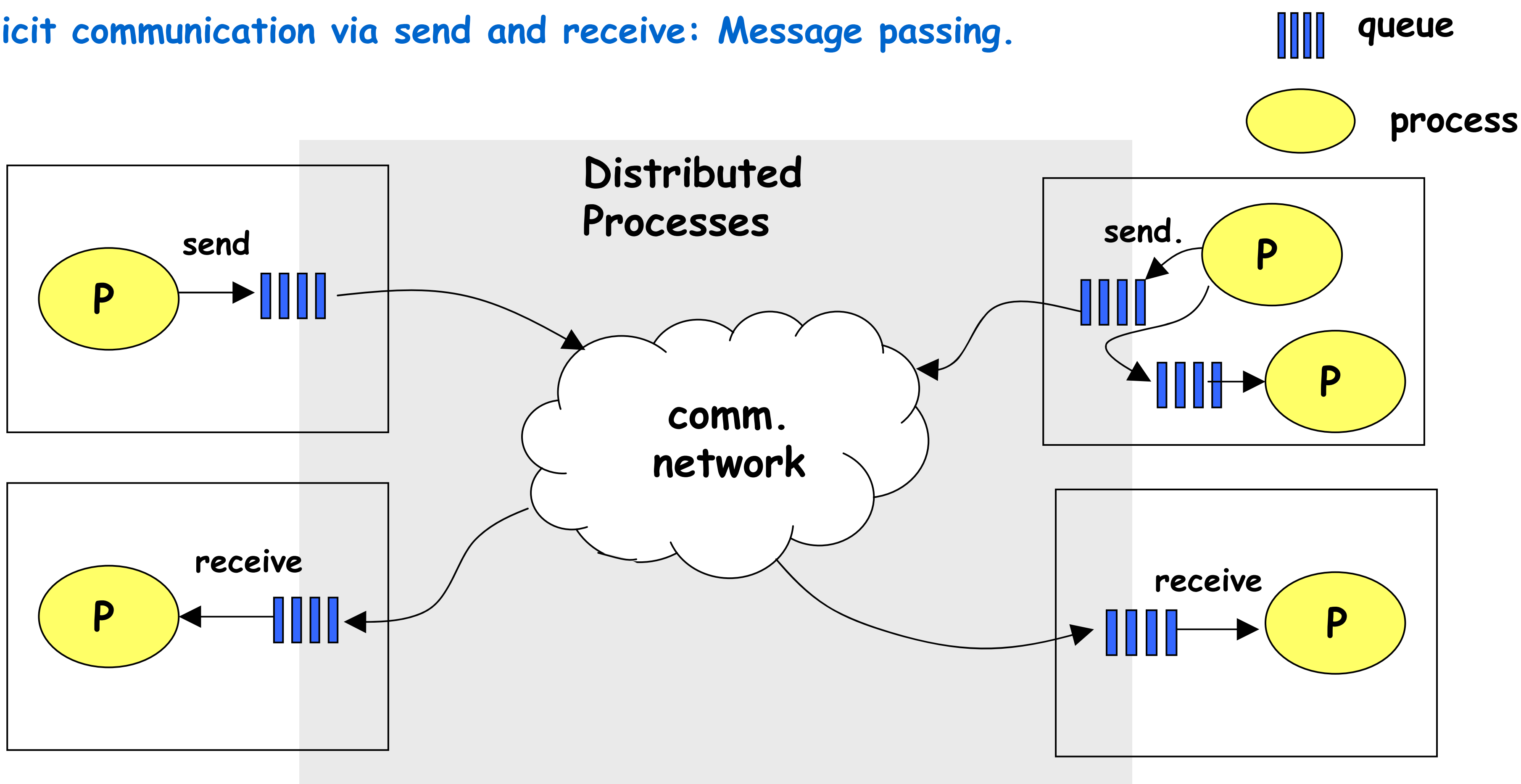
IPC
Inter Process Communication

Principles of distributed computations

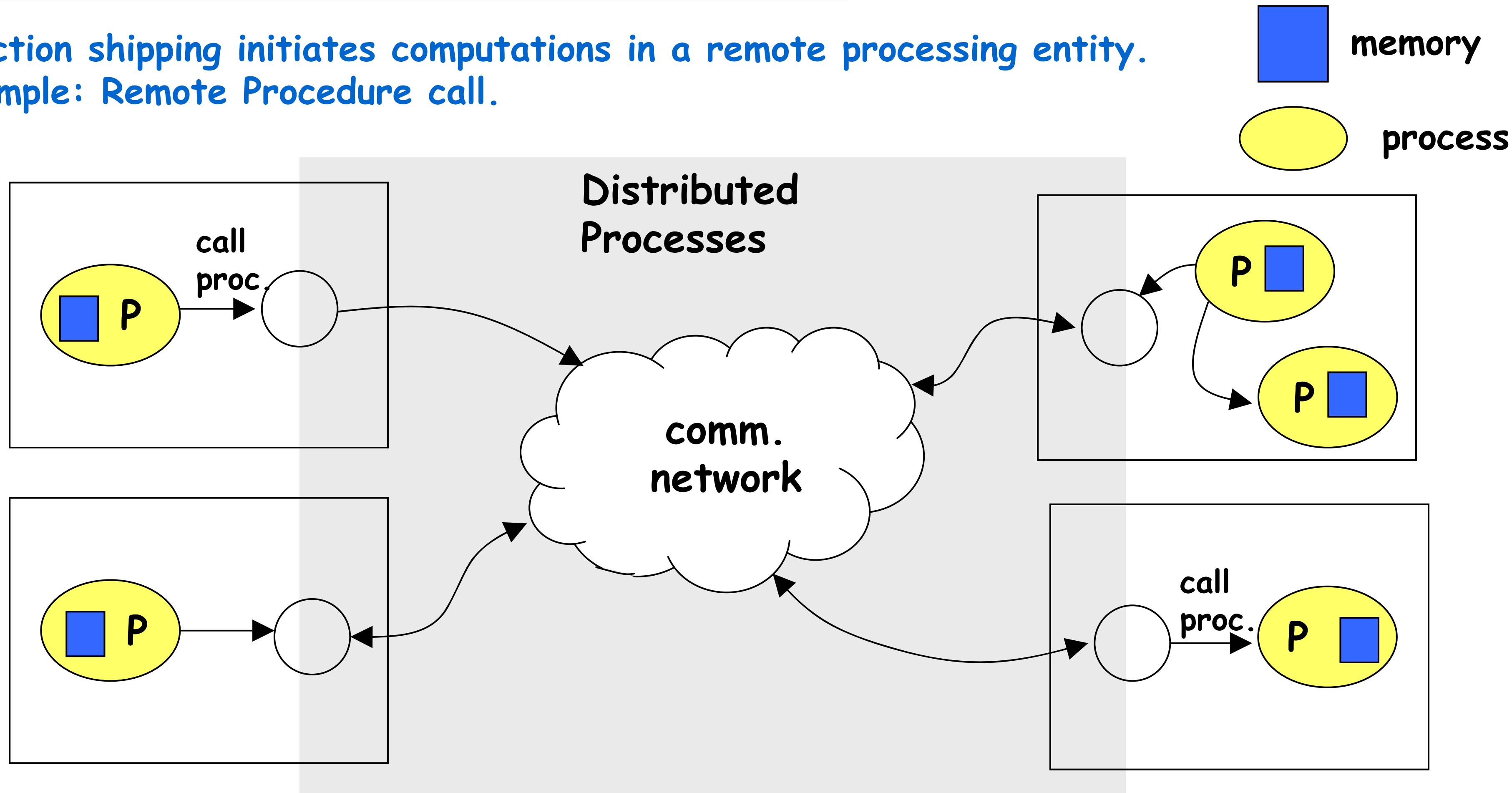
Explicit communication via send and receive: Message passing.



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

Principles of distributed computations

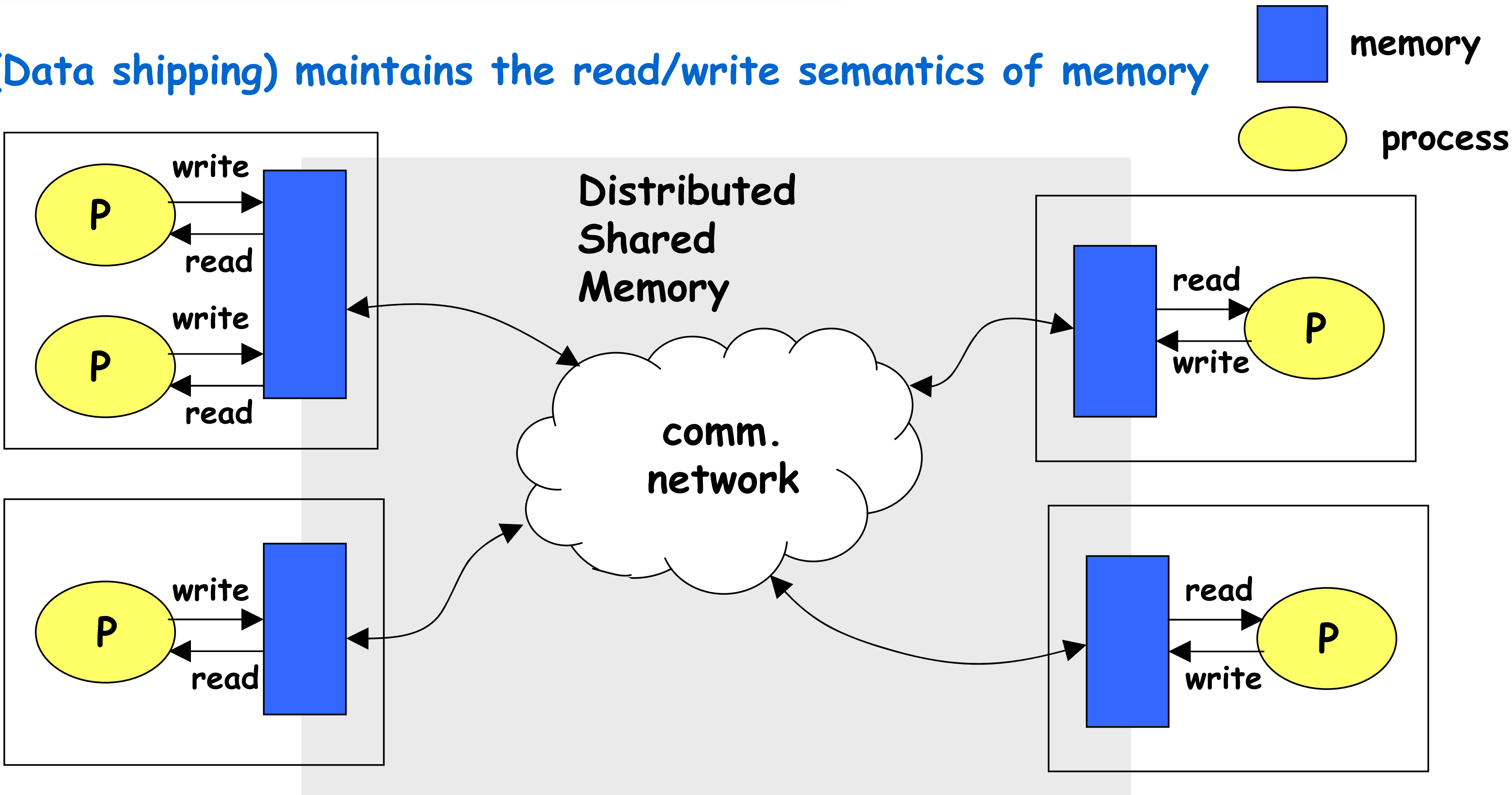
Function shipping initiates computations in a remote processing entity.
Example: Remote Procedure call.



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

Principles of distributed computations

DSM (Data shipping) maintains the read/write semantics of memory



Problem: Consistency in the presence of concurrency and communcation delays

IPC
Inter Process Communication

abstractions for communication

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

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 must be known

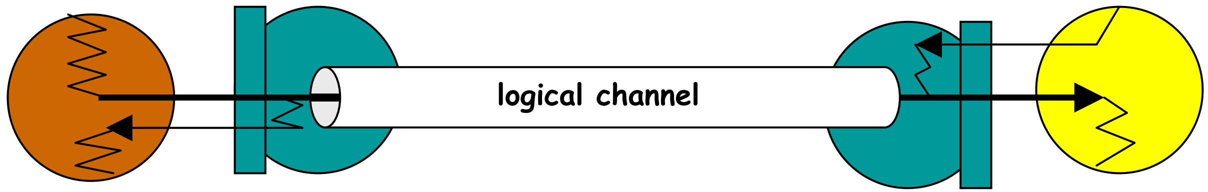
Explicit specification of the destination, i.e. producer must know where to send the message. Message contains an ID specifying an address or name.

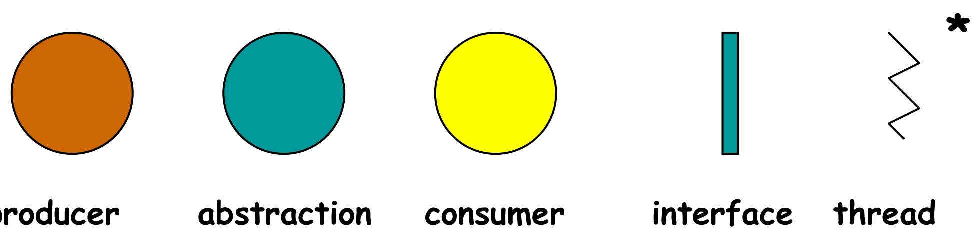
Coupling in time: Both sides must be active

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

Message passing

Connected socket, e.g. TCP





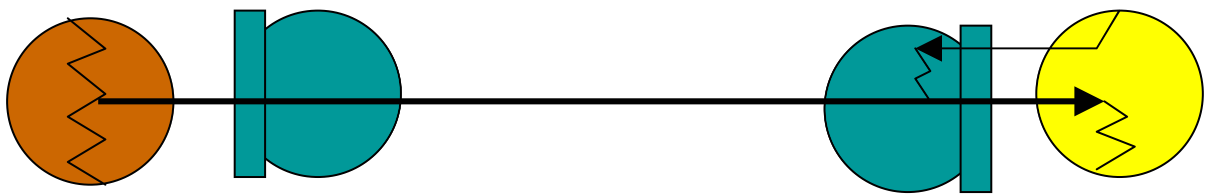
primitives: send (), receive ()

Coupling: time, space, flow

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

Message passing

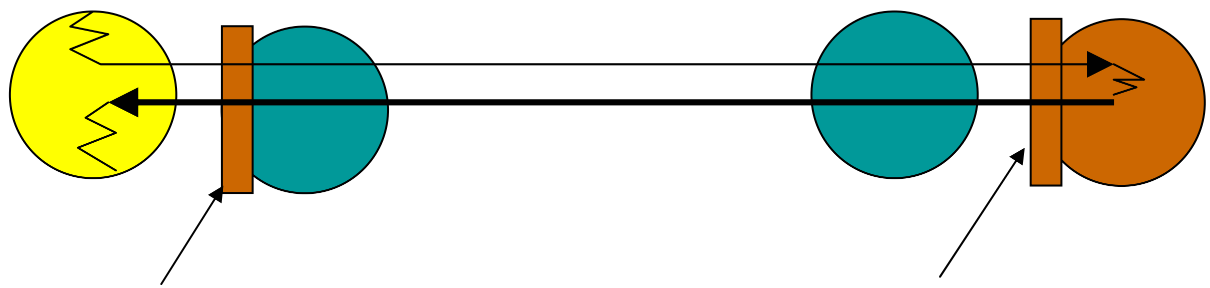
Unconnected socket, e.g. UDP



primitives: send (), receive ()

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

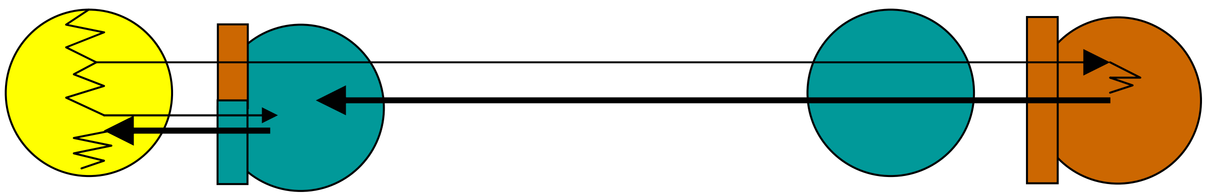
Remote Procedure Call (RPC)



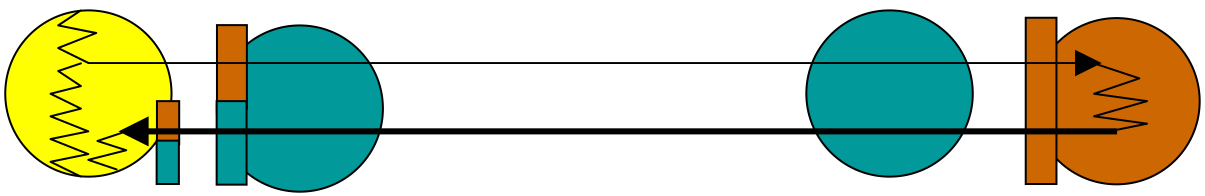
Relation: one-to-one

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



Asynchronous RPC with call-back

Example: Concurrent Smalltalk

Relation: one-to-one

Coupling:

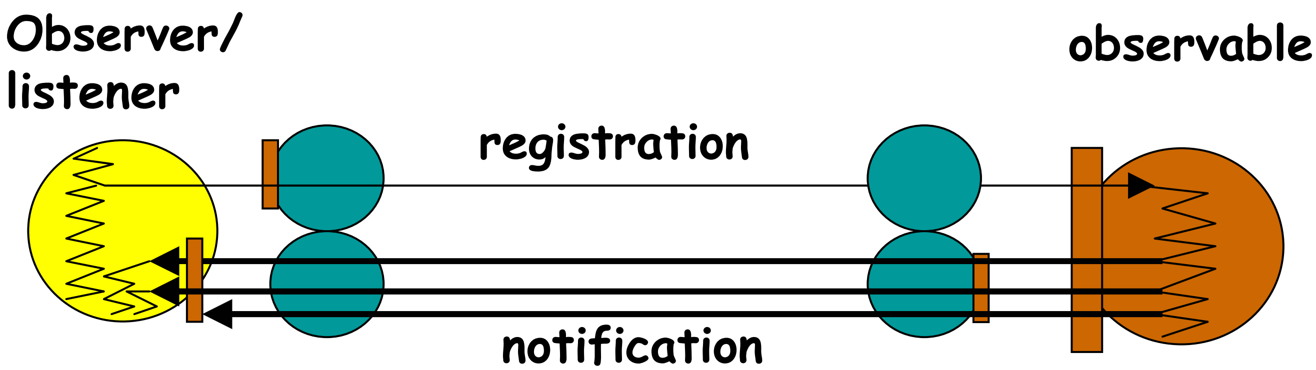
Space: destination is explicitely specified

Flow: no flow coupling

Time: both sides must be active

Example: Eiffel

Notification

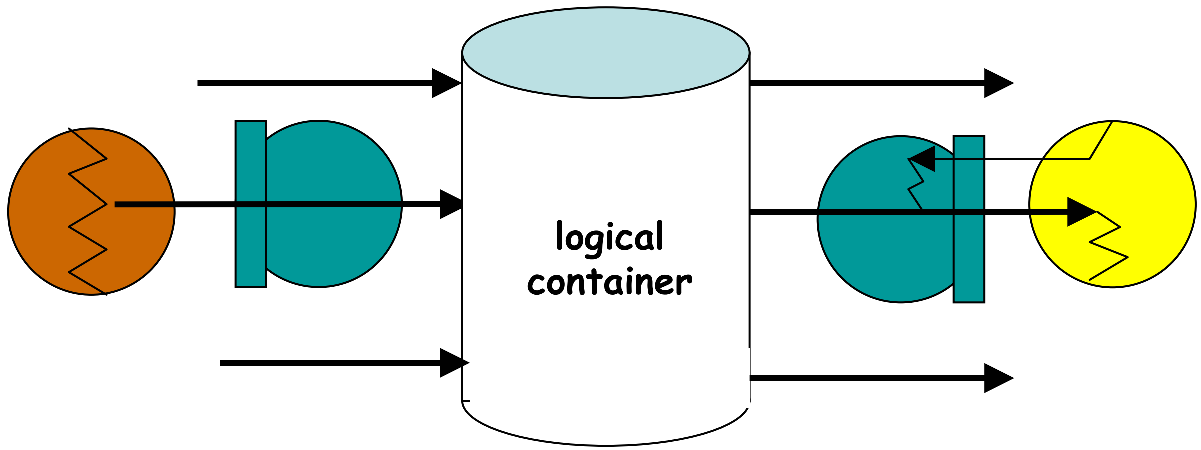


Examles:
Java

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

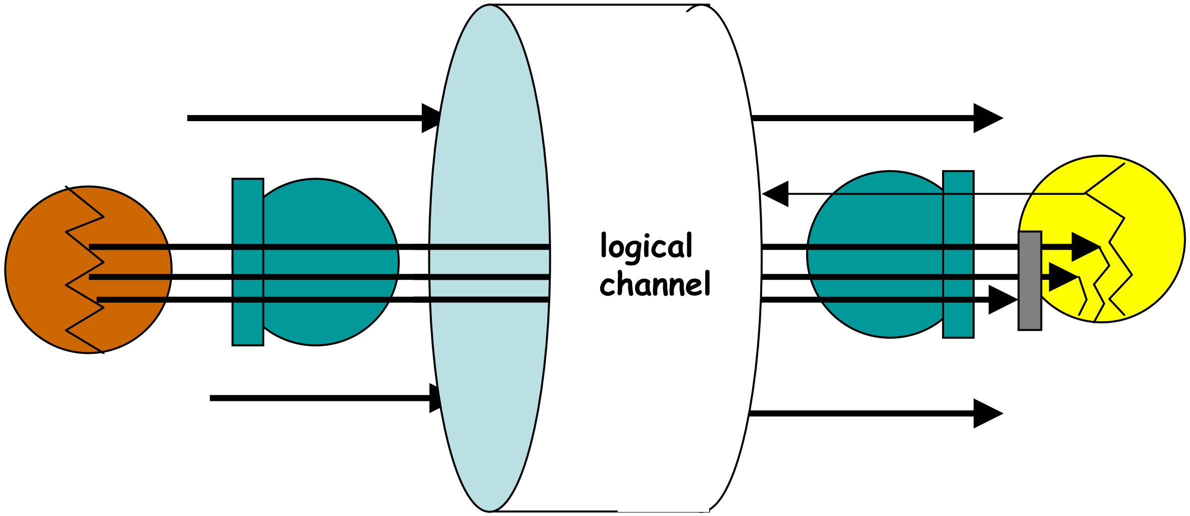


Relation: many-to-many

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

Examples:
Linda tuple Space
Java Spaces
ADS Data field

Publish/Subscribe



Relation: many-to-many

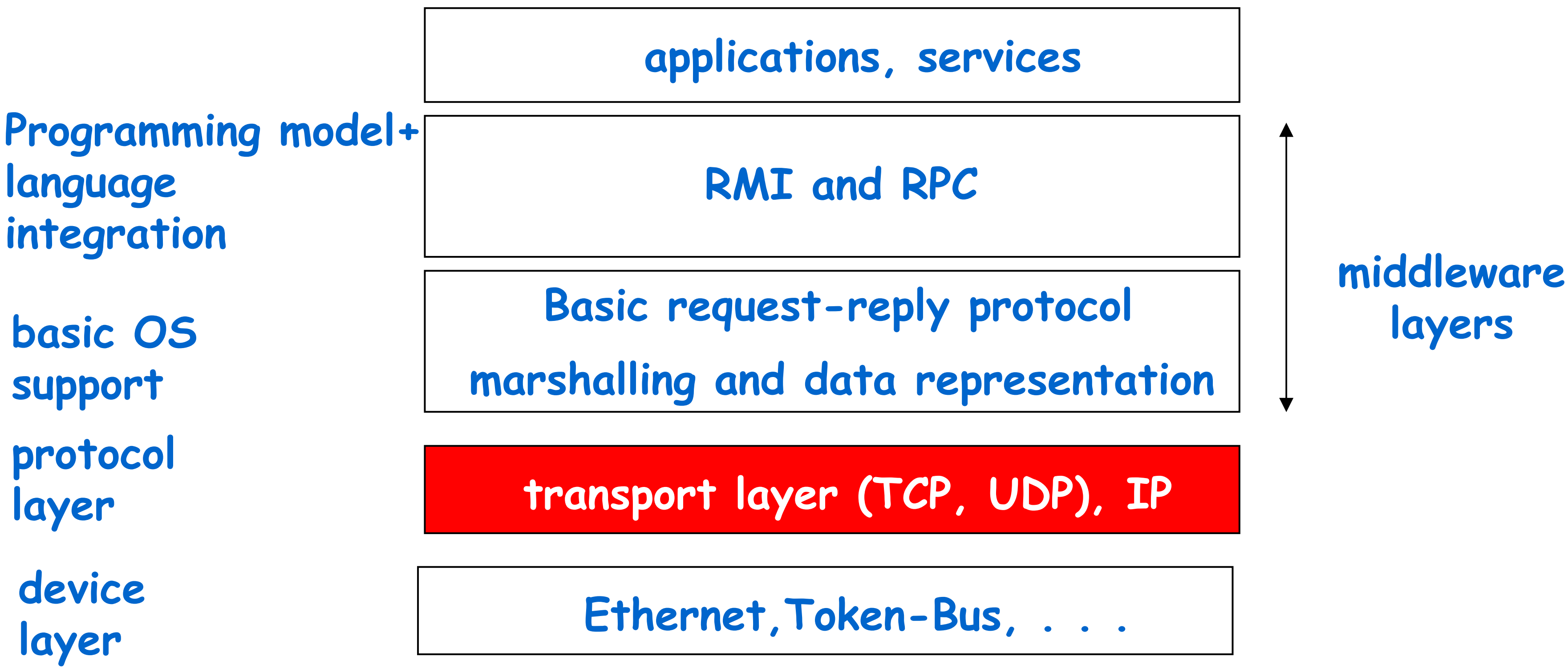
Coupling:
Space: none
Flow: none
Time: none

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
• async (Pull)	Yes	Yes	No
• async (Callback)	Yes	Yes	No
• Implicit Future	Yes	Yes	No
• Notications (Observer Design Pattern)	Yes	Yes	No
• 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

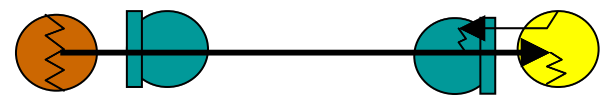
the lower layers of IPC



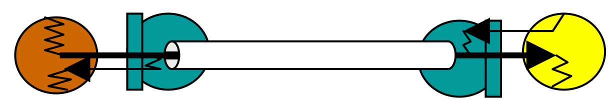
abstractions of the transport layer

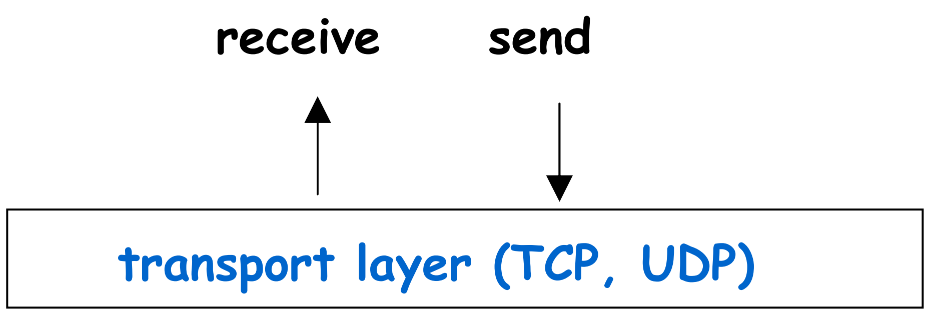
OS-abstraction: socket
Protocols: TCP, UDP

UDP: unconnected sockets, single messages
→ datagramm coomunication

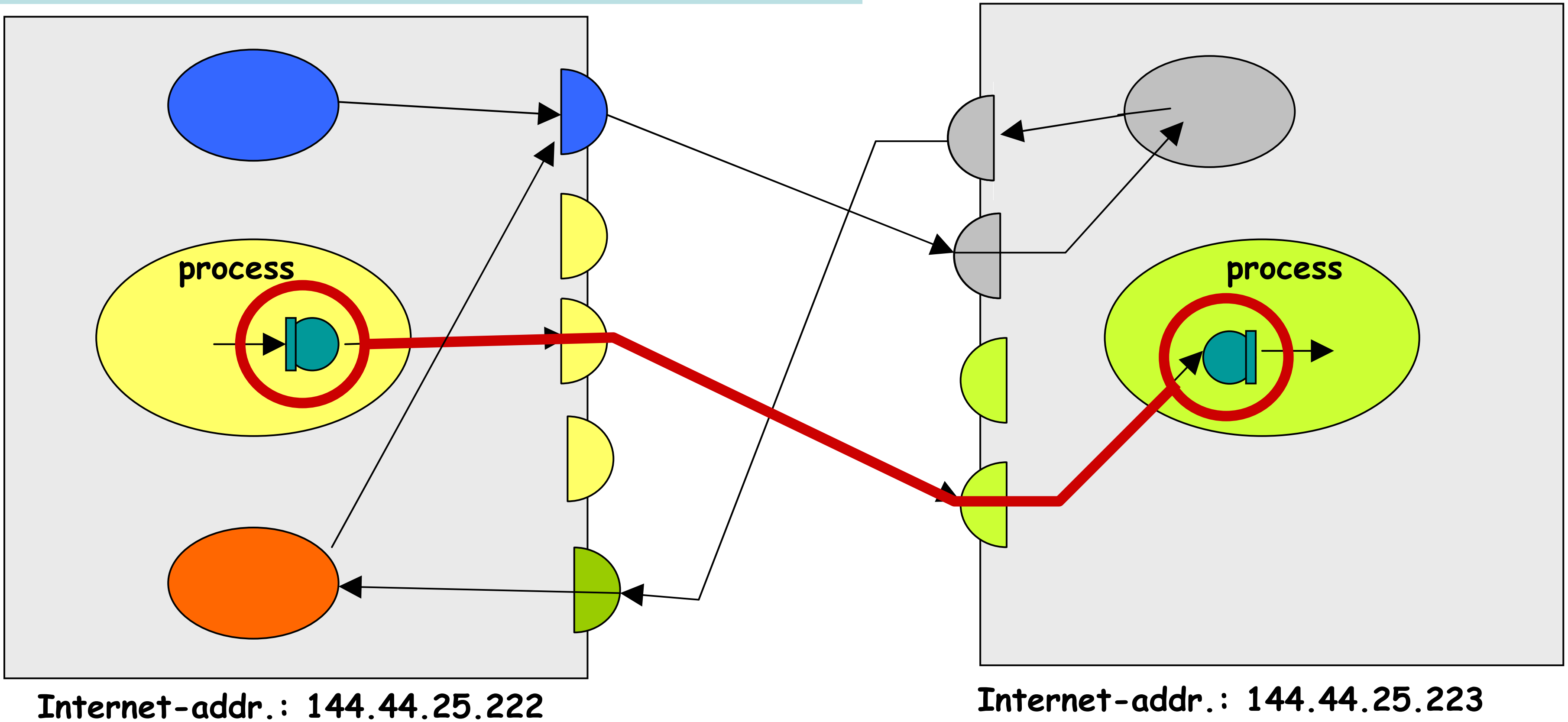


TCP: conn. sockets, two-way message streams between process pairs.
→ stream communication





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.

(server) IP-address

(local) 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 associate 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 element 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: stream 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.

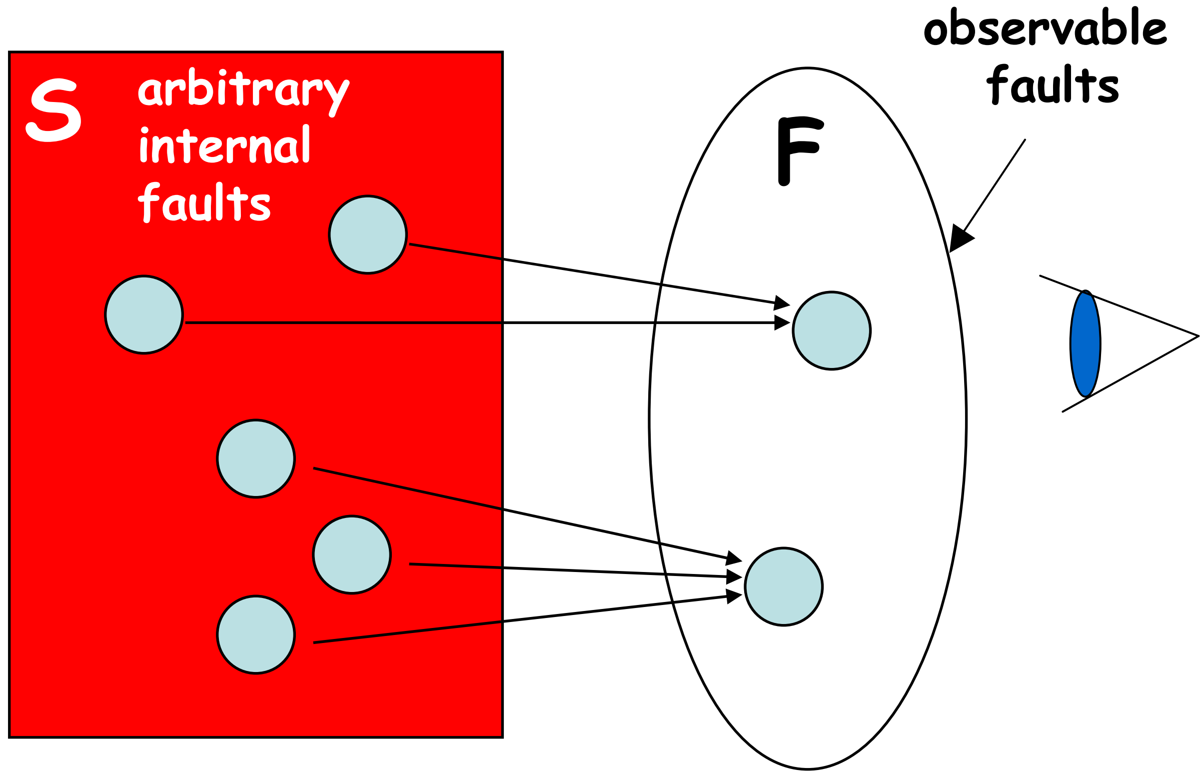
connect: requests a connection with the specified server via the previously specified socket.

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.

Fault model and failure Semantics

Problem:
For an application programmer it would be extremely hard to deal with arbitrary faults.

Approach:
System masks faults or maps fault to a class which can be handeled by a programmer easily.

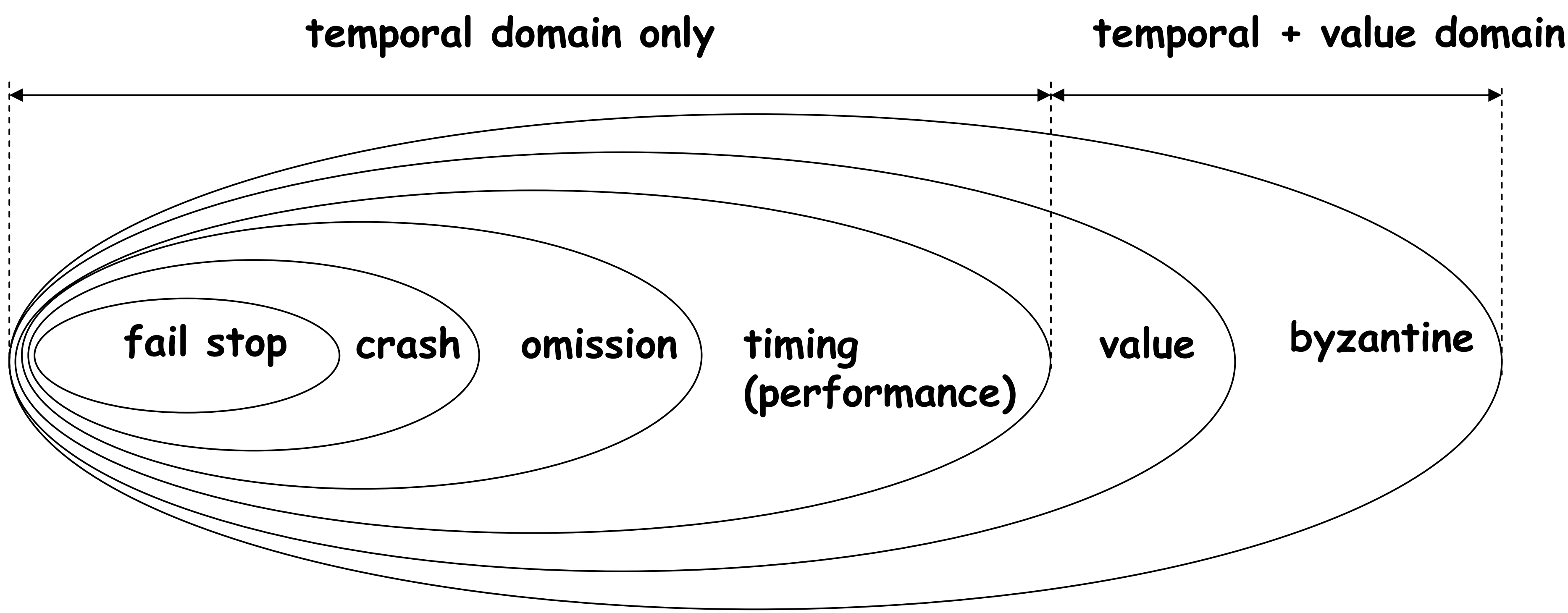


S has the failure semantics F

Fault model and failure Semantics

Fault Class	affects:	description
fail stop	process	A process crashes and remains inactive. All all participants safely detect this state.
crash	process	A process crashes and remains inactive. Other processes amy not detect this state.
omission	channel	A message in the output message buffer of one process never reaches the input message buffer of the other process.
-send om.	process	A process completes the send but the respective message is never written in its send output buffer.
-receive om.	process	A message is written in the input message buffer of a process but never processed.
byzantine	process or channel	An arbitrary behaviour of process or channel.

Fault model and failure Semantics



masking
mapping } resend, time-out, duplicate msg. recognition and removal, check sum, replication, majority voting.

Fault model and failure Semantics

Reliable 1-to-1 Communication:

Validity: every message which is sent (queued in the out-buffer of a correct process) will eventually be received (queued in the in-buffer of an correct process)

Integrity: the message received is identical with the message sent and no message is delivered more than once.

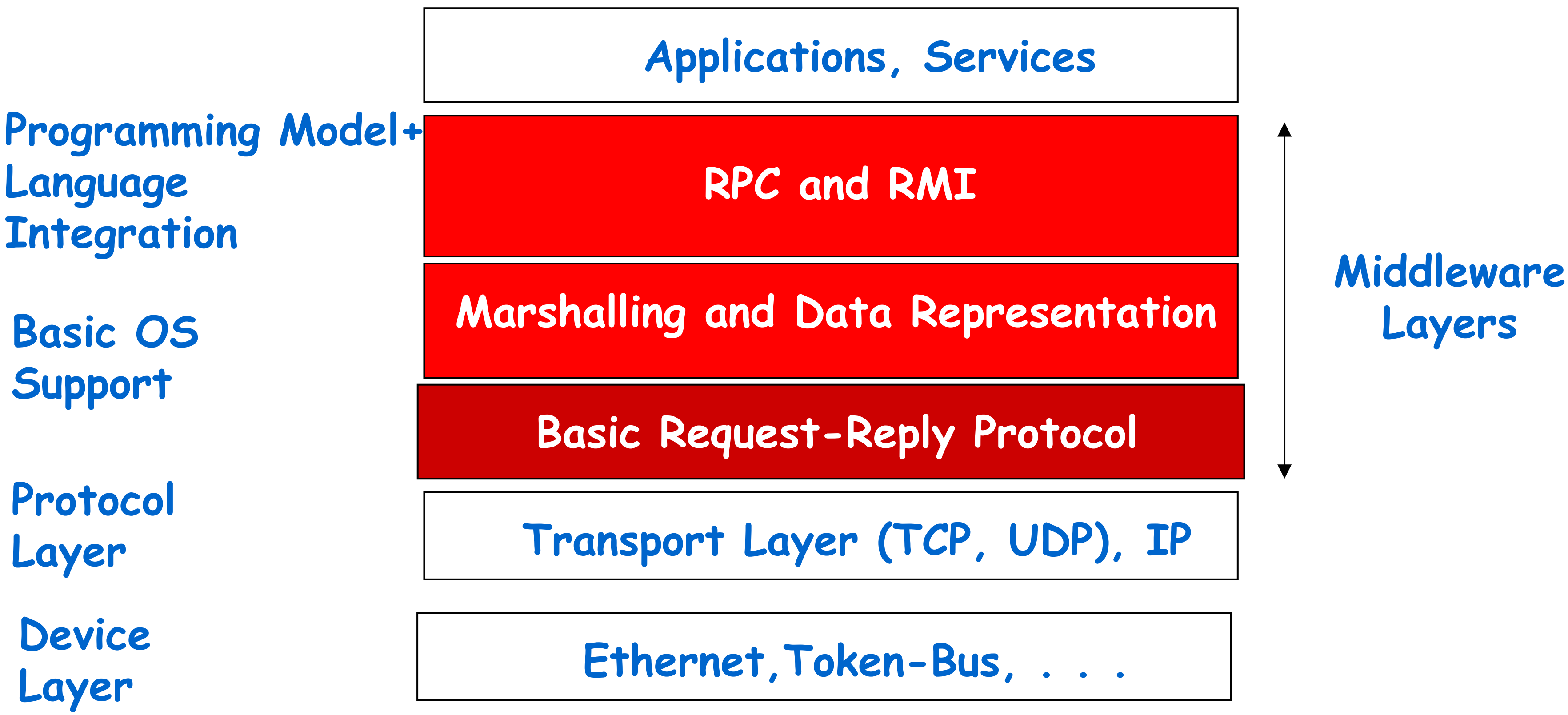
Validity and integrity are properties of a channel!

Fault model and failure Semantics

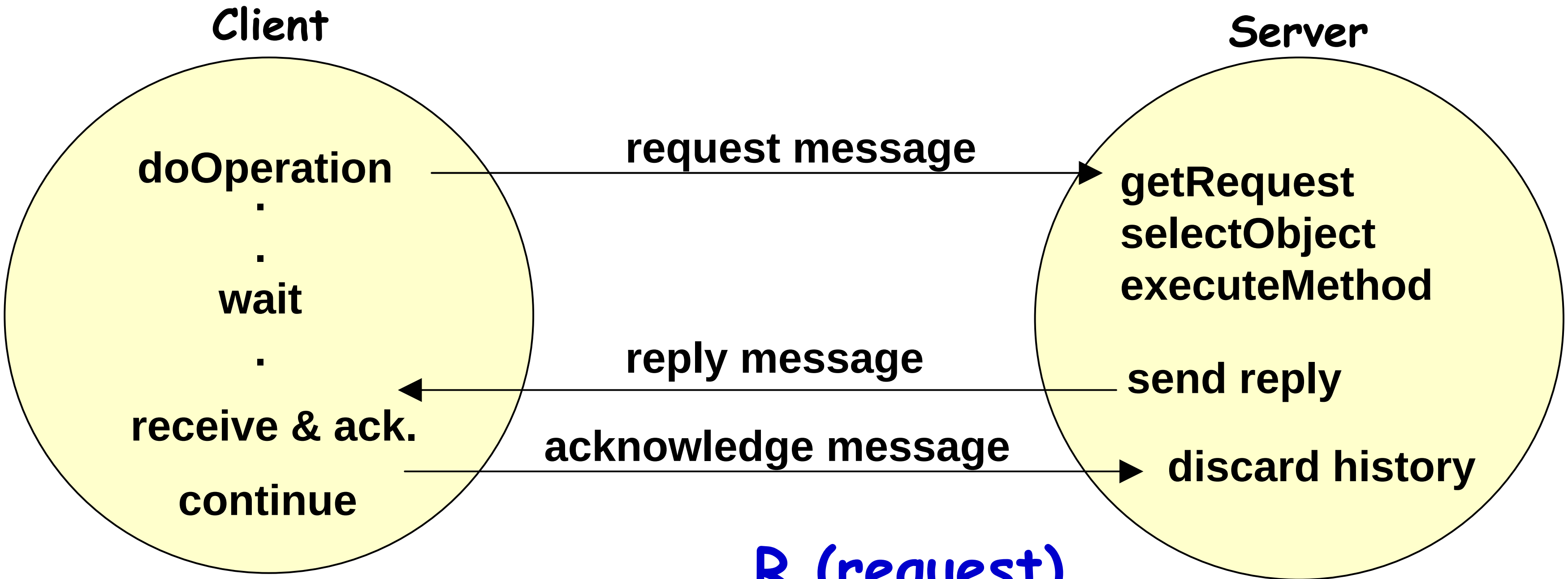
UDP provides Channels with Omission Faults and doesn't guarantee any order.
TCP provides a Reliable FiFo-Ordered Point-to-Point Connection (Channel)

Mechanisms	Effect
sequence numbers assigned to packets	FiFo between sender and receiver. Allows to detect duplicates.
acknowledge of packets	Allows to detect missing packets on the sender side and initiates retransmission
Checksum for data segments	Allows detection of value failures.
Flow Control	Receiver sends expected "window size" characterizing the amount of data for future transmissions together with ack.

Distributed Objects and Remote Invocation



Request-Reply Communication



R (request)
RR (request-reply)
RRA (request-reply-ack)

Request-Reply Communication

Operations:

public byte[] ***doOperation*** *(RemoteObjectRef o, int methodId, byte[] arguments)*
sends a request message to the remote object and returns the reply.
The arguments specify the remote object, the method to be invoked
and the arguments of that method.

public byte[] ***getRequest*** *();*
acquires a client request via the server port.

public void ***sendReply*** *(byte[] reply, InetAddress clientHost, int clientPort);*
sends the reply message reply to the client at its Internet address and port.

Request-Reply Communication

message structure

messageType	*int (0=Request, 1= Reply)*
requestId	*int (process specific sequence number)*
objectReference	*RemoteObjectRef*
methodId	*int or Method*
arguments	*array of bytes*

remote object reference

32 bits	*32 bits*	*32 bits*	*32 bits*	
Internet address	port number	time	object number	interface of remote object

Discussion: Fault Model of Request-Reply Communication

If the request-reply primitives are implemented on UDP sockets the designer has to cope with the following problems:

Omissions may occur,
Send order and delivery order may be different.

Detection of lost (request or reply) messages

Mechanism: Timeout in the client

Request was processed in the server - (reply is late or lost).
Request was not processed - (request was lost).

Removal of duplicated request messages in the server:

New request arrives before the old request has been processed (no reply yet).
New request arrives after the reply was sent.

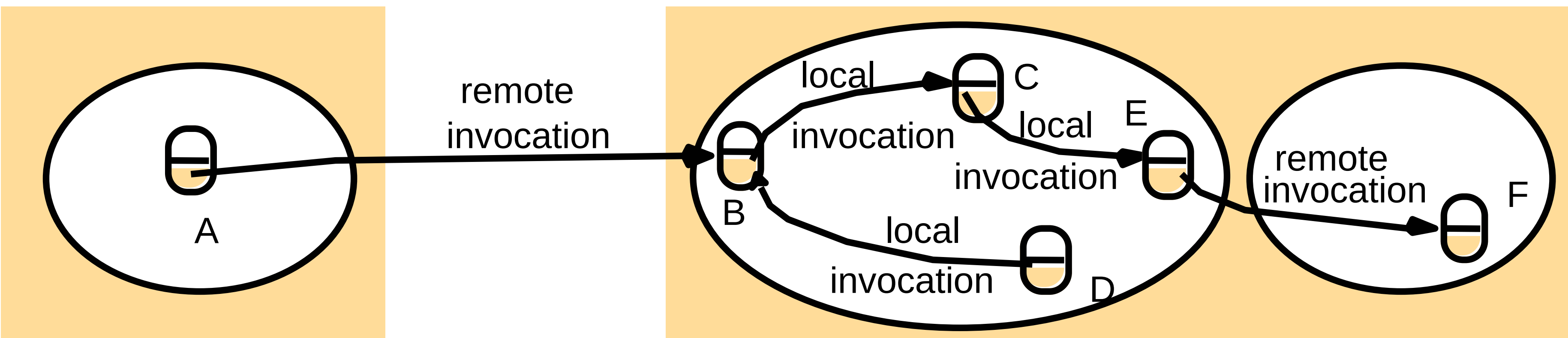
Semantics of "doOperation":

Idempotent operation: server simply (re-) executes operation.
Non-idempotent operation: server needs to maintain request history.

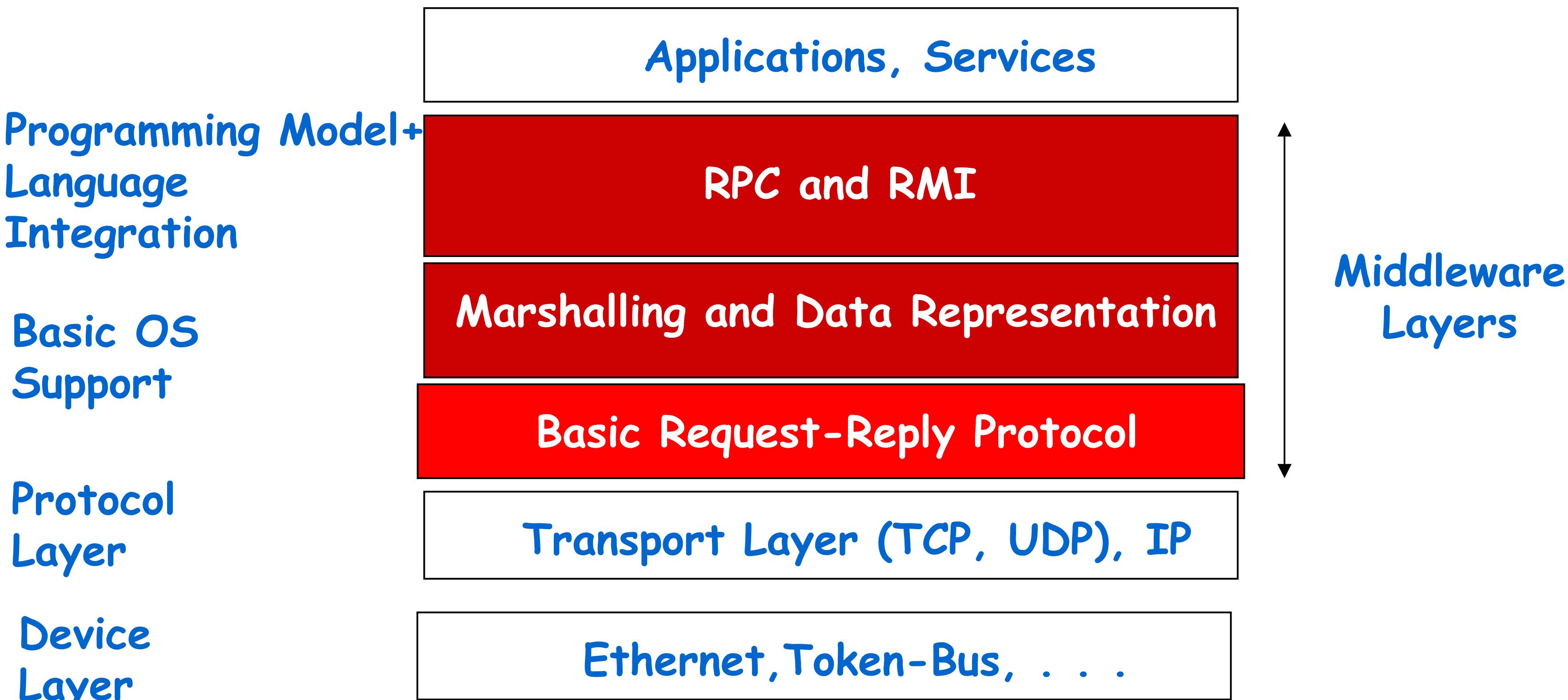
Removal of duplicated reply messages in the client.

Problems to solve

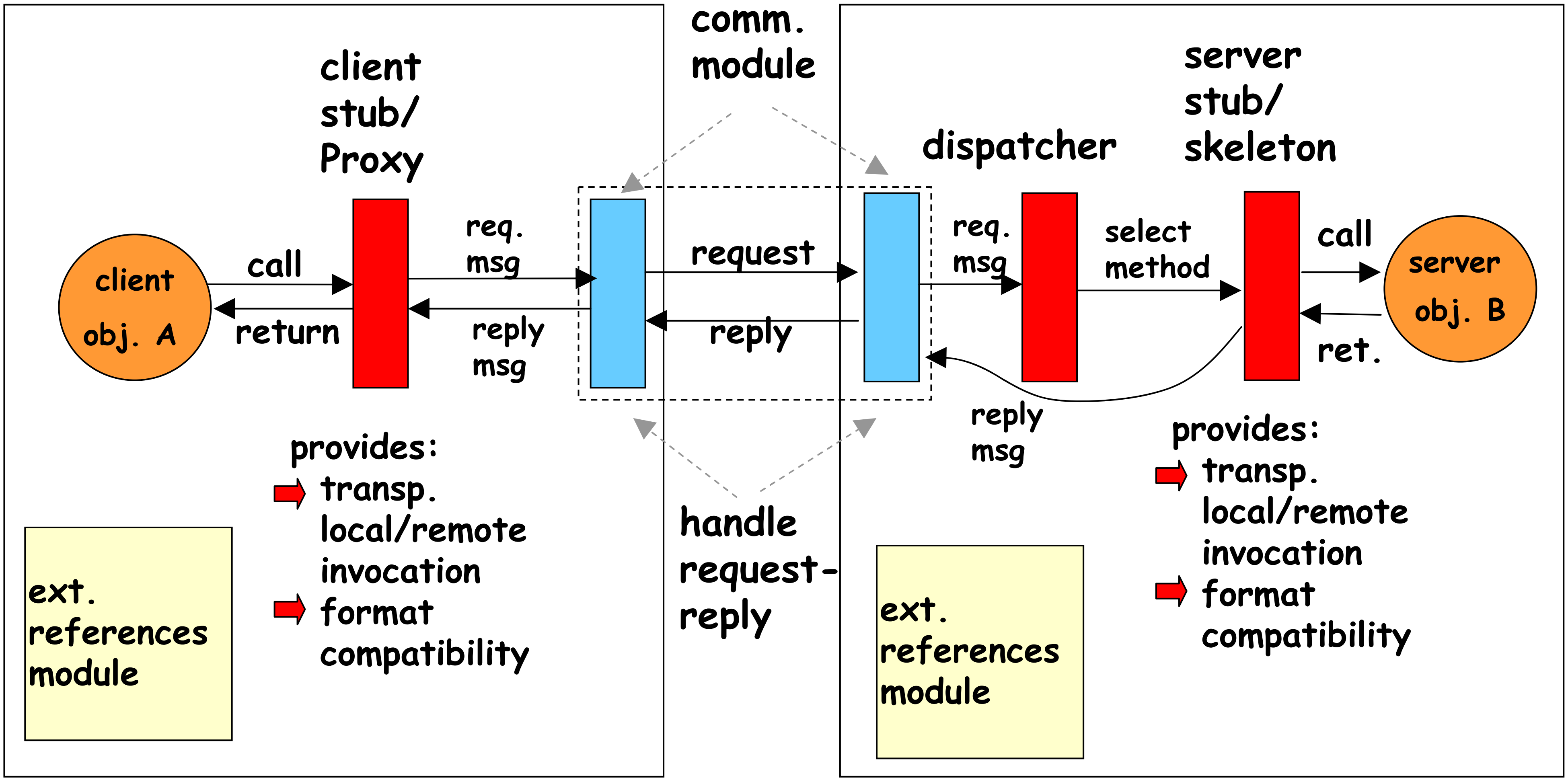
- Route invocation to the target object.
- Convert parameters into a compatible format.
  - Data Description
  - Marshalling ->External Data representation
- Enforce a well-defined invocation sematics wrt. faults.



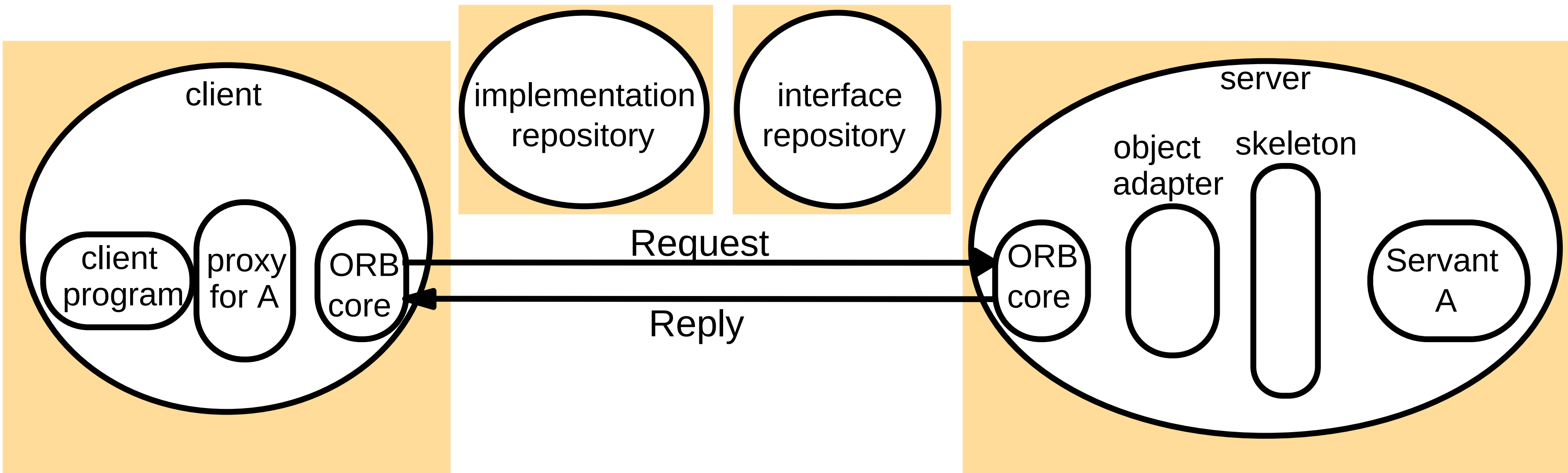
Distributed Objects and Remote Invocation



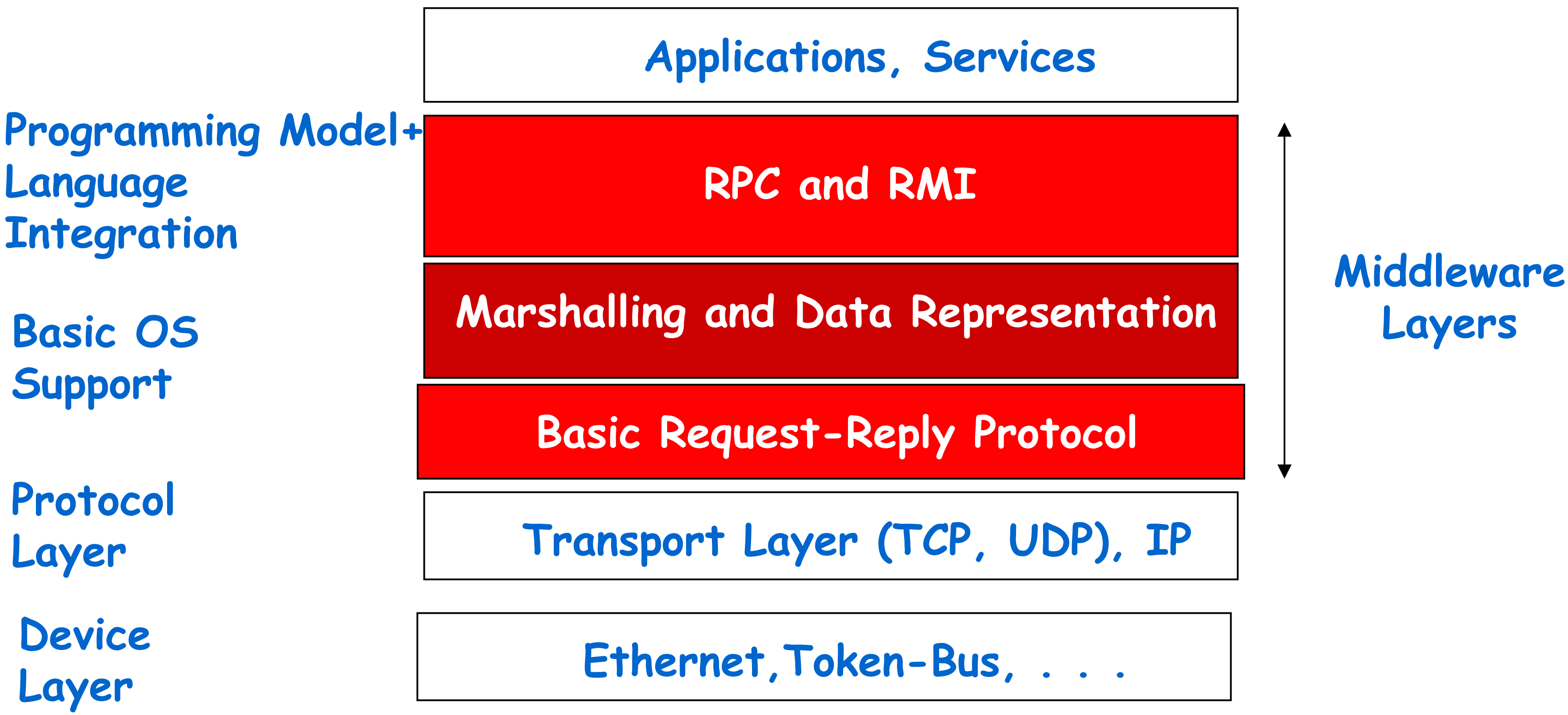
Remote {Method Invocation(RMI),Procedure Call (RPC)}



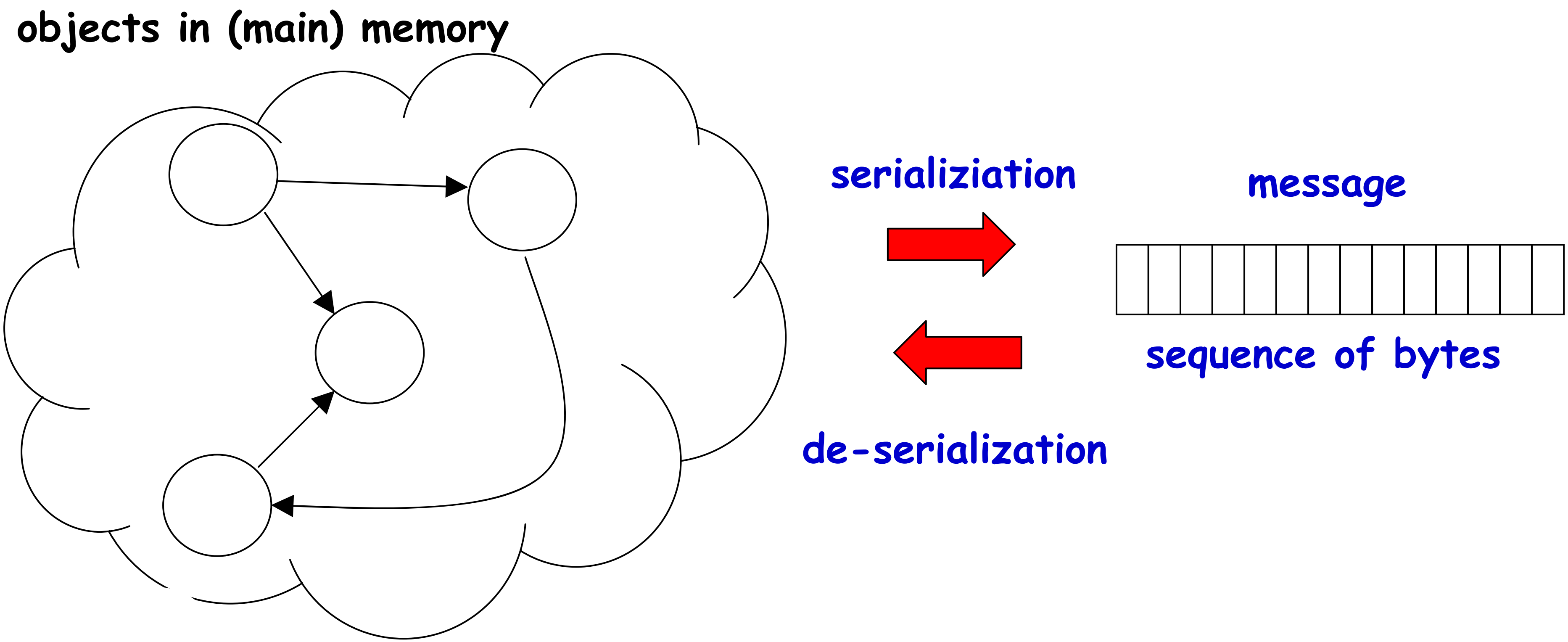
Components in the CORBA RMI



Distributed Objects and Remote Invocation



External Data Representation



Support for RPC and RMI requires for every data type which may be passed as a parameter or a result:

1. it has to be converted into a "flat" structure (of elementary data types).
2. the elementary data types must be converted to a commonly agreed format.

External Data Representation

Problems:

- multiple heterogeneous Hardware and OS Architecture
  - ⇨ little/big endian data representation
  - ⇨ different character encoding (ASCII, Unicode, EBCDIC)
- multiple programming laguages
  - ⇨ different representation and length of data types.

Solutions:

- Middleware defines common format for data representation and Specific middleware versions for hardware/OS-platform conversion.
  - ⇨ not practical for multiple programming languages
- Definition of common data format and bindings to the specific language.

External Data Representation

Platform Independent
heterogeneous
agree on a common way to describe the formats and representations

independent data representation and description

example: XML (character-oriented)

(Middleware-) Platform Specific
homogeneaous
agree on the same formats and representations

defined by the respective platform which may run on heterogeneous hardware and OS.

example: XDR, CDR (byte-oriented)

External Data Representation

```
<xs:element name="Event">
    <xs:complexType>
        <xs:sequence>
            <xs:element name="Subject" type="xs:string" />
            <xs:element name="SubjectUID" type="CODESID" />
            <xs:element name="Description" type="xs:string" minOccurs="0" />
            <xs:element ref="DataStructure" />
            <xs:element ref="MayTrigger" minOccurs="0" />
            <xs:element ref="WillTrigger" minOccurs="0" />
        </xs:sequence>
    </xs:complexType>
</xs:element>
```

```
<xs:simpleType name="CODESID">
    <xs:restriction base="xs:string">
        <xs:pattern value="0x[0-9A-Fa-f]{16}"/>
    </xs:restriction>
</xs:simpleType>
```

External Data Representation

Type	*Representation*
sequence	length (unsigned long) followed by elements in order
string	length (unsigned long) followed by characters in order (can also can have wide characters)
array	array elements in order (no length specified because it is fixed)
struct	in the order of declaration of the components
enumerated	unsigned long (the values are specified by the order declared)
union	type tag followed by the selected member

Corba CDR for Constructed Types

External Data Representation (Corba CDR)

index in sequence of bytes	← 4 bytes →	notes on representation
0–3	5	length of string
4–7	"Smit"	'Smith'
8–11	"h___"	
12–15	6	length of string
16–19	"Lond"	'London'
20-23	"on__"	
24–27	1934	unsigned long

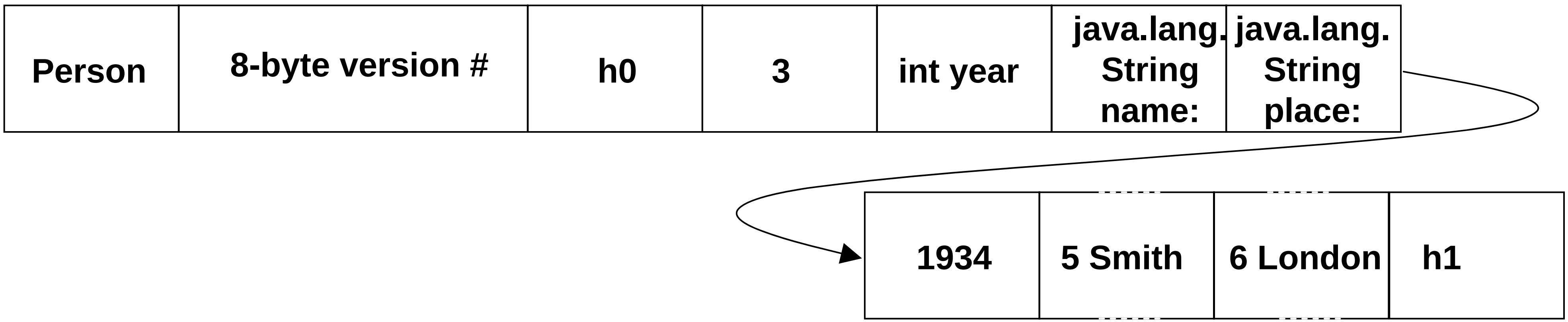
The flattened form represents a *Person* struct with value: {'Smith', 'London', 1934}

```
struct Person{
        string name;
        string place;
        long year;
};
```

CORBA CDR message

CORBA IDL description of the data structure

External Data Representation (Java)




```
public class Person implements Serializable {
        private String name;
        private String place;
        private String year;
            public Person(String aName, String aPlace, String aYear) {
                name= aName;
                place=aPlace;
                year= aYear;
        }
// followed by the methods to access the instance variables
}
```

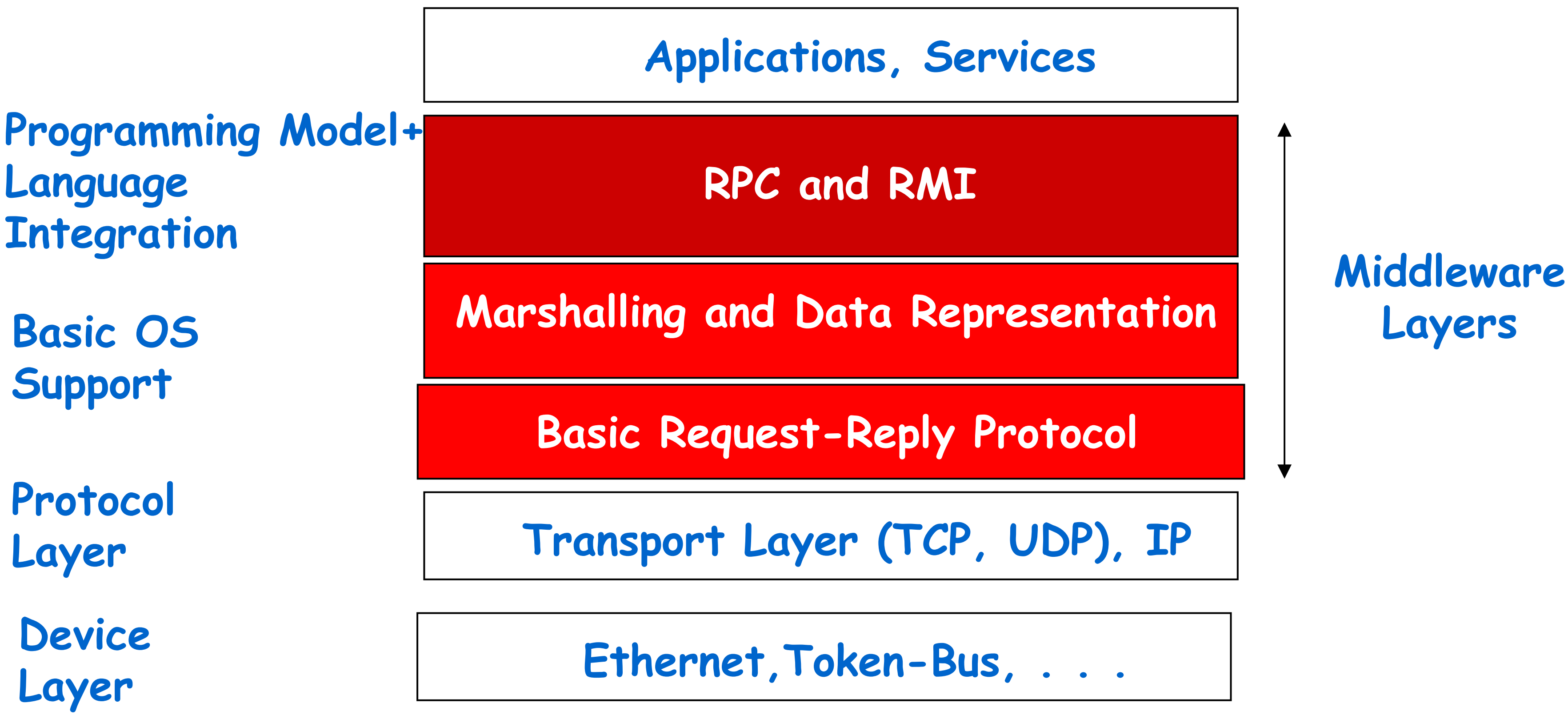
eXternal Data Representation example SUN

```
const MAX = 1000;
typedef int FileIdentifier;
typedef int FilePointer;
typedef int Length;
struct Data {
	int length;
	char buffer[MAX];
};
struct writeargs {
	FileIdentifier f;
	FilePointer position;
	Data data;
};
```

```
struct readargs {
	FileIdentifier f;
	FilePointer position;
	Length length;
};

program FILEREADWRITE {
  version VERSION {
	void WRITE(writeargs)=1;
	Data READ(readargs)=2;
  }=2;
} = 9999;
```

Distributed Objects and Remote Invocation



RMI Invocation Semantics

Exactly once sematics

- Approximates the semantics of a local procedure call.
- A procedure is executed exactly once.
- Very difficult to implement (efficiently) in the presence of network delays, lost messages or server failures. Needs fault-tolerance and forward error recovery.

RMI Invocation Semantics

repeat request	filter duplicates	execution of remote procedure	invocation semantics	Comments
= 0	no	#exec=1	exactly-once	very difficult to achieve, because of delays and faults.
= 0	no/n.a.	#exce $\leq$ 1	may be	simple, but application has to care about the cases which did not succeed
$\geq$ 0	no	#exec $\geq$ 1	at-least-once	simple, but application has to prevent multiple exec.+ duplicates
$\geq$ 0	yes	#exec $\leq$ 1	at-most-once	difficult to achieve, needs extensive fault-tolerance mechanism.

Distributed Objects and Remote Invocation

Module: --> Interface specifies procedures and variables.

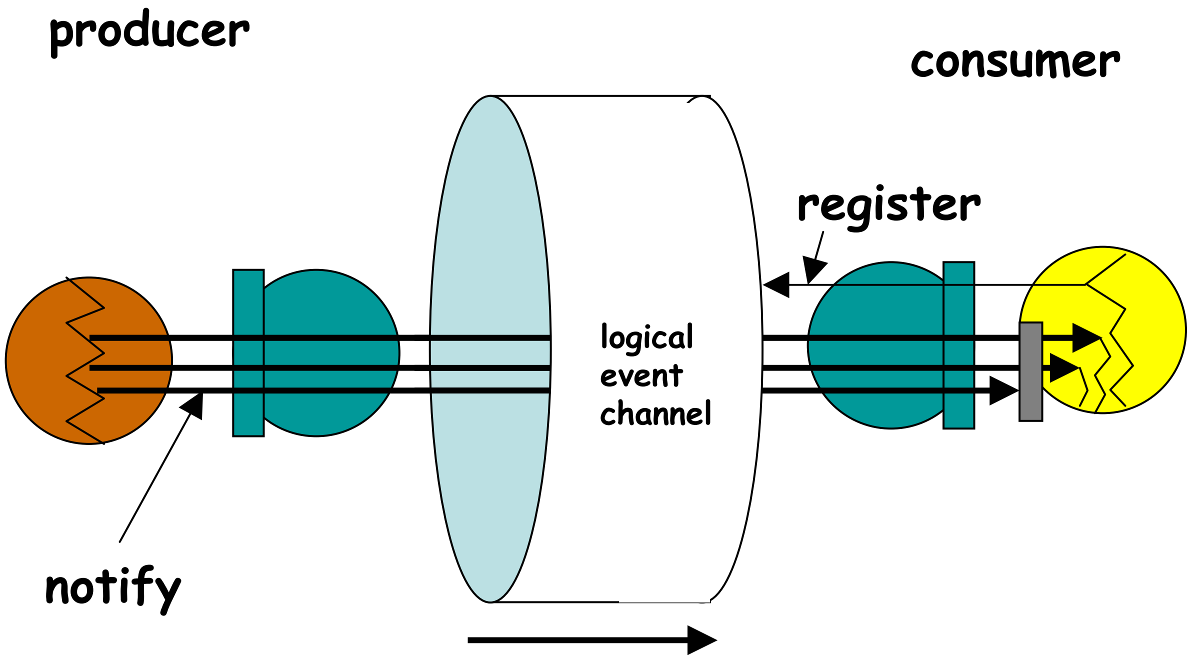
Service Interface: specifies the procedures of a server including arguments and return values.

Remote Interface: Like service interface.

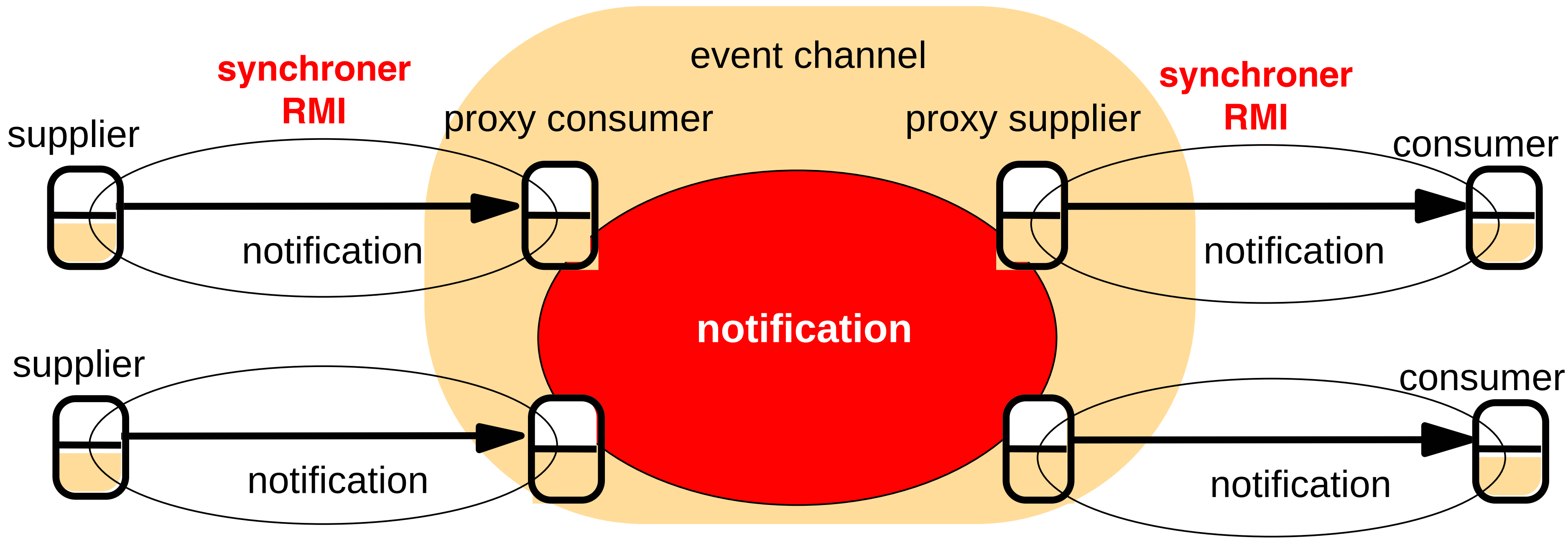
Difference:
- Objects can be passed as arguments to methods.
- Objects can be returned as results.
- Object references can be passed as parameters.

Modules:
--> No direct access to instance variables possible
--> Access only via procedure interface.

Corba Event Service



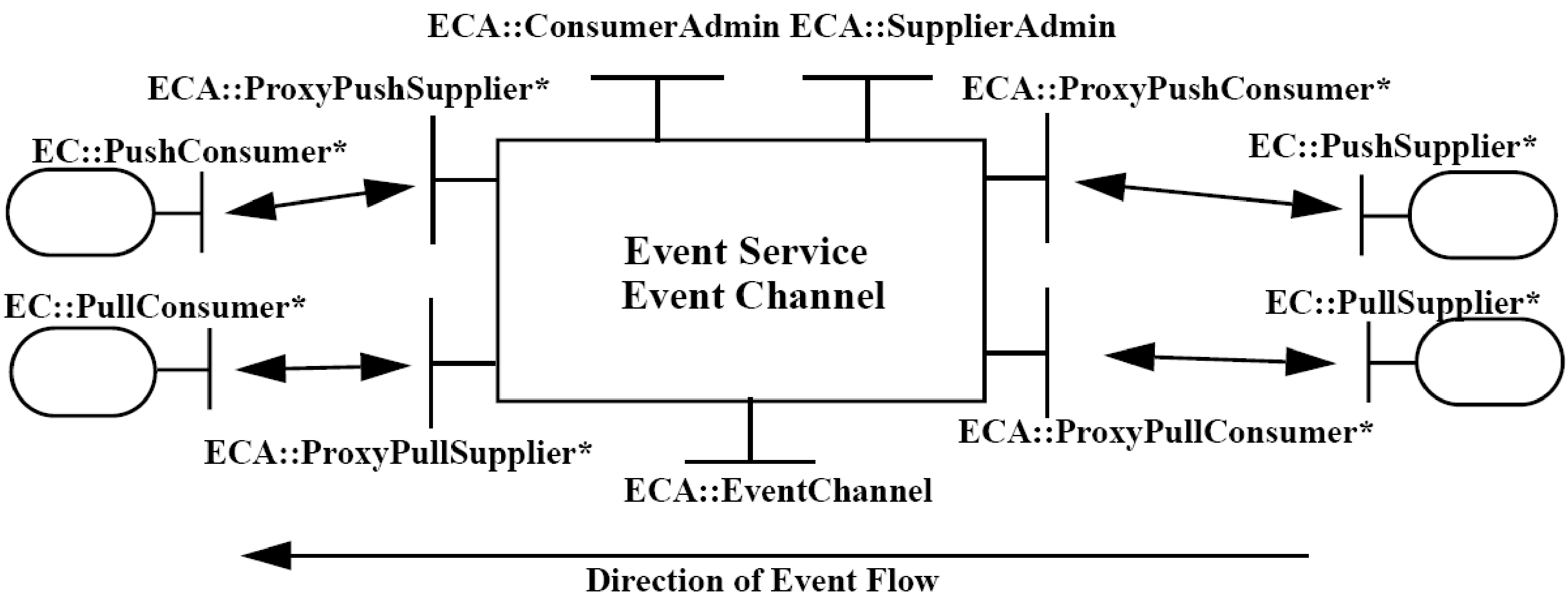
Corba Event Service



Asynchronous, anonymous event transfer from the supplier to the consumer.

push supplier, push consumer: supplier originated event transfer.
pull supplier, pull consumer: consumer originated event transfer.

Corba Event Service



Corba Event Service

Limitations of the event channel:

1. supports no event filtering capability, and
2. no ability to be configured to support different qualities of service.

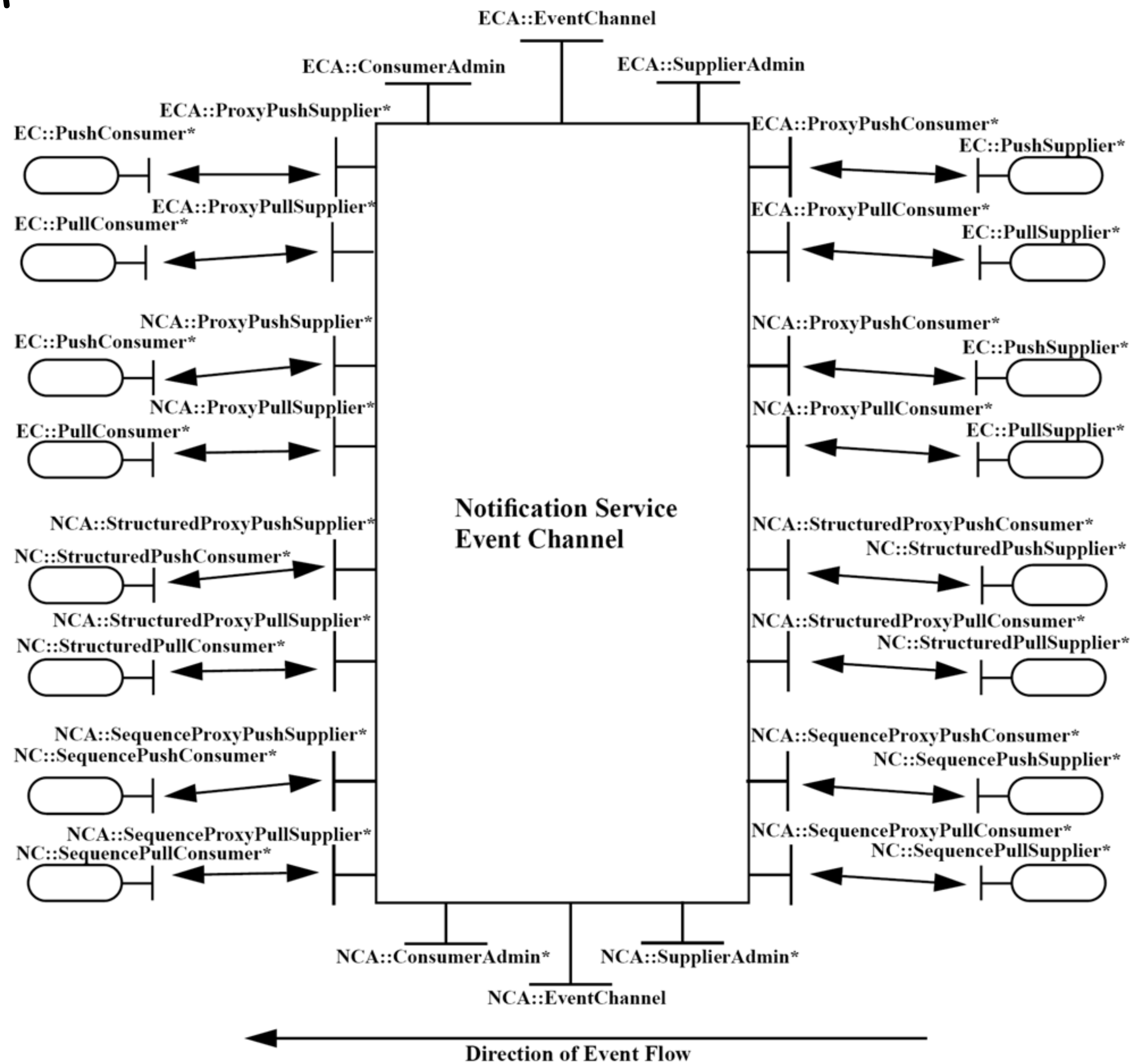
The Notification Service enhances the Event Service by introducing the concepts of filtering, and configurability according to various quality of service requirements.

Corba Notification Service

extends event service by:

- consumers can define filter objects to define which events they are interested in.
- quality properties of a channel can be configured, e.g. reliability properties or order preferences like FIFO or priorities.
- consumer can detect event types which are advertised by producers.
- producers can discover interests of the consumers
- optional event-type repository allows access to event structures. Supports definition of filter contraints.

Corba Notification Service



Corba Notification Service

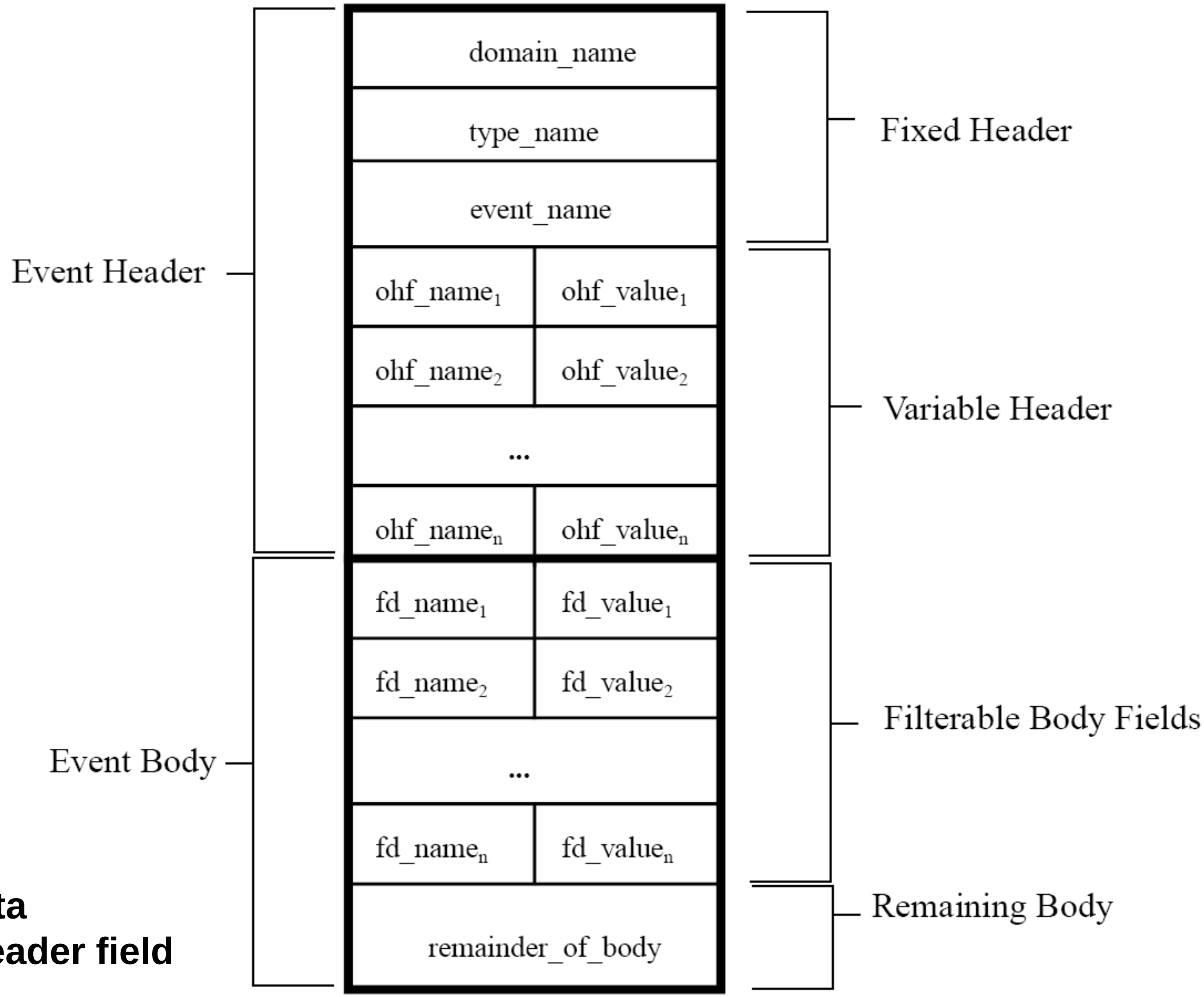
Structured Event:

event header				event body	
domain name	type name	event name	optional h-field(s)	filterable body fields	rest

Example:

health care	patient supervision	heartbeat low alarm 22.06.06 9:30	priority 10	ring alarm	blood pressure	respiration	rest

The structure of a Structured Event



fd: filterable data
ohf: optional header field