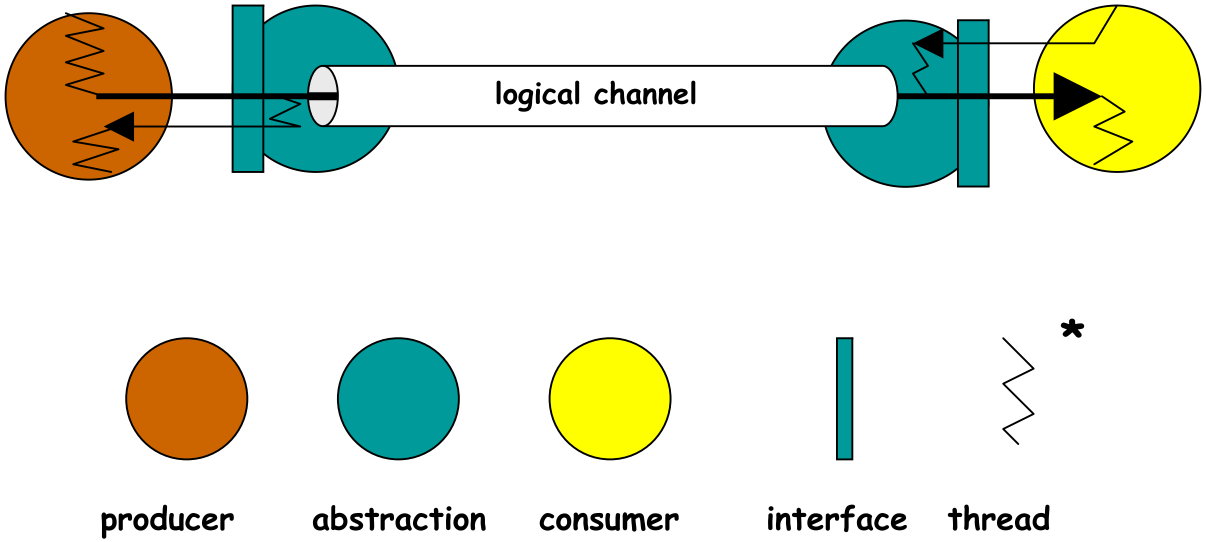
Explicit specification of the destination, i.e. producer must know where to send the message. Message contains an ID specifying an address or name.

Coupling in time: Both sides must be active

Communication can only take place if all partners are up and active.

Message passing

Connected socket, e.g. TCP



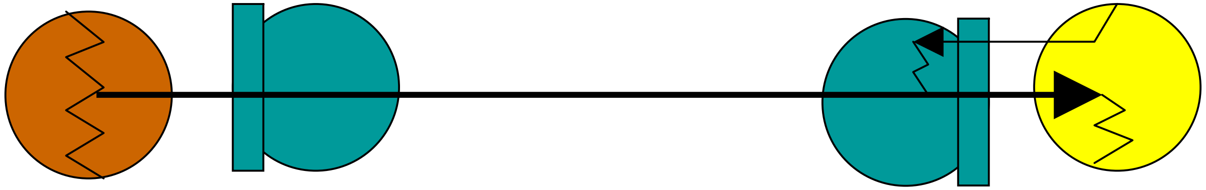
primitives: send (), receive ()

Coupling: time, space, flow

* Notation acc. P. Eugster: Type-Based Publish Subscribe, PhD-thesis, EPFL, Nr. 2503, 2001

Message passing

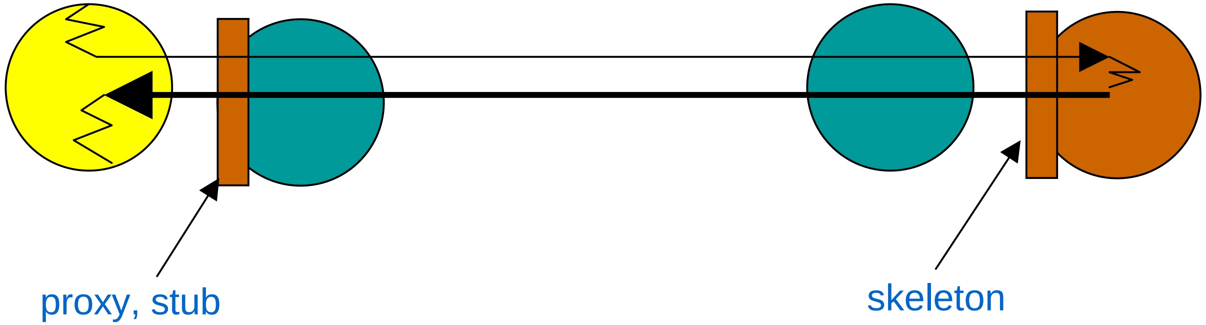
Unconnected socket, e.g. UDP



primitives: send (), receive ()

Coupling: time, space, (flow? unsuccessful if flow is not coordinated)

Remote Procedure Call (RPC)

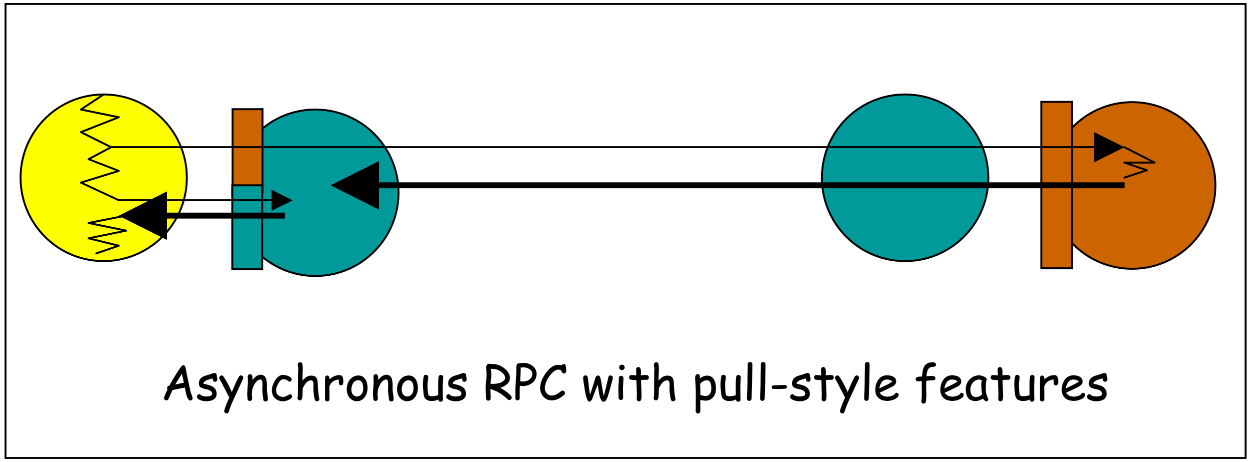


Relation: one-to-one

Coupling:

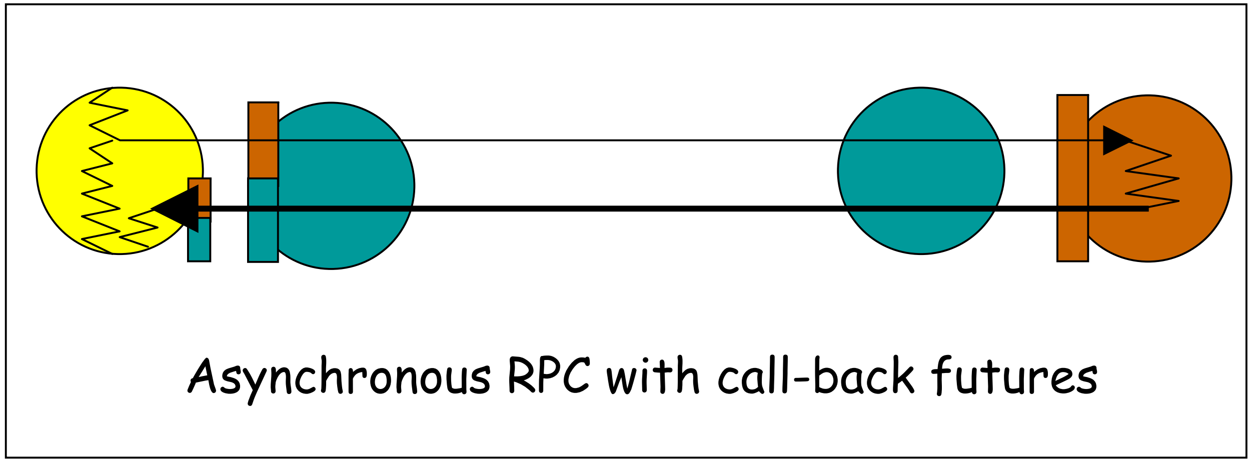
Space:	destination is explicitely specified
Flow:	blocks until message is delivered
Time:	both sides must be active

Variations of RPC



Asynchronous RPC with pull-style features

Example: Concurrent Smalltalk



Asynchronous RPC with call-back futures

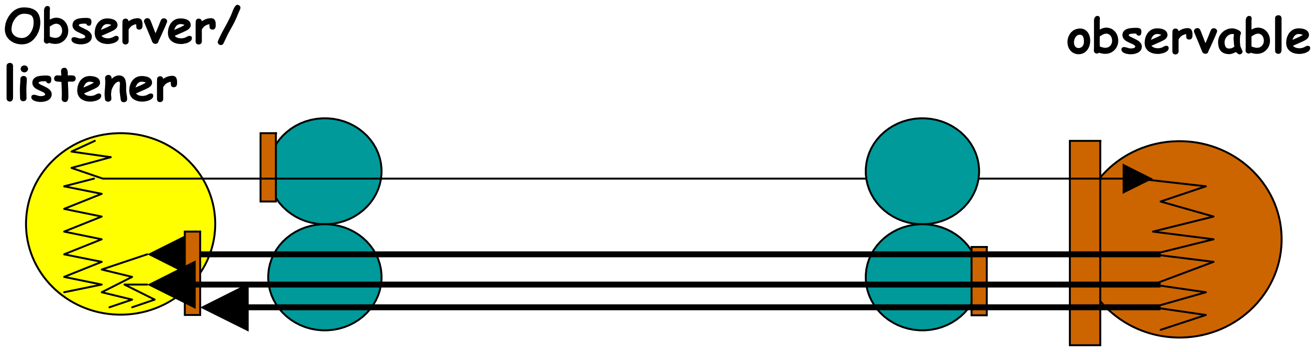
Relation: one-to-one

Coupling:

Space:	destination is explicitely specified
Flow:	no flow coupling
Time:	both sides must be active

Example: Eiffel

Notification

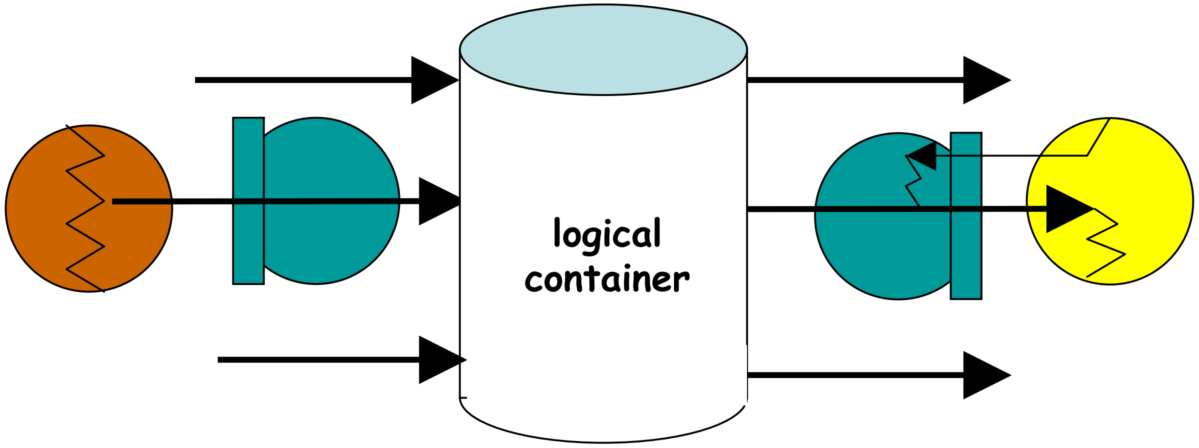


Examles:
Java

Relation: one-to-many

Coupling:
Space: Yes (Observable/Observer pattern (delegation))
Flow: none
Time: both sides must be active (notification performed by RMI)

Shared Data Spaces

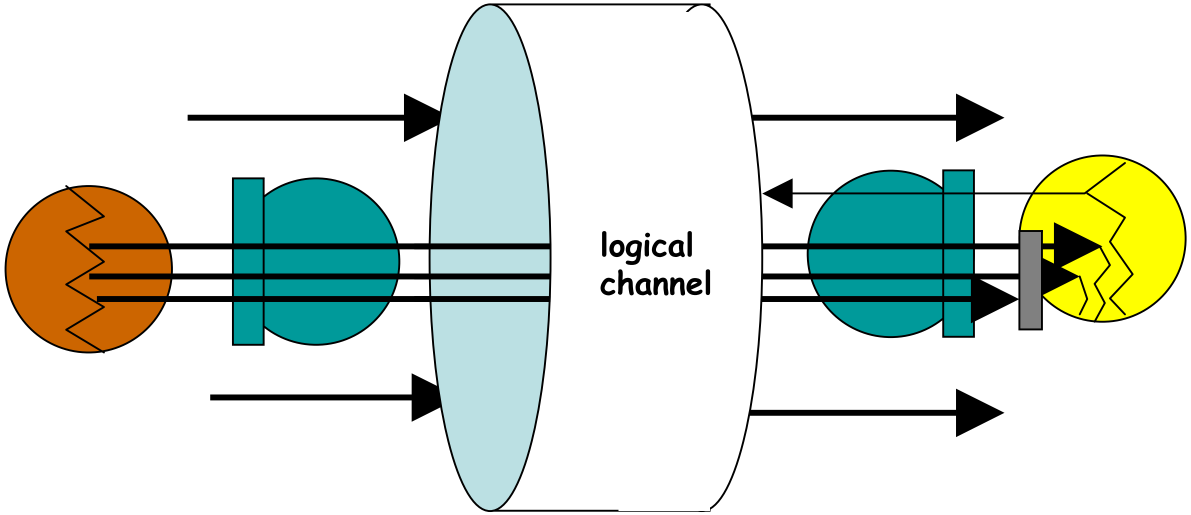


Relation: many-to-many

Coupling:
Space: none
Flow: consumer side
Time: none

Examples:
Linda tuple Space
Java Spaces
ADS Data field

Publish/Subscribe



Relation: many-to-many

Coupling:

Space: none/indirect

Flow: none

Time: none

Examples:

Information Bus

NDDS

Real-Time P/S

COSMIC

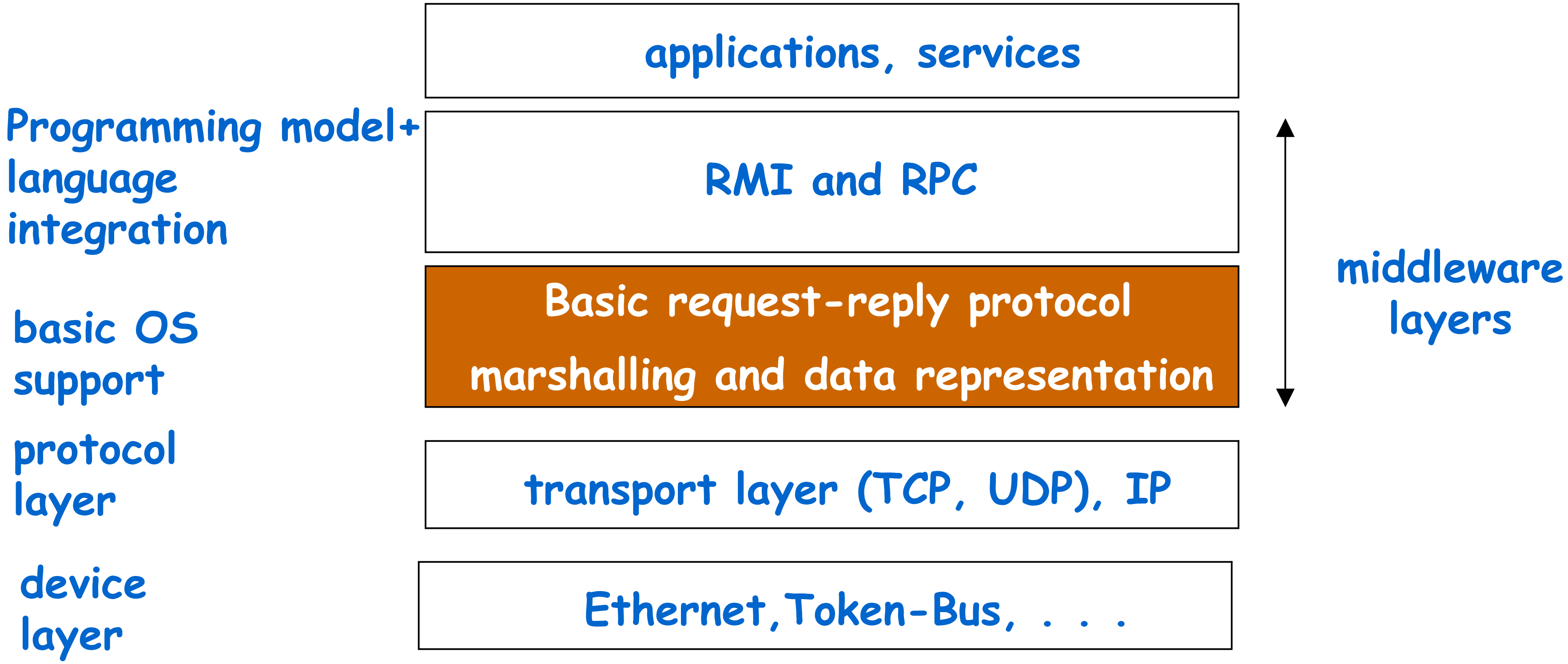
....

....

Overview

Abstraction	Space Coupling	Time Coupling	Flow Coupling
• Connected Sockets	Yes	Yes	Yes
• Unconnected Sockets	Yes	Yes	Consumer
• RPC	Yes	Yes	Consumer
• Oneway RPC	Yes	Yes	No
• Explicit Future (Pull)	Yes	Yes	No
• Explicit FutureCallback)	Yes	Yes	No
• Implicit Future	Yes	Yes	No
• Notications	Yes	Yes	No
(Observer Design Pattern)			
• Tuple Spaces (Pull)	No	No	Consumer
• Message Queues (Pull)	No	No	Consumer
• Subject-Based P/S	No	No	No
• Content-Based P/S	No	No	No

Building IPC from bottom-up



applications, services

Programming model+ language integration

RMI and RPC

basic OS support

Basic request-reply protocol

marshalling and data representation

middleware layers

protocol layer

transport layer (TCP, UDP), IP

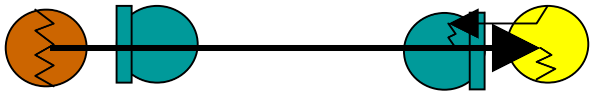
device layer

Ethernet,Token-Bus, . . .

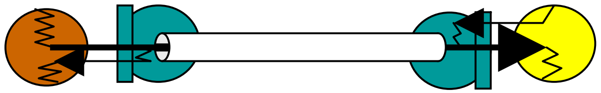
abstractions of the transport layer

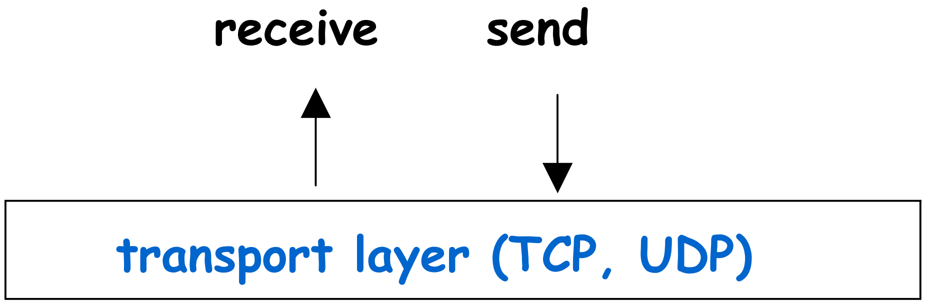
OS-abstraction: socket
Protocols: TCP, UDP

UDP: unconnected sockets, single messages
→ datagramm coomunication



TCP: conn. sockets, two-way message streams between process pairs.
→ stream communication



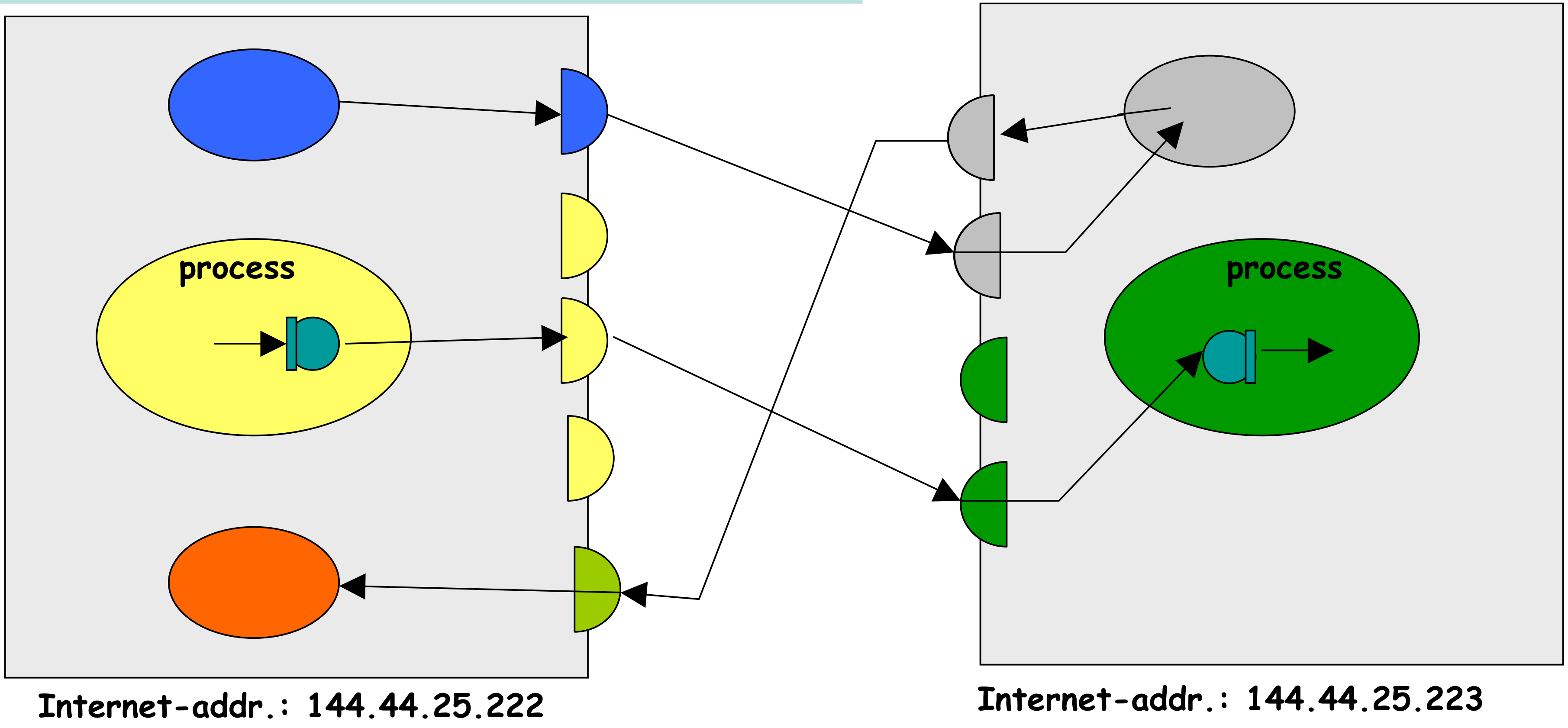


receive

send

transport layer (TCP, UDP)

sockets and ports



How to route a message to a process?
- IP-Adress addresses a computer.
- Port: is associated with a process

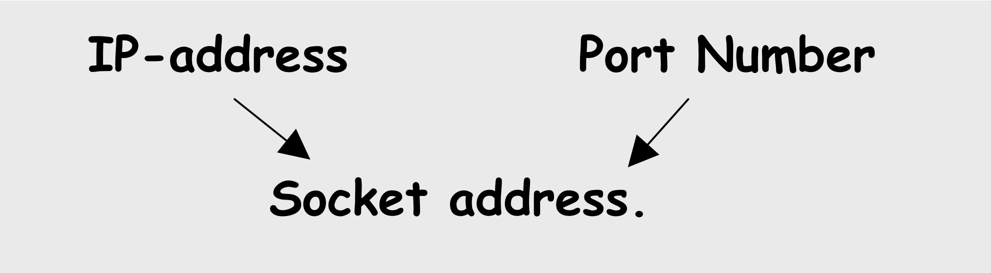
sockets and ports

What is needed to send/receive a message through a socket?

1. Internet-address of the local node.

2. Local port (every computer has a large number (2^{16}) of possible port numbers).

3. A binding mechanism.

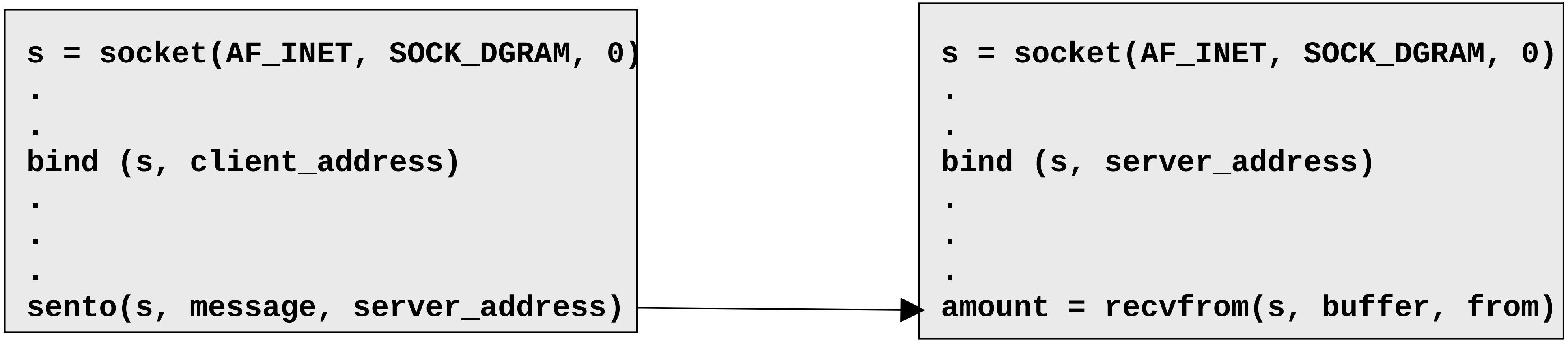


IP-address

Port Number

Socket address.

Example: datagram sockets in Unix



```
s = socket(AF_INET, SOCK_DGRAM, 0)
.
.
bind (s, client_address)
.
.
.
sento(s, message, server_address)
```

```
s = socket(AF_INET, SOCK_DGRAM, 0)
.
.
bind (s, server_address)
.
.
.
amount = recvfrom(s, buffer, from)
```

`socket:` system call to create a socket data structure and obtain the resp. descriptor

`AF_INET:` communication domain as Internet domain

`SOCK-DGRAM:` type of communication: datagram communication

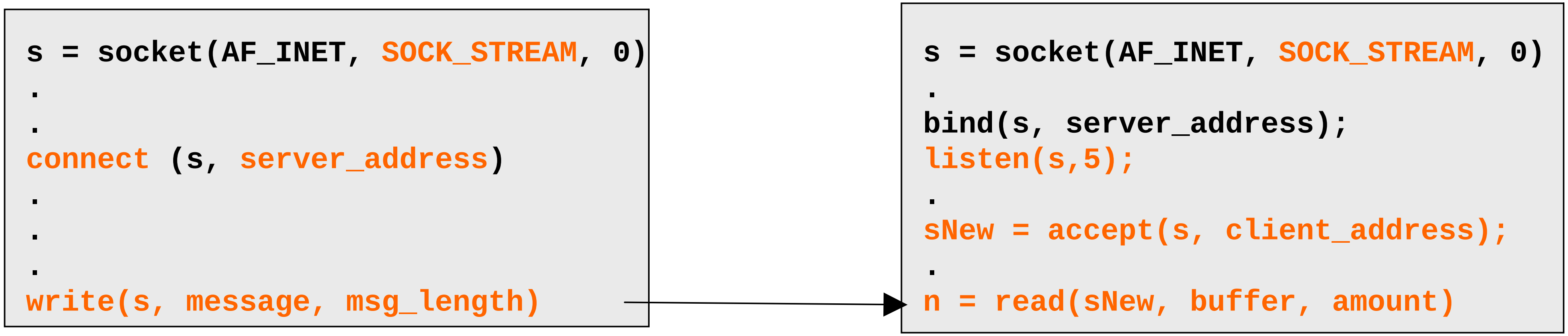
`0:` optional specification of the protocol. If “`0`“ is specified, the protocol is automatically selected. Default: UDP for datagram comm., TCP for stream comm.

`bind:` system call to asociate the socket “`s`“ with a socket address <IP address, port number>.

`sento:` system call to send a message via socket “`s`“ to the specified server socket `"server_address"`.

`recfrom:` system call to receive a message from socket “`s`“ and put it at memory location “`buffer`“. “`from`“ specifies the pointer to the data structure which contains the sending socket‘s address. recvfrom takes the first elemet from a queue and blocks if the queue is empty.

Example: stream sockets in Unix



```
s = socket(AF_INET, SOCK_STREAM, 0)
.
.
connect (s, server_address)
.
.
.
write(s, message, msg_length)
```

```
s = socket(AF_INET, SOCK_STREAM, 0)
.
bind(s, server_address);
listen(s,5);
.
sNew = accept(s, client_address);
.
n = read(sNew, buffer, amount)
```

Differences to the datagram communication interface:

`SOCK_STREAM:` type of communication: datagram communication

`listen:` server waits for a connection request of a client. "5" specifies the max. number of requested connections waiting for acceptance.

`acccept:` system call to accept a new connection and create a new dedicated socketfor this connection.

`connect:` requests a connection with the specified server via the previously specified socket.

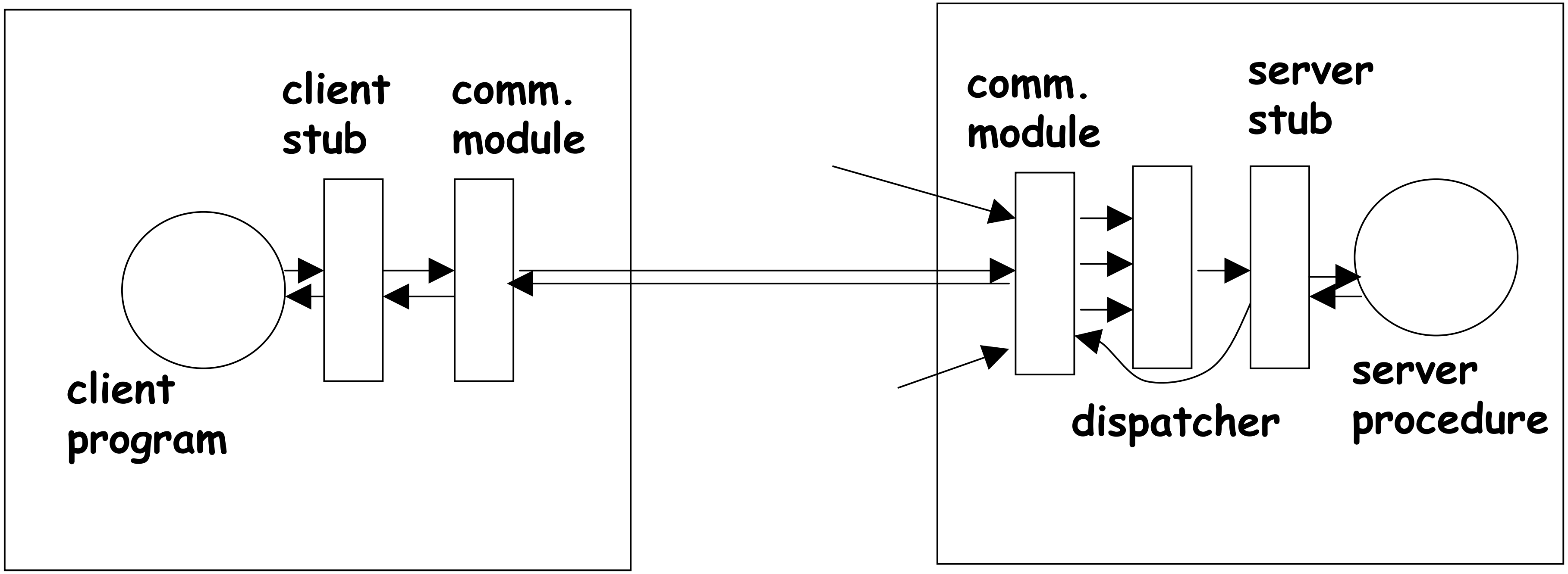
`read/write:` after the connection is established, write and read calls on the sockets can be used to send and receive byte streams.
`write` forwards the byte stream to the underlying protocol and returns number of bytes sent successfully.
`read` receives a byte stream in the respective buffer and returns the number of received bytes.

Remote Procedure Call (RPC)

Archtitecture: defines layers and interfaces between layers

Organization: defines components, behaviour and interaction

Remote Procedure Call (RPC)



Interface-definition-language (e.g.XDR)
Binding (Server - Portnr.)
Authentication

Inter-Process-Communication (IPC)

multiple processes cooperate

Advantages: performance by concurent activities

structuring of application

Message oriented communication

Explicit message exchange between processes

Shared memory

Access to a set of memory cells

Classification of message-oriented IPC:

Abstractions:

- channels (Pipes)
- Communication end points (Sockets, Ports)
- Mailboxes, Queues
- Signals

Channel classification

- unidirektional
- bidirektional (full-duplex, half-duplex)