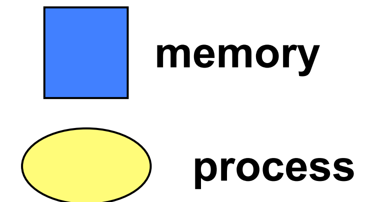
Remote Procedure Calls Layers, Problems and Variation

Principles of distributed computations

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



Distributed Processes

call proc.

P

comm. network

call proc.

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

The layers of IPC

	applications, services
Programming model+ language integration	RPC and RMI
basic support	Basic request-reply protocol marshalling and data representation
protocol layer	transport layer (TCP, UDP), IP
device layer	Ethernet,Token-Bus, . . .

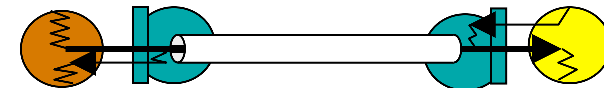
middleware layers (RPC and RMI; Basic request-reply protocol, marshalling and data representation)

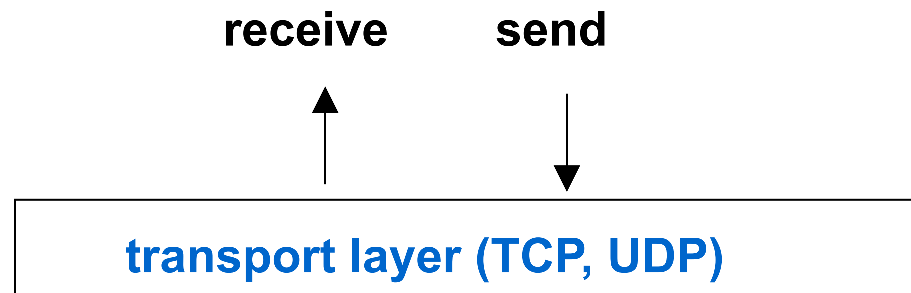
Abstractions of the Transport Layer

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

process

process

Internet-addr.: 144.44.25.222

Internet-addr.: 144.44.25.223

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

Examples of functions or methods typically provided by the API library]:

socket() creates a new socket of a certain socket type, identified by an integer number, and allocates system resources to it.

bind() is typically used on the server side, and associates a socket with a socket address structure, i.e. a specified local port number and IP address.

listen() is used on the server side, and causes a bound TCP socket to enter listening state.

connect() is used on the client side, and assigns a free local port number to a socket. In case of a TCP socket, it causes an attempt to establish a new TCP connection.

accept() is used on the server side. It accepts a received incoming attempt to create a new TCP connection from the remote client, and creates a new socket associated with the socket address pair of this connection.

send() and recv(), or write() and read(), or recvfrom() and sendto(), are used for sending and receiving data to/from a remote socket.

close() causes the system to release resources allocated to a socket. In case of TCP, the connection is terminated.

gethostbyname() and gethostbyaddr() are used to resolve host names and addresses.

select() is used to prune a provided list of sockets for those that are ready to read, ready to write or have errors

poll() is used to check on the state of a socket. The socket can be tested to see if it can be written to, read from or has errors.

Example: datagram sockets in Unix

```
s = socket(AF_INET, SOCK_DGRAM, 0)
.
.
bind (s, sender_address)
.
.
.
sento(s, message,L, receiver_address)
```

→

```
s = socket(AF_INET, SOCK_DGRAM, 0)
.
.
bind (s, receiver_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 (local) socket address <IP address, port number>.
sento:	system call to send a bit stream at memory location "message" of length L via socket “s“ to the specified server socket "receiver_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.

UDP client sends &receives msg

G. Coulouris, J. Dollimore, T. Kindberg: Verteite

```
import java.net.*;
import java.io.*;
public class UDPClient{
    public static void main(String args[]){
                // args give message contents and destination hostname
                DatagramSocket aSocket = null;
                try {
                        aSocket = new DatagramSocket();
                        byte [] m = args[0].getBytes();
                        InetAddress aHost = InetAddress.getByName(args[1]);
                        int serverPort = 6789;
                        DatagramPacket request = new DatagramPacket(m,  args[0].length(), aHost, serverPort);
                        aSocket.send(request);
                        byte[] buffer = new byte[1000];
                        DatagramPacket reply = new DatagramPacket(buffer, buffer.length);
                        aSocket.receive(reply);
                        System.out.println("Reply: " + new String(reply.getData()));
                }catch (SocketException e){System.out.println("Socket: " + e.getMessage());
                }catch (IOException e){System.out.println("IO: " + e.getMessage());
                }finally {if(aSocket != null) aSocket.close();}
        }
}
```

UDP server receives requ. and sends reply

```
import java.net.*;
import java.io.*;
public class UDPServer{
    public static void main(String args[]){
            DatagramSocket aSocket = null;
              try{
                    aSocket = new DatagramSocket(6789);       // create socket at agreed port
                    byte[] buffer = new byte[1000];
                    while(true){
                      DatagramPacket request = new DatagramPacket(buffer, buffer.length);
                      aSocket.receive(request);
                      DatagramPacket reply = new DatagramPacket(request.getData(),
                      request.getLength(), request.getAddress(), request.getPort());
                      aSocket.send(reply);
                    }
              }catch (SocketException e){System.out.println("Socket: " + e.getMessage());
              }catch (IOException e) {System.out.println("IO: " + e.getMessage());
              }finally {if(aSocket != null) aSocket.close();}
    }
}
```

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 socket for 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 the number of bytes sent successfully.
read receives a byte stream in the respective buffer and returns the number of received bytes.

TCP client side

client establishes connection, sends request and receives reply

```
import java.net.*;
import java.io.*;
public class TCPClient {
    public static void main (String args[]) {
        // arguments supply message and hostname
            Socket s = null;
            try{
                int serverPort = 7896;
                        s = new Socket(args[1], serverPort);
                            DataInputStream in = new DataInputStream( s.getInputStream());
                DataOutputStream out =new DataOutputStream( s.getOutputStream());
                            out.writeUTF(args[0]);          // UTF is a string encoding see Sn. 4.4
                            String data = in.readUTF();        // read a line of data from the stream
                            System.out.println("Received: "+ data) ;
            }catch (UnknownHostException e){System.out.println("Socket:"+e.getMessage());
            }catch (EOFException e){System.out.println("EOF:"+e.getMessage());
            }catch (IOException e){System.out.println("readline:"+e.getMessage());
            }finally {if(s!=null) try {s.close();}catch (IOException e){System.out.println("close:"+e.getMessage());}}
    }
}
```

TCP server side

server establishes connection for each client

```
import java.net.*;
import java.io.*;
public class TCPServer {
        public static void main (String args[]) {
                try{
                        int serverPort = 7896; // the server port
                        ServerSocket listenSocket = new ServerSocket(serverPort);
                        while(true) {
                                Socket clientSocket = listenSocket.accept();
                                Connection c = new Connection(clientSocket);
                        }
                } catch(IOException e) {System.out.println("Listen socket:"+e.getMessage());}
        }
}
```

TCP server side

server indicates request of a client

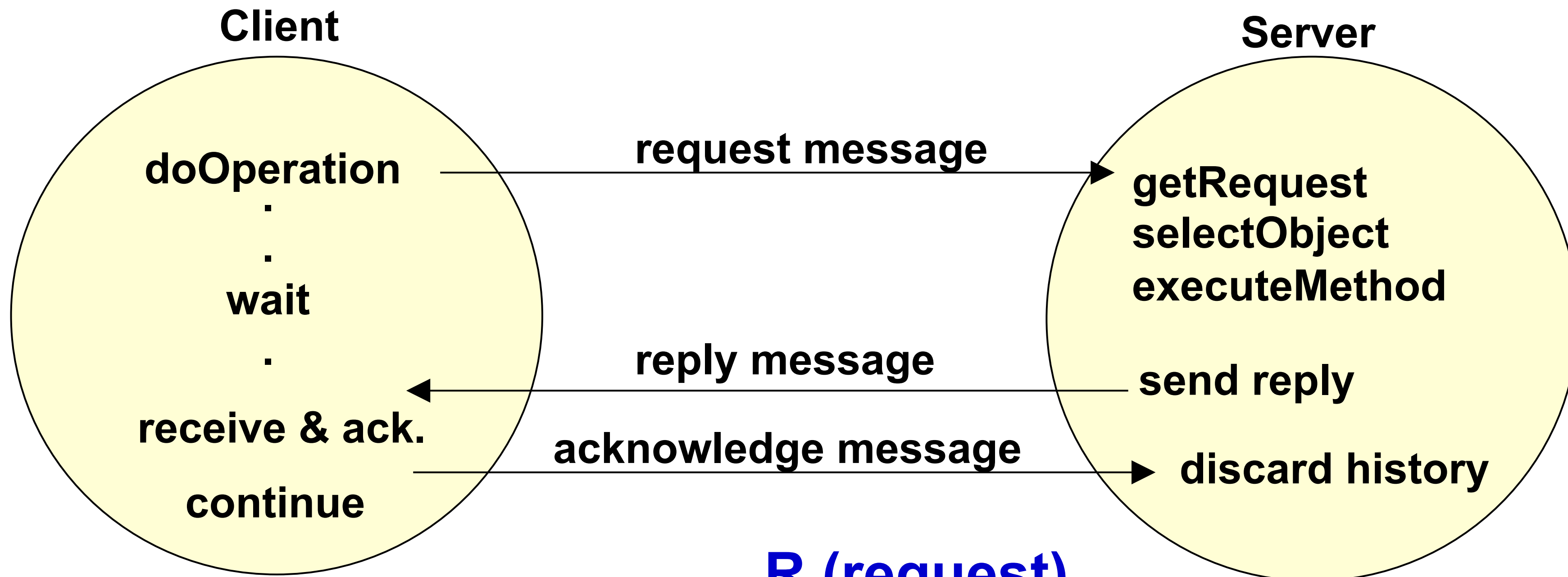
```
class Connection extends Thread {
        DataInputStream in;
        DataOutputStream out;
        Socket clientSocket;
        public Connection (Socket aClientSocket) {
                try {
                        clientSocket = aClientSocket;
                        in = new DataInputStream( clientSocket.getInputStream());
                        out =new DataOutputStream( clientSocket.getOutputStream());
                        this.start();
                        } catch(IOException e)
                                {System.out.println("Connection:"+e.getMessage());}
        }
        public void run(){
                try {                                        // an echo server

                  String data = in.readUTF();                // read a line of data from the stream
                  out.writeUTF(data);
                } catch (EOFException e){System.out.println("EOF:"+e.getMessage());
                } catch(IOException e) {System.out.println("readline:"+e.getMessage());
                } finally{ try {clientSocket.close(); }catch (IOException e){/*close failed*/}}
        }
}
```

Distributed Objects and Remote Invocation

	Layer	
	Applications, Services	
Programming Model+ Language Integration	RPC and RMI	Middleware Layers
Basic Support	Marshalling and Data Representation	
	Basic Request-Reply Protocol	
Protocol Layer	Transport Layer (TCP, UDP), IP	
Device Layer	Ethernet,Token-Bus, . . .	

Request-Reply Communication



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

Coulouris, Dollimore and Kindberg Distributed Systems: Concepts and Design Edn. 3
© Addison-Wesley Publishers 2000

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.

© Addison-Wesley Publishers 2000

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":

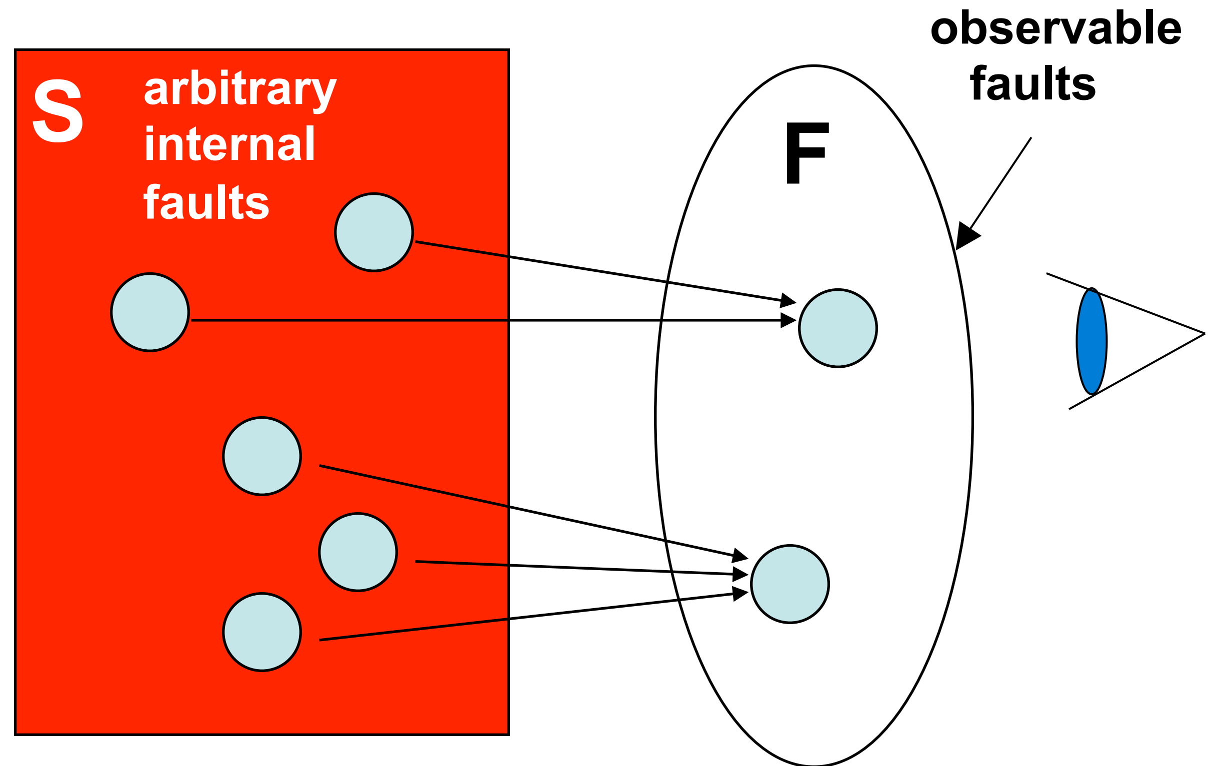
- Idempotent operation: server simply (re-) executes operation.
- Non-idempotent operation: server needs to maintain request history.

Removal of duplicated reply messages in the client.

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

temporal domain only | temporal + value domain

fail stop ⊂ crash ⊂ omission ⊂ timing (performance) ⊂ value ⊂ byzantine

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.

Remote Procedure Call

node 1

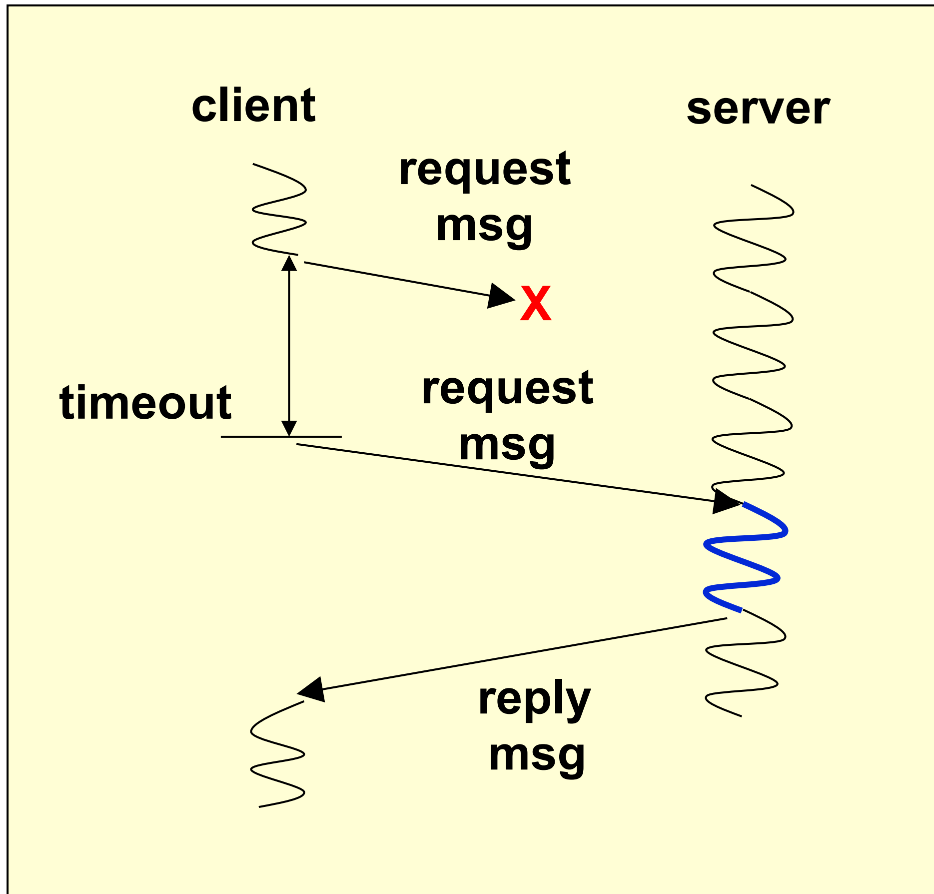
node 2

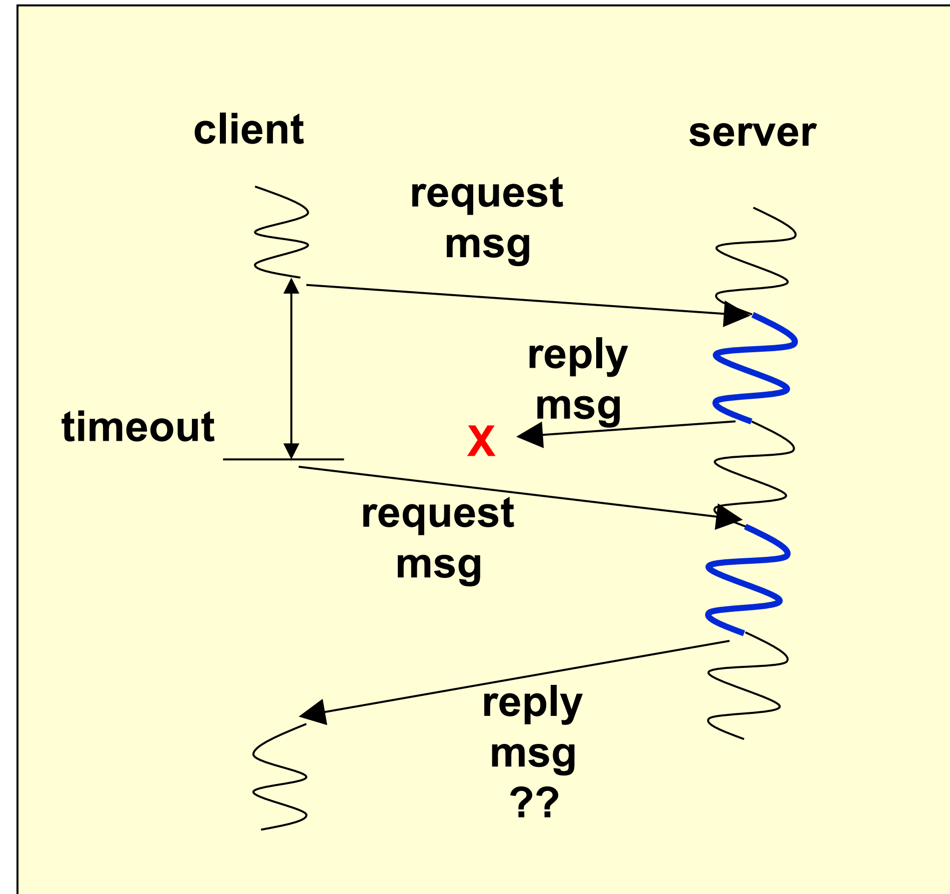
node 3

client stub/ proxy

server stub/ skeleton

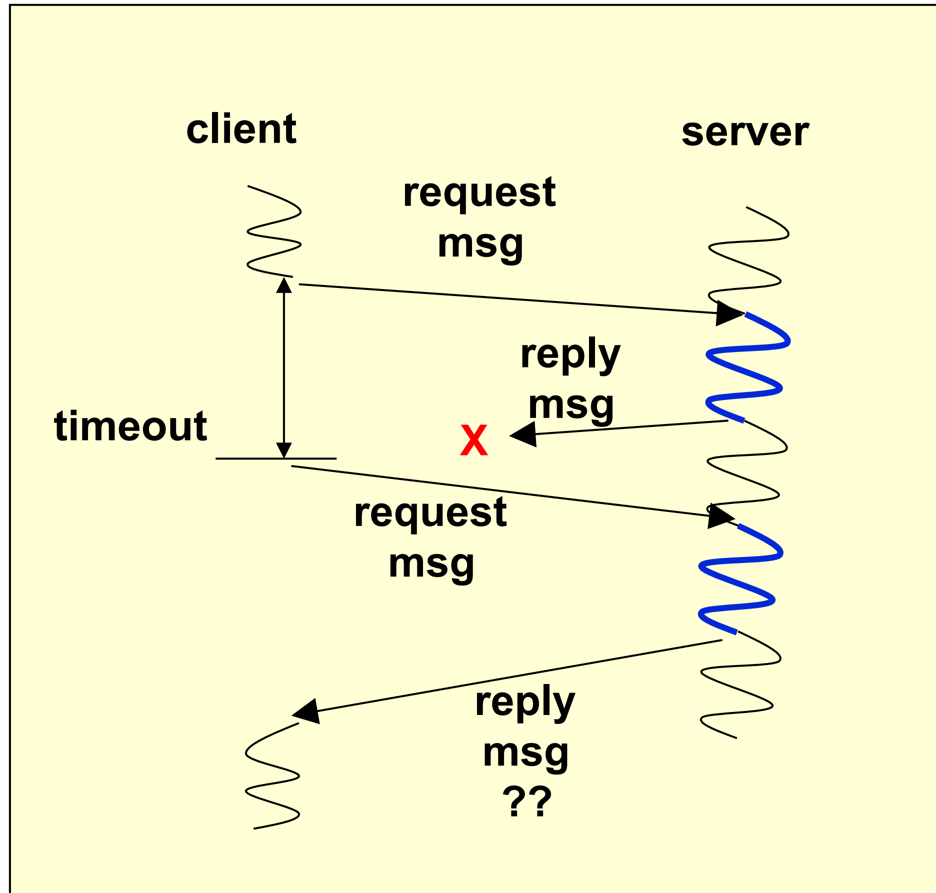
RPC Semantics





additional mechanisms needed to deal with failures.

RPC Semantics



add 5 to old value;
return new value;

add 5

add 5

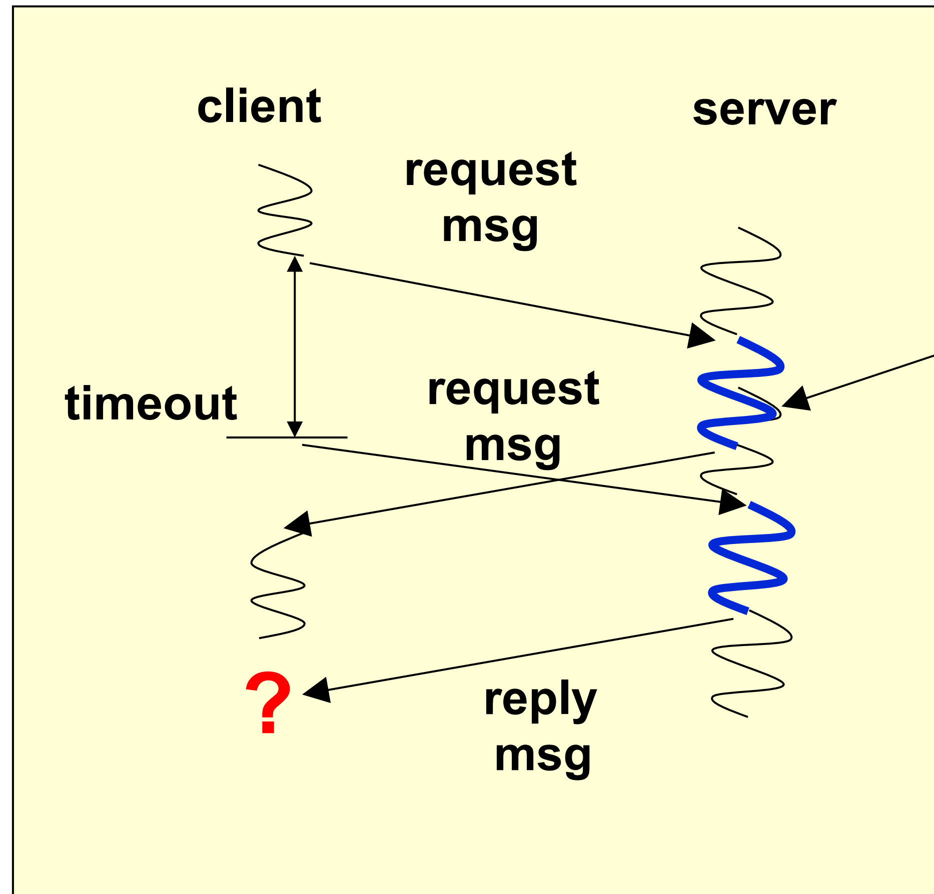
return
new value =
old value+10
??

RPC Semantics

which to select?

return value =
old value+5

return value =
old value+10



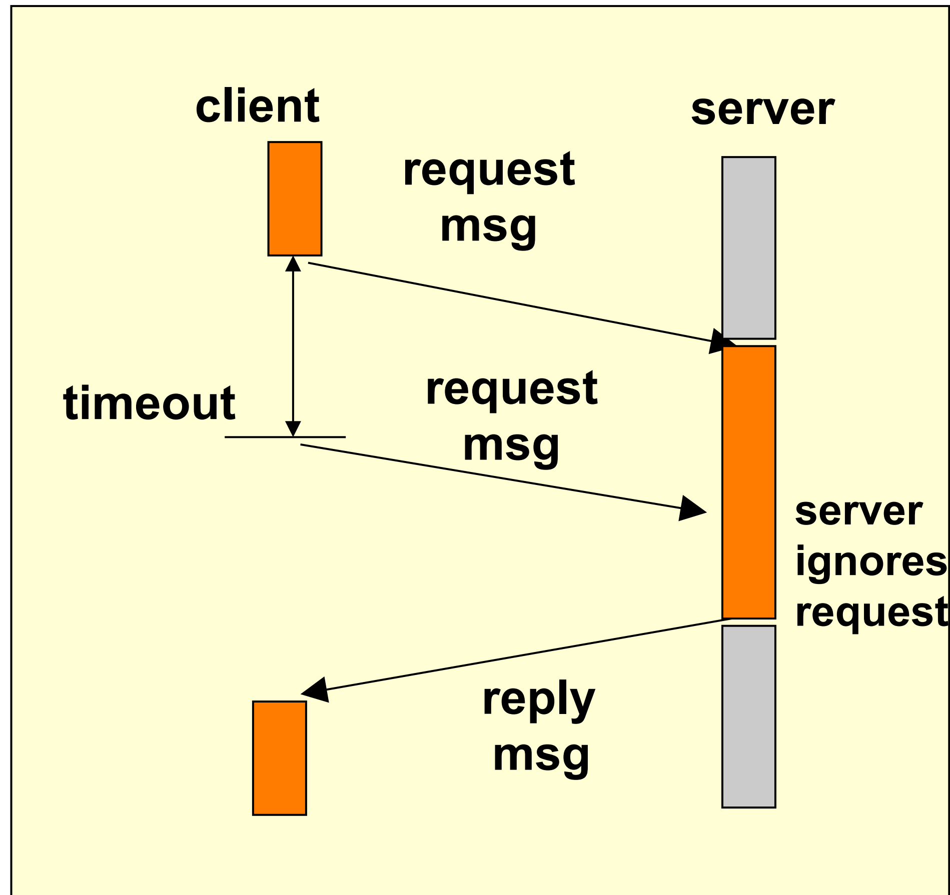
add 5 to account;
return new value;

slow server

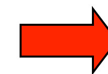
add 5

add 5

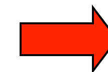
RPC Semantics



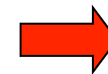
Options:



sequence numbers to identify request messages.



idempotent operations



save call history

RMI Invocation Semantics

Goal: Transparency of local and remote procedure call

- **Approximates the semantics of a local procedure call.**
- **A procedure is executed exactly once.**

Can we achieve exactly once semantics ?

RMI Invocation Semantics

Alternatives?

- Wait until the server comes up again. The server sends replies until the client has received it at least once.

  Example: SUN RPC

- The client generates an error after a time out. No later replies are accepted. At most once.

  Examples: Java RMI and CORBA

- May be semantics. Used by CORBA for remote method invocation that doesn't deliver results back to the client.

Failures in an RPC

1.Client unable to locate the server

2. Request message lost

3. Server crashes after receiving the request

4 Reply message is lost

5. Client crashes after sending request

Example

Client sends request to the server to print a text

Server acknowledgement policies:
- Server sends an ack when request is received.
- Additionally, the server sends a completion message:
either - when text has been sent to printer
or - when text has been printed successfully

Server crashes, recovers and sends a message that it is up again.

Client reaction policies:
C1: client always re-issues request
C2: client never re-issues request
C3: client only re-issues if it received an ack for the print request
C4: client only re-issues if no ack

Example

M: Completion message
P: Print
C: Crash

Possible Combinations:

M → P → C
M → C (→ P)
P → M → C
P → C (→ M)
C (→ P → M)
C (→ M → P)

	Server policy					
	M → P			P → M		
	MPC	MC(P)	C	PMC	PC(M)	C
C1	DUP	✔	✔	DUP	DUP	✔
C2	✔	-	-	✔	✔	-
C3	DUP	✔	-	DUP	✔	-
C4	✔	-	✔	✔	DUP	✔

✔: text printed once
-: text never printed

Bottom Line !

1.) Client can never know whether server crashed before printing

2.) Possibility of independent client and server crashes radically changes the nature of RPC and clearly distinguishes single processor systems from distributed systems.

Orphans !

Client crashes before server reply

Policies:

- **extermination**
- **reincarnation**
- **expiration**

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.	#excec≤1	may be	simple, but application has to care about the cases which did not succeed
≥ 0	yes	#exec≥1	at-least-once	simple, but application has to prevent multiple exec.+ duplicates
= 0	no	#exec≤1	at-most-once	remote operation may not be executed at all. Late results must be deleted.

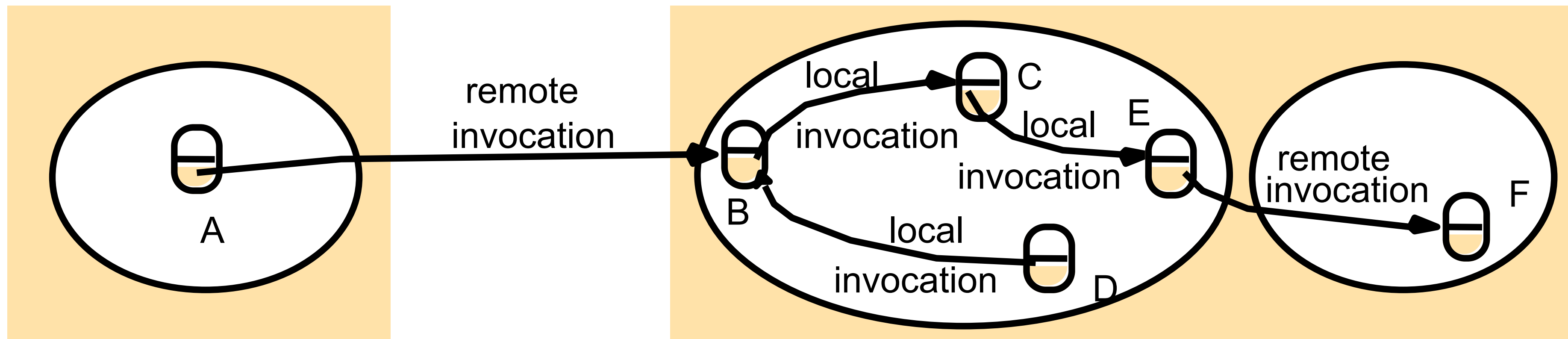
Distributed Objects and Remote Invocation

	Applications, Services
Programming Model+ Language Integration	RPC and RMI
	Marshalling and Data Representation
Basic OS Support	Basic Request-Reply Protocol
Protocol Layer	Transport Layer (TCP, UDP), IP
Device Layer	Ethernet,Token-Bus, . . .

Middleware Layers (RPC and RMI; Marshalling and Data Representation; Basic Request-Reply Protocol)

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.**



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

comm.
module

client
stub/
Proxy

server
stub/
skeleton

dispatcher

client obj. A — call → client stub/Proxy; return ←

req. msg → ; ← reply msg

request → ; ← reply

req. msg → dispatcher → select method → server stub/skeleton → call → server obj. B; ← ret.

reply msg

provides:
- transp. local/remote invocation
- format compatibility

handle
request-
reply

remote
references
module

provides:
- transp. local/remote invocation
- format compatibility

remote
references
module

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

(in the ORB kernel)

comm. module

request

reply

req. msg

external reference module

remote ref – local ref (proxy)
local ref -- remote ref (proxy)

messageType
requestId
objectReference
methodId
arguments

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

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

client stub/ Proxy

client obj. A

call

return

req. msg

reply msg

provides:
- transp. local/remote invocation
- format compatibility

remote references module

req reply

(in Corba: Object Adapter) dispatcher

server stub/ skeleton

req. msg

select method

call

ret.

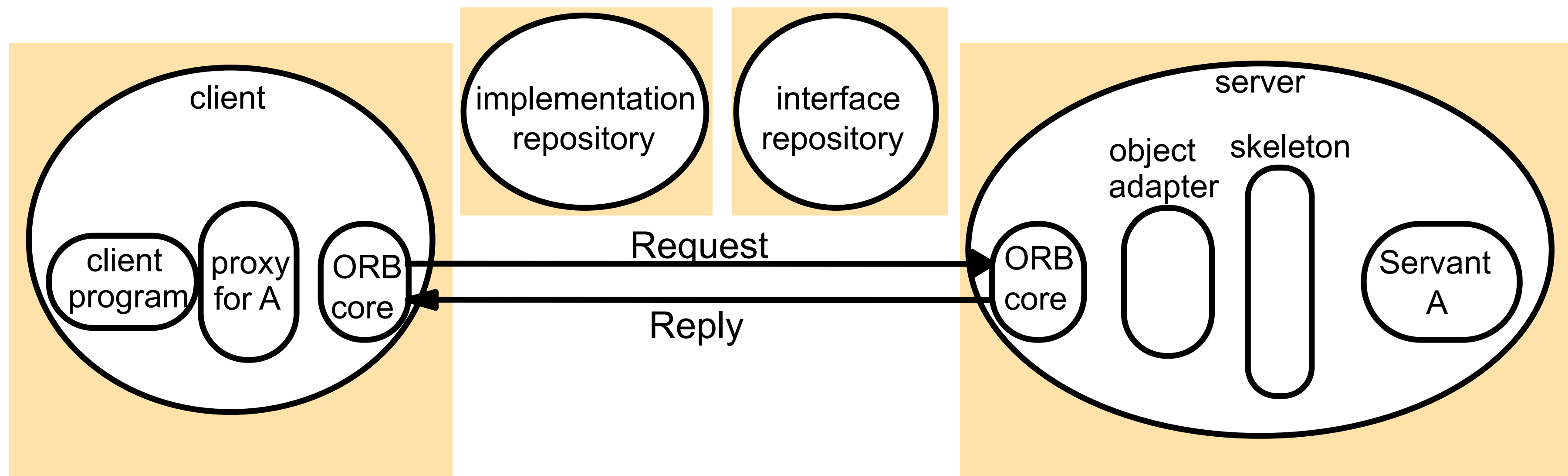
server obj. B

reply msg

provides:
- transp. local/remote invocation
- format compatibility

remote references module

Components in the CORBA RMI



Instructor's Guide for Coulouris, Dollimore and Kindberg Distributed Systems: Concepts and Design Edn. 3

© Addison-Wesley Publishers 2000

Distributed Objects and Remote Invocation

	Applications, Services	
Programming Model+ Language Integration	RPC and RMI	Middleware Layers
Basic OS Support	Marshalling and Data Representation	Middleware Layers
	Basic Request-Reply Protocol	Middleware Layers
Protocol Layer	Transport Layer (TCP, UDP), IP	
Device Layer	Ethernet,Token-Bus, . . .	

External Data Representation

objects in (main) memory

serializiation

message

sequence of bytes

de-serialization

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

(Middleware-)
Platform Specific
homogeneous
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)

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

independent data representation and description

example: XML (character-oriented)

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*

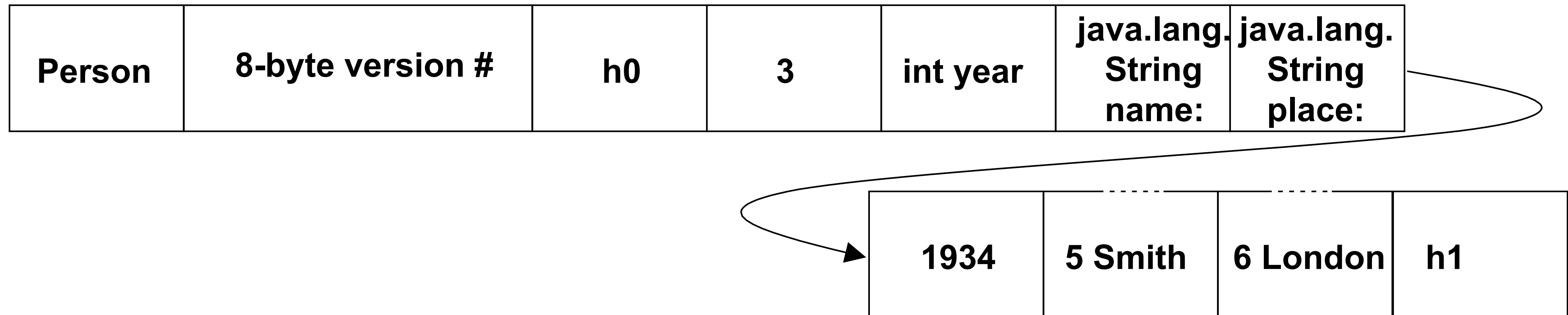
The flattened form represents a *Person* struct with value: {'Smith', 'London', 1934}

CORBA CDR message

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

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;
```

External Data Representation (P-independent)

```
<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>
```

Distributed Objects and Remote Invocation

	Layer	
	Applications, Services	
Programming Model+ Language Integration	RPC and RMI	Middleware Layers
Basic OS Support	Marshalling and Data Representation	
	Basic Request-Reply Protocol	
Protocol Layer	Transport Layer (TCP, UDP), IP	
Device Layer	Ethernet,Token-Bus, . . .	