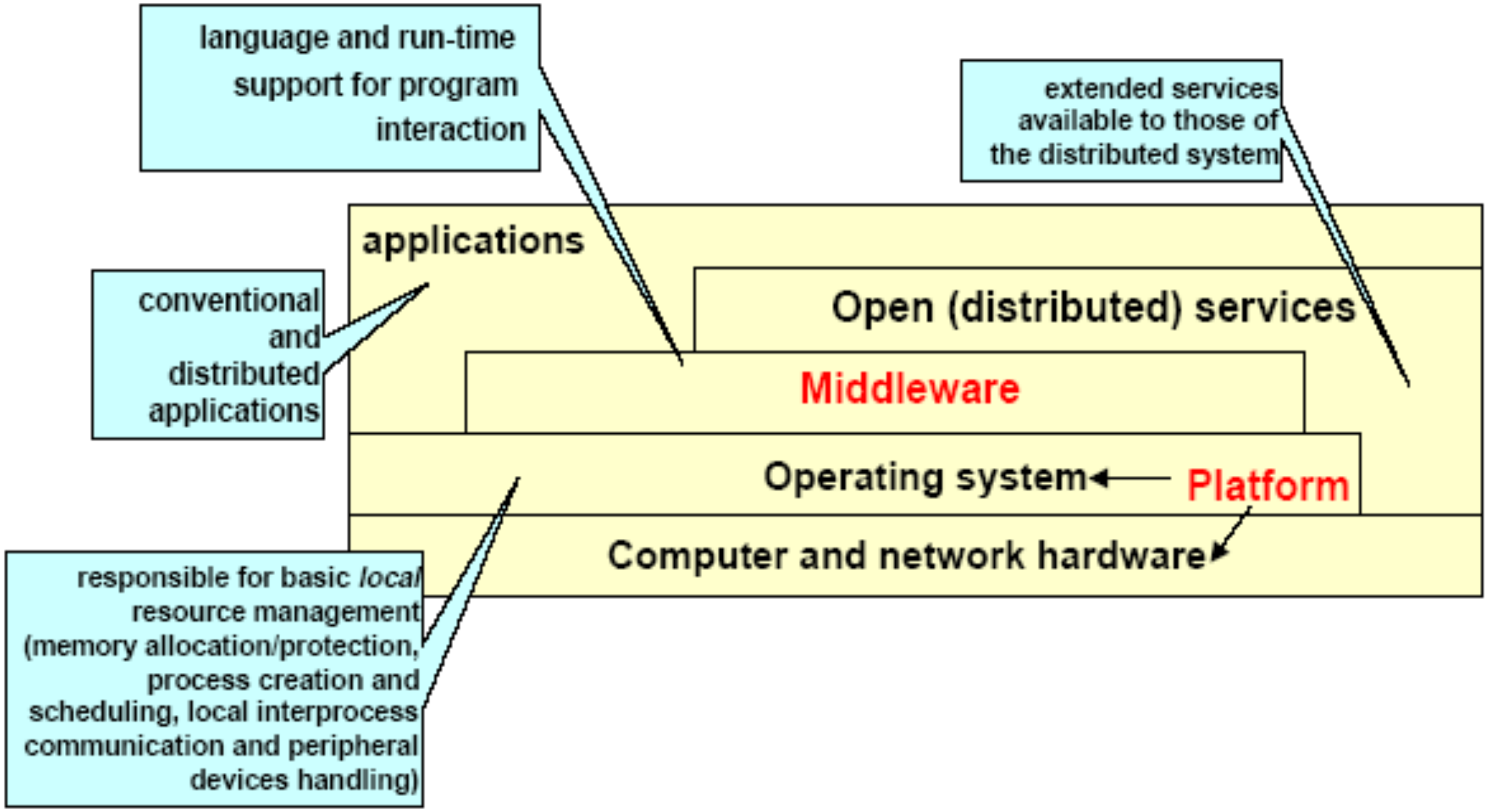
Distributed Systems

a.y. 2023/2024

Distributed Systems: lecture 7

Middleware

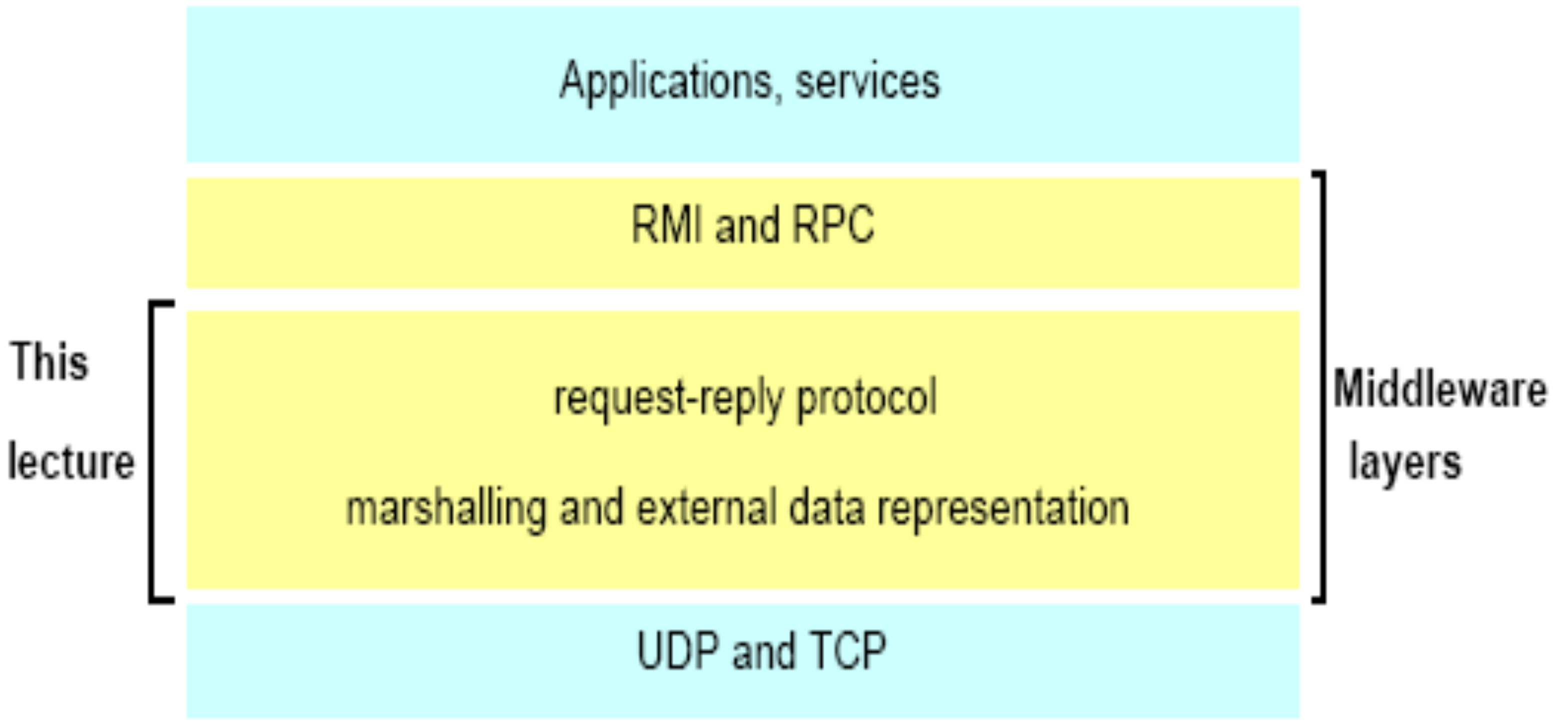
Software Layers


language and run-time support for program interaction
extended services available to those of the distributed system
applications
conventional and distributed applications
Open (distributed) services
Middleware
Operating system
Platform
Computer and network hardware
responsible for basic *local* resource management (memory allocation/protection, process creation and scheduling, local interprocess communication and peripheral devices handling)

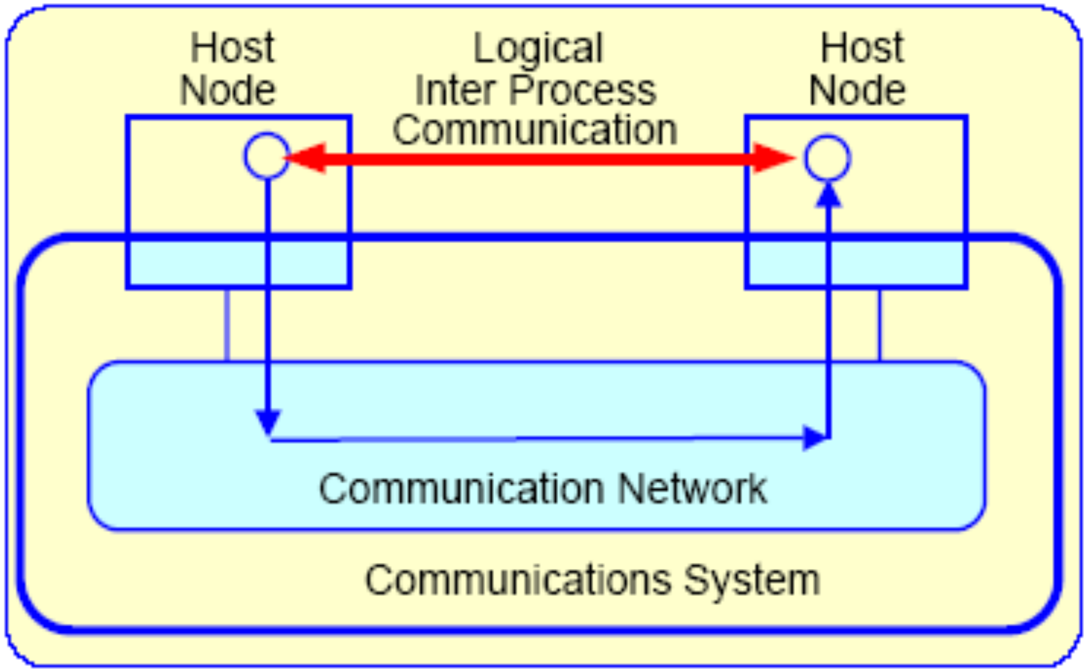
Inter-Process Communication

- … Middleware level … It provides programming language support, i.e., it does not support low-level untyped data primitives (OS) and implements higher-level language primitives + typed data
- Support for communication between objects
  remote method invocation (Java RMI), or
  remote procedure call (Sun RPC)
- Client-server
- Group communication

API for Internet programming...



Inter-process communication



Possibly several processes on each host (use ports).
Send and receive primitives.

Inter-Process Communication

Message passing

- send, receive, group communication
- synchronous versus asynchronous
- types of failure, consequences
- socket abstraction

Java API for sockets

- connectionless communication (UDP)
- connection-oriented communication (TCP)

Communication service types

- Connectionless: UDP
  - ‘send and pray’ unreliable delivery
  - efficient and easy to implement
- Connection-oriented: TCP
  - with basic reliability guarantees
  - less efficient, memory and time overhead for error correction

Connectionless service

UDP (User Datagram Protocol)

– messages possibly lost, duplicated, delivered out of order,

– maintains no state information, so cannot detect lost, duplicate or out-of-order messages

– may discard corrupted messages due to no error correction (simple checksum) or congestion

- Used e.g. for DNS (Domain Name System).

Connection-oriented service

TCP (Transmission Control Protocol)

– establishes data stream connection to ensure reliable, in-sequence delivery

– error checking and reporting to both ends

– attempts to match speeds (timeouts, buffering)

– sliding window: state information includes

- unacknowledged messages
- message sequence numbers
- flow control information (matching the speeds)

Used e.g. for HTTP, FTP, SMTP on Internet.

Timing issues

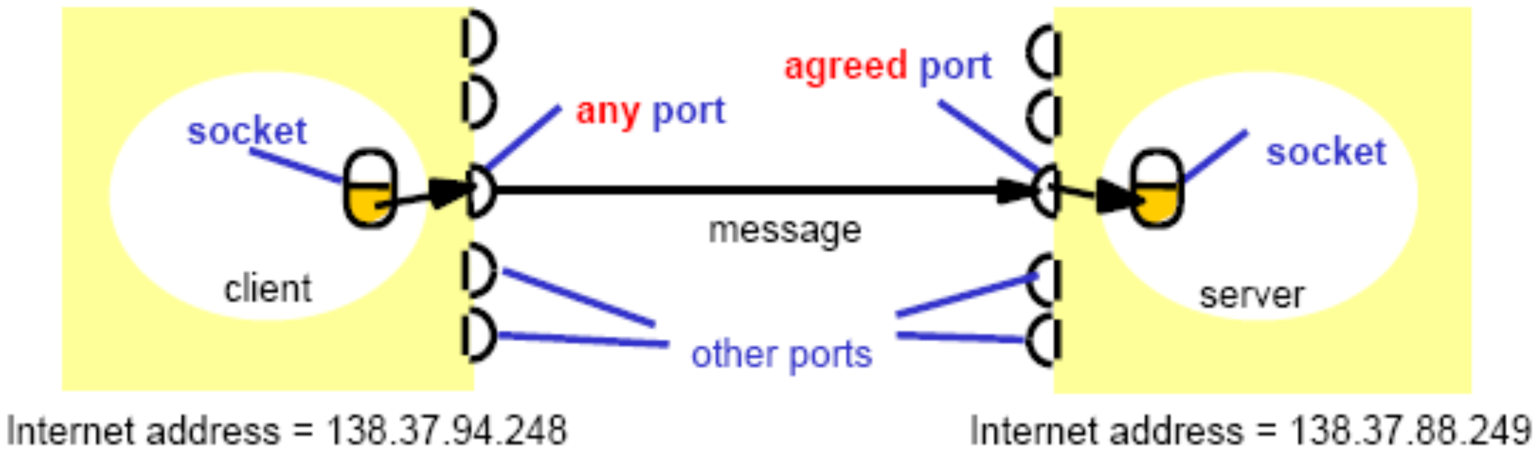
- Computer clocks
  - – may have varying drift rate
  - – rely on GPS radio signals (not always reliable), or synchronise via clock synchronisation algorithms
- Event ordering (message sending, arrival)
  - – carry timestamps
  - – may arrive in wrong order due to transmission delays

Types of interaction

- Synchronous interaction model:
  - known upper/lower bounds on execution speeds, message transmission delays and clock drift rates
  - more difficult to build, conceptually simpler model

- Asynchronous interaction model (more common, cf Internet, more general):
  - arbitrary process execution speeds, message transmission delays and clock drift rates
  - some problems impossible to solve (e.g. agreement)
  - if solution valid for asynchronous then also valid for synchronous.

Sockets and ports


socket
client
any port
agreed port
socket
server
message
other ports
Internet address = 138.37.94.248
Internet address = 138.37.88.249

Socket = Internet address + port number.
Only one receiver but multiple senders per port.
Disadvantages: location dependence (but see Mach study, chap 18)
Advantages: several points of entry to process.

Sockets

- Characteristics:
  - endpoint for inter-process communication
  - message transmission between sockets
  - socket associated with either UDP or TCP
  - processes bound to sockets, can use multiple ports
  - no port sharing unless IP multicast
- Implementations
  - originally BSD Unix, but available in Linux, Windows,…
  - here Java API for Internet programming

Send and receive

- Send
  – send a message to a socket bound to a process
  – can be blocking or non-blocking
- Receive
  – receive a message on a socket
  – can be blocking or non-blocking
- Broadcast/multicast
  – send to all processes/all processes in a group

Receive

- Blocked receive
  - destination process is blocked until message arrival
  - most commonly used
- Variations
  - conditional receive (continue until receiving indication that message arrived or polling)
  - timeout
  - selective receive (wait for message from one of a number of ports)

Asynchronous Send

- Characteristics:
  – send is non-blocking (process continues after the message is sent out)
  – buffering needed (at receive end)
  – mostly used with blocking receive
  – usable for multicast

- Problems
  – buffer overflow
  – error reporting (difficult to match error with message)

- Maps closely onto connectionless service.

Synchronous Send

- Characteristics:
  - send is blocking (sender suspended until message received)
  - synchronisation point for both sender & receiver
  - easier to reason about
- Problems
  - failure and indefinite delay causes indefinite blocking (use timeout)
  - multicasting/broadcasting not supported
  - implementation is more complex

Maps closely onto connection-oriented service.

Location transparency in send/receive

- Clients refer to services by name: a name server (binder) translates names into server locations at run-time.

Java API for Internet addresses

- • Class *InetAddress*

  – uses DNS (Domain Name System)

  InetAddress aC = InetAddress.getByName("gromit.cs.bham.ac.uk");

  throws *UnknownHostException*
- encapsulates detail of IP address (4 bytes for IPv4 and16 bytes for IPv6)

Java API for Datagram Comms

- Simple send/receive, with messages possibly lost/out of order
- Class *DatagramPacket*

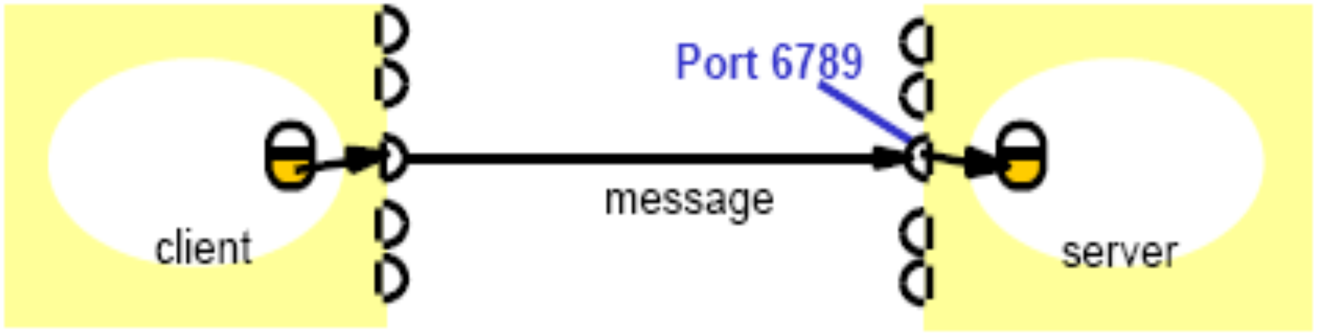
message (=array of bytes)	message length	Internet addr	port no

  – packets may be transmitted between sockets

  – packets truncated if too long

  – provides *getData*, *getPort*, *getAddress*

- UDP Client
  - sends a message and gets a reply
- UDP Server
  - repeatedly receives a request and sends it back to the client



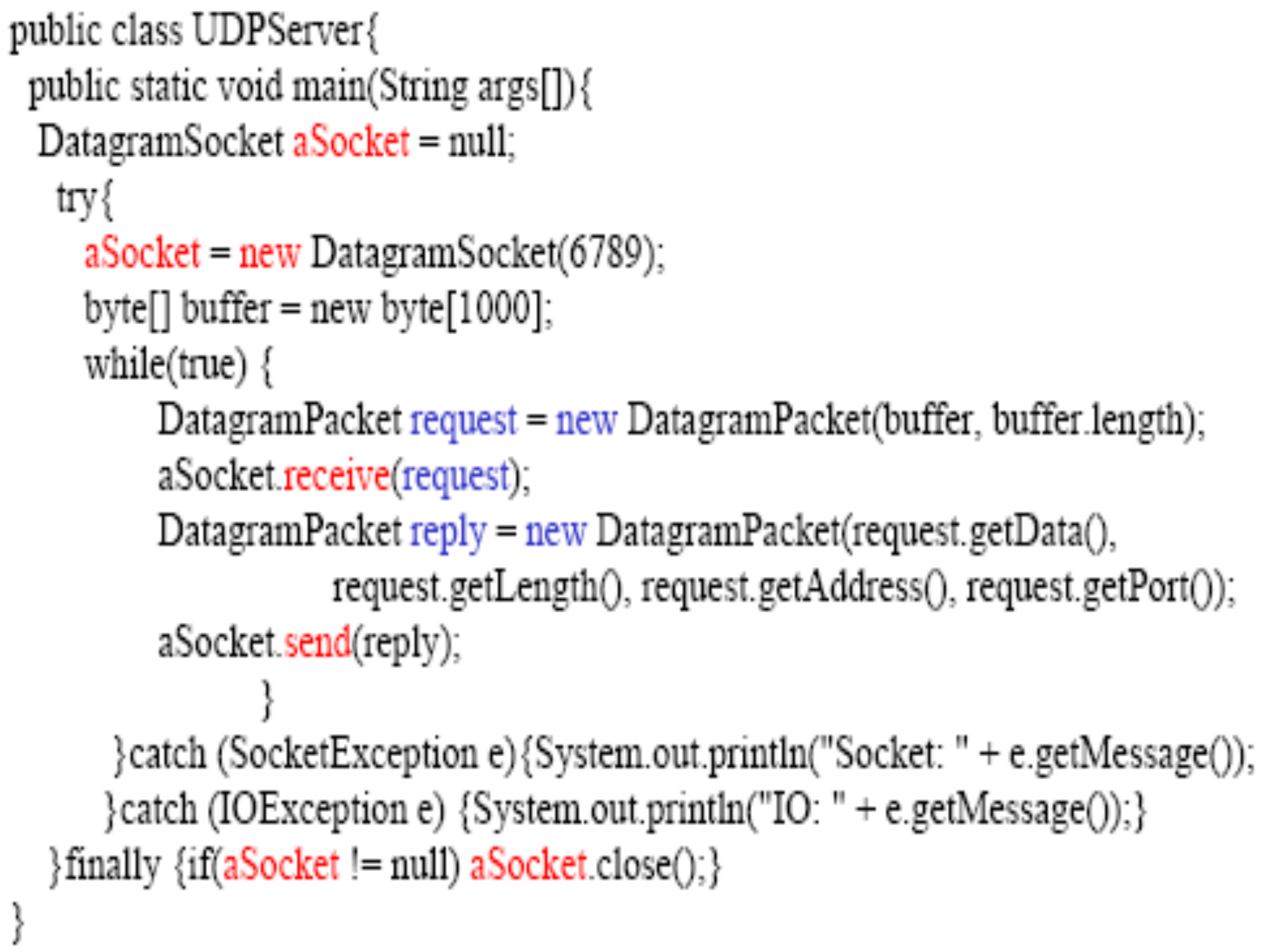
Java API for Datagram Comms

- Class *DatagramSocket*
  – *socket constructor* (returns free port if no arg)
  – *send a DatagramPacket*, non-blocking
  – *receive DatagramPacket*, blocking
  – *setSoTimeout* (receive blocks for time T and throws *InterruptedIOException*)
  – *close* DatagramSocket
  – throws *SocketException* if port unknown or in use
  See textbook site cdk3.net/ipc for complete code.

UDP client example

```
public class UDPClient{
public static void main(String args[]){
// args give message contents and server 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);
   }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 example



```
public class UDPServer{
  public static void main(String args[]){
   DatagramSocket aSocket = null;
     try{
       aSocket = new DatagramSocket(6789);
       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();}
}
```

Java API for Data Stream Comms

- Data stream abstraction
  - attempts to match the data between sender/receiver
  - marshaling/unmarshaling
- Class *Socket*
  - used by processes with a connection
  - *connect,* request sent from client
  - *accept,* issued from server; waits for a connect request,blocked if none availabl

Java API for Data Stream Comms

Class *ServerSocket*

– socket constructor (for listening at a server port)

– *getInputStream, getOutputStream*

– *DataInputStream, DataOutputStream* (automatic marshaling/unmarshaling)

– *close* to close a socket

– raises *UnknownHost*, *IOException*, etc

- In the next example... TCP Client

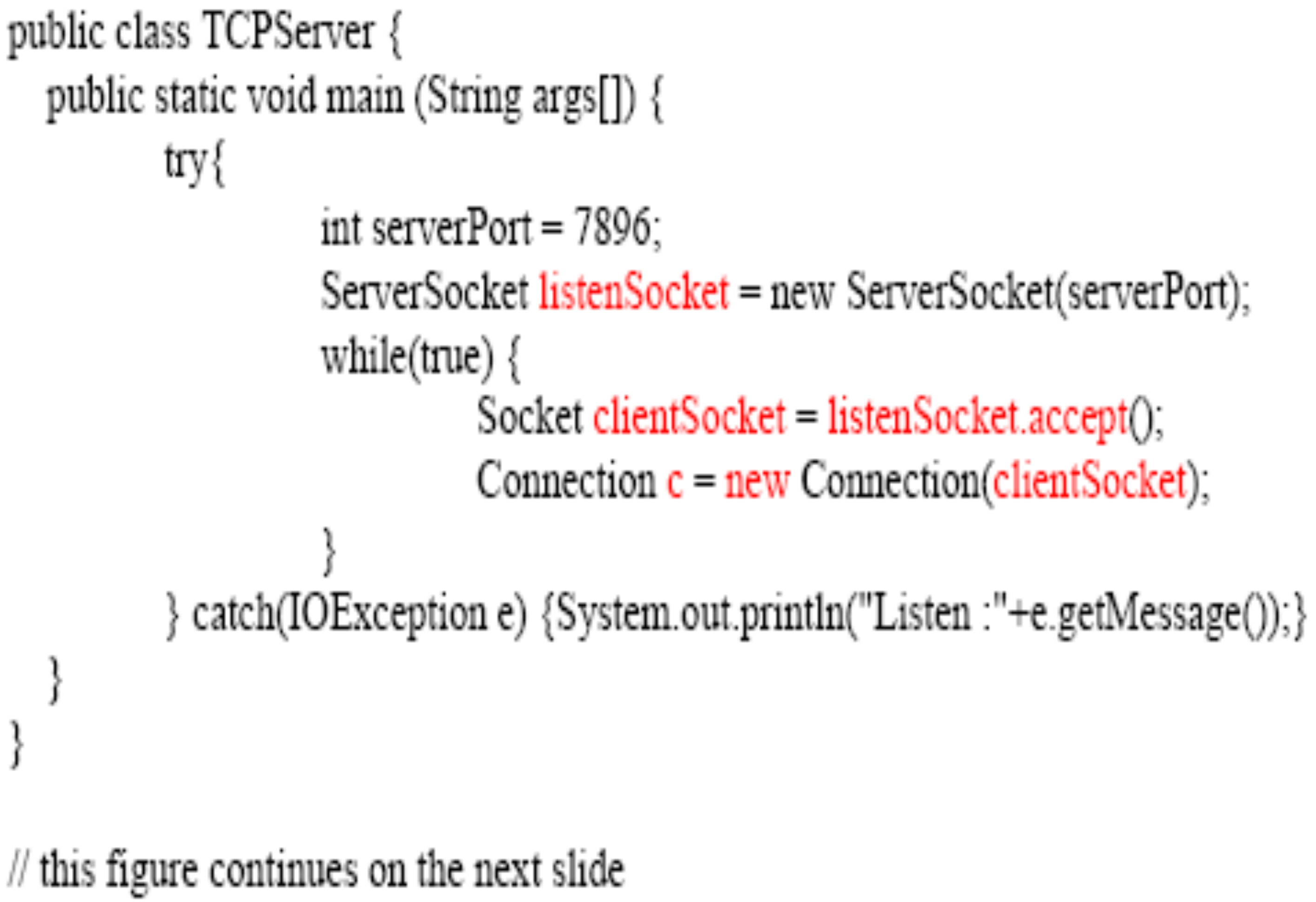
  makes connection, sends a request and receives a reply
- TCP Server

  makes a connection for each client and then echoes the client’s request

TCP client example

```
public class TCPClient {
            public static void main (String args[]) {
              // arguments supply message and hostname of destination
              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 Sec 4.3
                       String data = in.readUTF();
                       System.out.println("Received: "+ data) ;
                       s.close();
               }catch (UnknownHostException e){
                                   System.out.println("Sock:"+e.getMessage());
               }catch (EOFException e){System.out.println("EOF:"+e.getMessage());
               }catch (IOException e){System.out.println("IO:"+e.getMessage());}
            }finally {if(s!=null} try {s.close();}catch (IOException e)....}
```

TCP server example



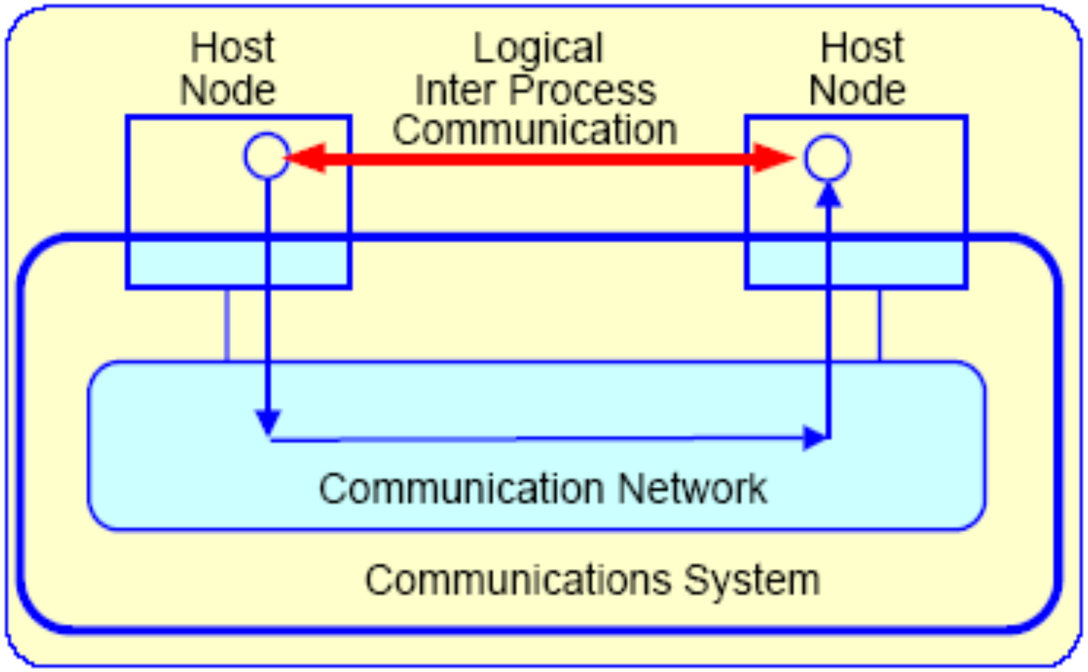
```
public class TCPServer {
   public static void main (String args[]) {
           try{
                   int serverPort = 7896;
                   ServerSocket listenSocket = new ServerSocket(serverPort);
                   while(true) {
                           Socket clientSocket = listenSocket.accept();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

// this figure continues on the next slide
```

TCP server example ctd

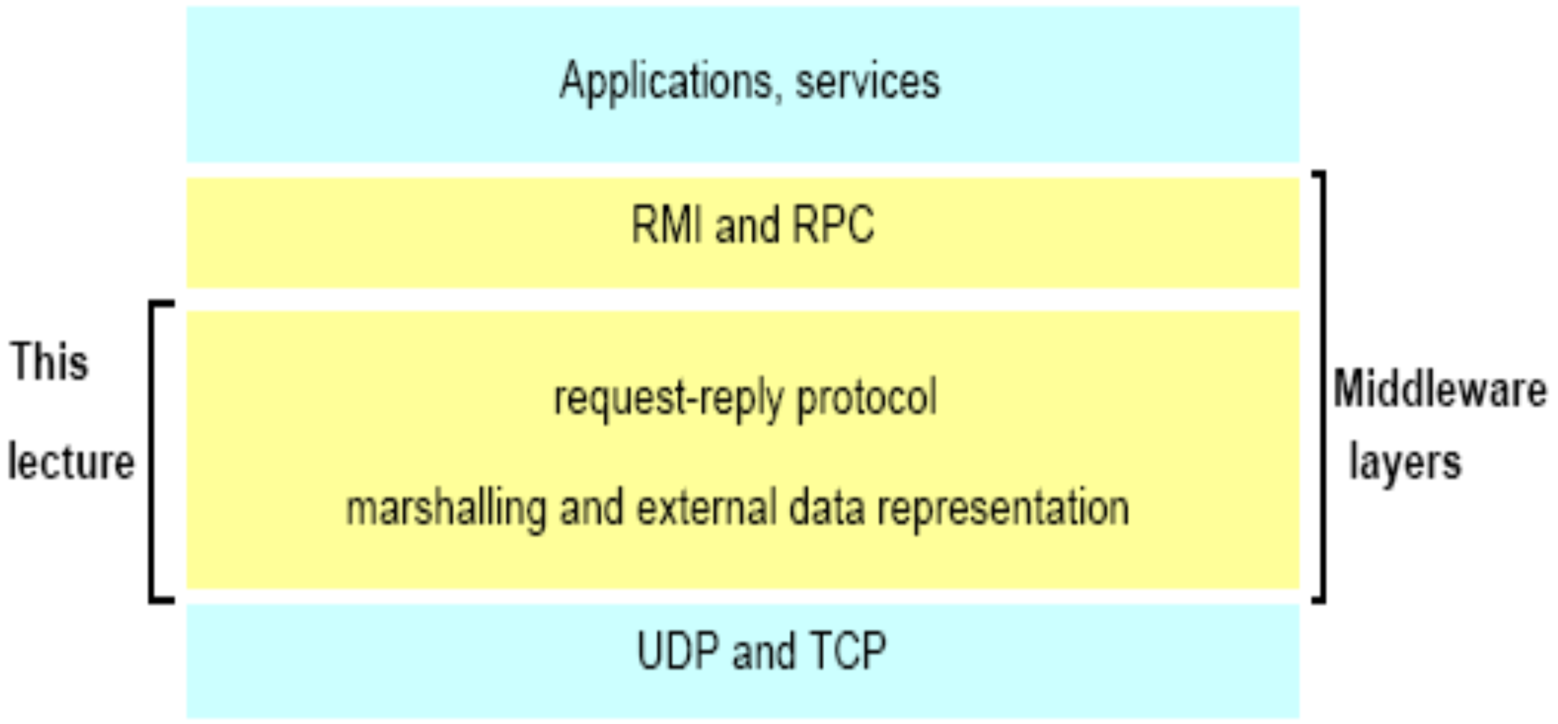
```
class Connection extends Thread {
        DataInputStream in;
        DataOutputStream out;
        Socket clientSocket;
        public Connection (Socket aClientSocket) {
          try {
                clientSocket = aClientSocket;
                in = new DataInputStream( clientSocket.getInputStream());
                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();
                out.writeUTF(data);
          } catch(EOFException e) {System.out.println("EOF:"+e.getMessage());
          } catch(IOException e) {System.out.println("IO:"+e.getMessage());}
        } finally {try {clientSocket.close();}catch (IOException e)…..}
}
```

Inter-process communication



Possibly several processes on each host (use ports).
Send and receive primitives.

API for Internet programming...



Client-Server Interaction

- Request-reply: it supports the roles and message exchanges in client-server interaction

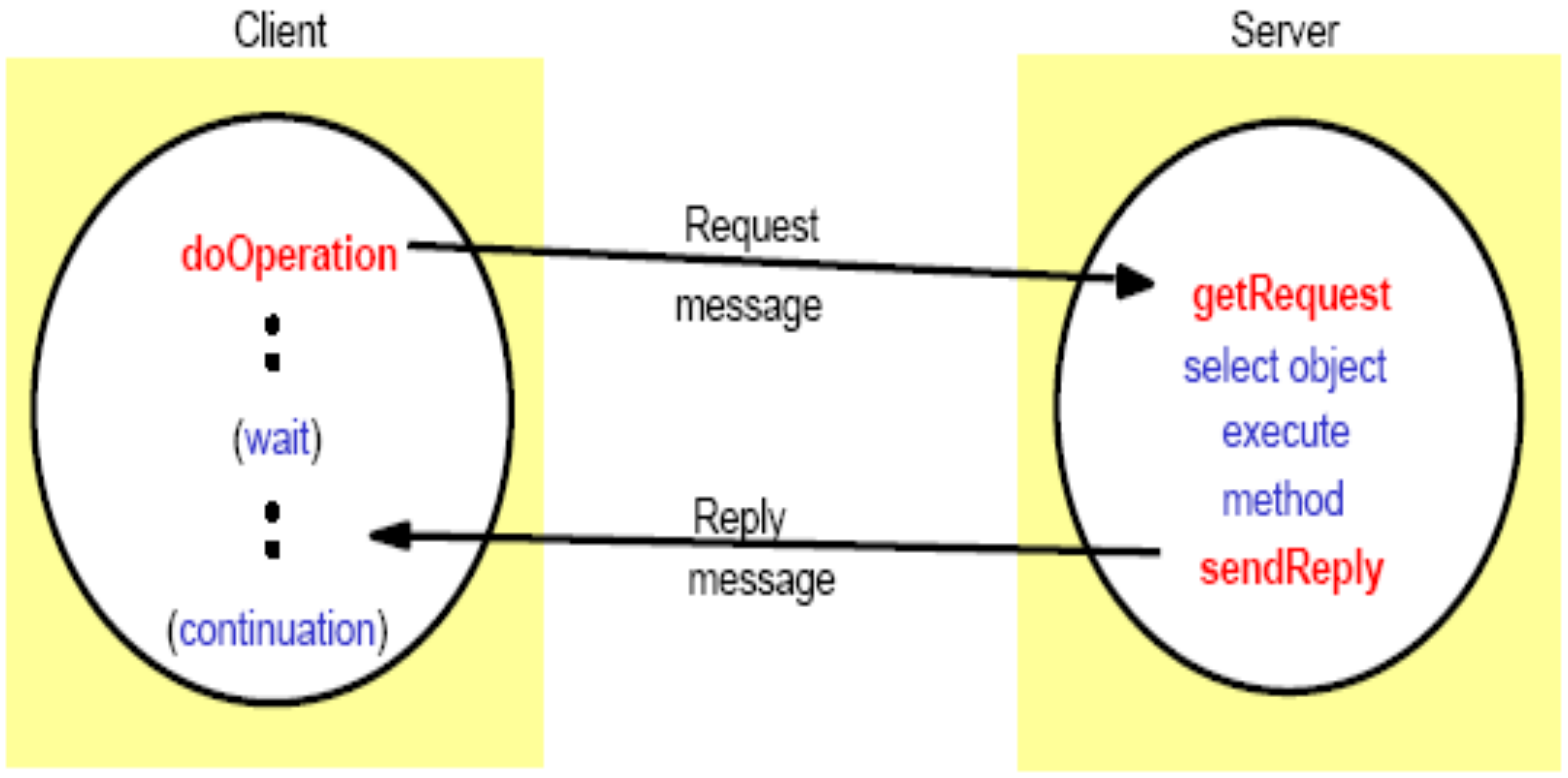
  – port must be known to client processes (usually published on a server)

  – client has a private port to receive replies

Client-server communications

- Notes:
  - ACK are redundant ....
  - Flow control may be redundant ...
  - send/receive can be implemented either using UDP or TCP

Request-Reply Communication



Operations of Request-Reply

- `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.

Message structure

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

Remote Object Reference

An identifier for an object that is valid throughout the distributed system

- must be unique
- may be passed as argument, hence need external representation

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

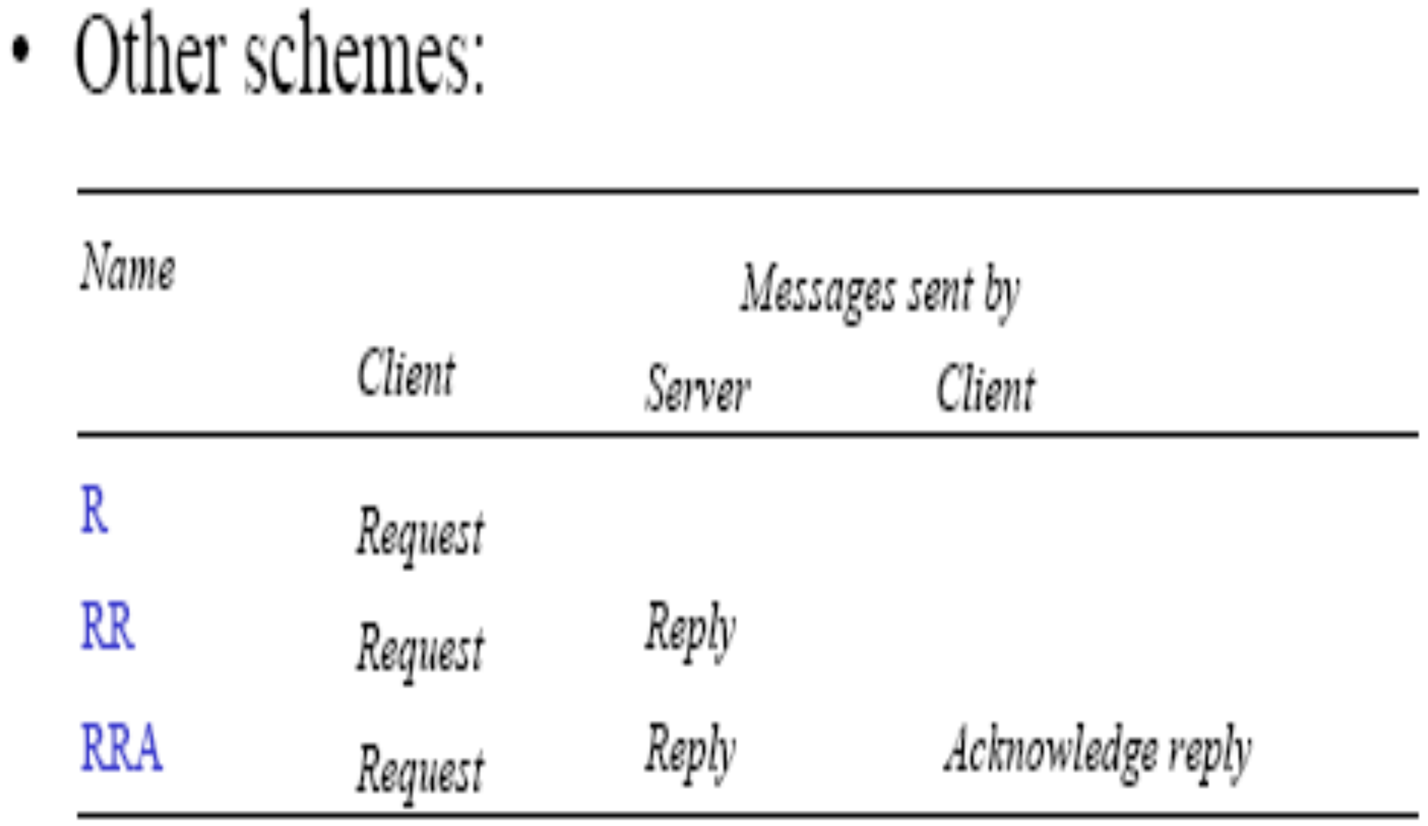
Failure model of the R-R protocol

- Omission failure
- Crash failure
- Order not guaranteed

- Timeouts
- Discarding duplicate requests

- Lost reply ... Repetition
- Idempotent operation = it has the same effects in spite of the number of time it is executed.
- Trace / History

Exchange protocols



- Other schemes:

Name	*Messages sent by*		
	Client	*Server*	*Client*
R	*Request*		
RR	*Request*	*Reply*	
RRA	*Request*	*Reply*	*Acknowledge reply*

Marshalling

- Program oriented data representation (data structures) message oriented data representation (sequences of bytes)
- Communication requires data conversion
- Conversion methods:
  - Sender’s format
  - Agreed external format (external data repr.)
  - Needed in RMI and RPC

Data Marshaling/Unmarshaling

- Marshalling = conversion of data into machine independent format, suitable for transmission
  - necessary due to heterogeneity & varying formats of internal data representation
- Unmarshalling = the inverse process ...

- Approaches
  - CORBA CDR (Common Data Representation)
  - Java object serialisation, cf DataInputStream,

CORBA Common Data Representation (CDR)

- External data representation defined in CORBA 2.0
- Represent all the data type in arguments and return values in remote invocation.
- Short, long, unsigned short, unsigned long, float, double, char, boolean, octet any + composit types
- n.b. No pointers .....

CORBA Common Data Representation (CDR)

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

- struct Person {

  ```
          string name;
          string place;
          long year;
  }
  ```

index in sequence of bytes	← 4 bytes →	*notes on representation*
0–3	`5`	*length of string*
4–7	`"Smith"`	*‘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}

- The type of the data items are not given
- Sender and recipient know types and order
- Corba IDL is used to describe data ...
- Corba interface compiler generate (un)marshalling operations

(Java) Object serialization

```
public class Person implements Serializable {
    private String name;
    private String place;
    private int year;
    public Person(String aName, String aPlace, int aYear) {
        name = aName;
        place = aPlace;
        year = aYear;
}

}
```

- Information about the class of each object is included in the serialized form.
- “objects in objects” are serialized too ...
- Look at class ObjectOutputStream, ObjectInputStream ...

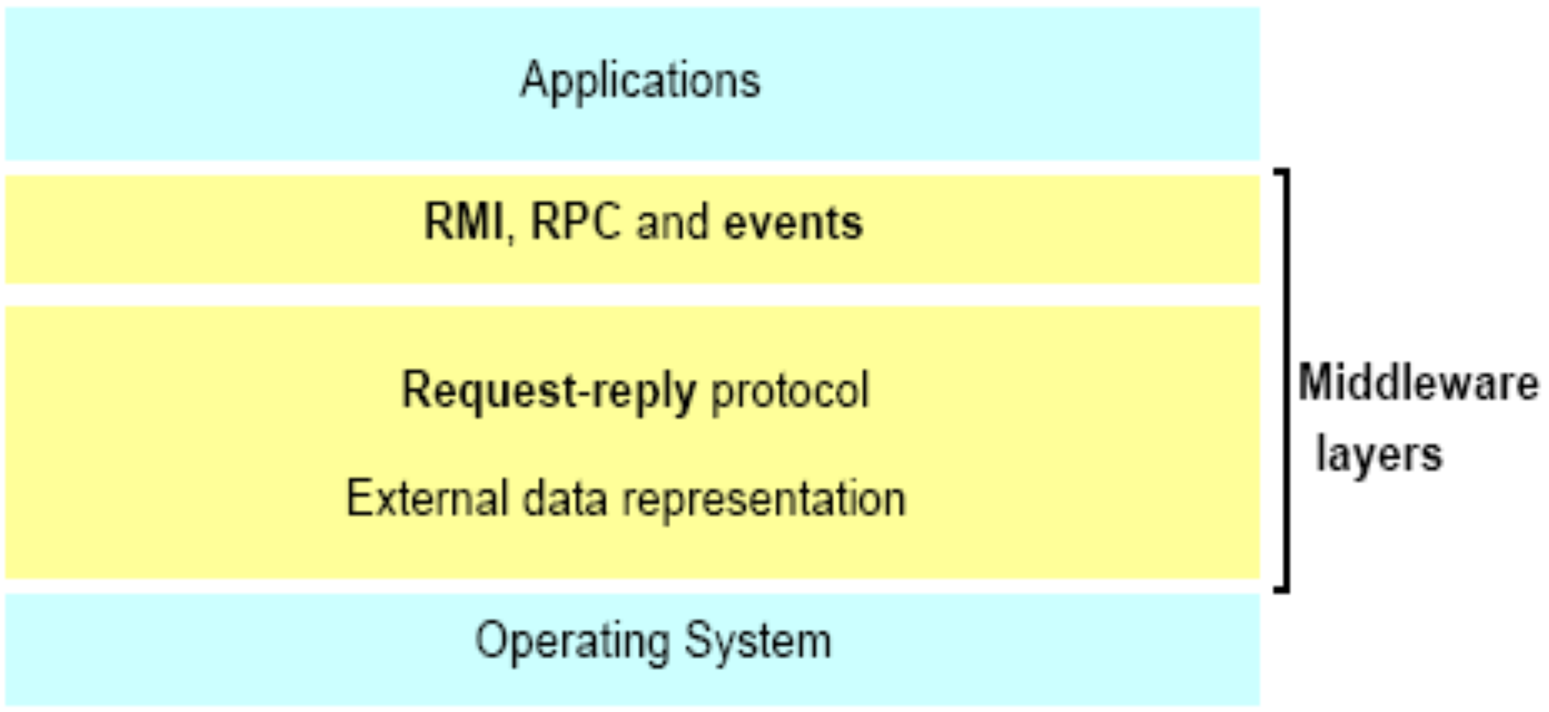
Serialized values				*Explanation*
Person	8-byte version number		h0	*class name, version number*
3	int year	java.lang.String name:	java.lang.String place:	*number, type and name of instance variables*
1934	5 Smith	6 London	h1	*values of instance variables*

The true serialized form contains additional type markers; h0 and h1 are handles

Reflection

- Ability to enquire about the properties of a class (names and types)

…layers…


Applications
RMI, RPC and events
Request-reply protocol
External data representation
Operating System
Middleware
layers

Distributed applications programming

- Object Interaction: RMI (RPC)
  - distributed objects model
  - RMI … invocation semantics
  - RPC
  - events and notifications

- Products
  - Java RMI, CORBA, DCOM
  - Sun RPC
  - Jini

...RMI and RPC…

…stand on top of OS, independent of:

– computer hardware… external data representation
– operating system… socket abstraction
– communication protocols… abstract request-reply protocols over UDP or TCP
– different programming languages, e.g. CORBA supports Java, C++…

…moreover…

Location transparency

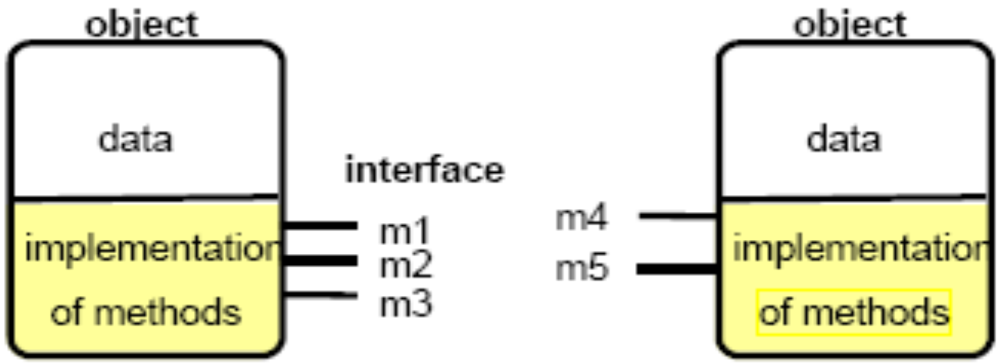
- client/server do not need to know their location

The object model

- Programs are logically partitioned into objects
  - Interfaces
  - signature definition

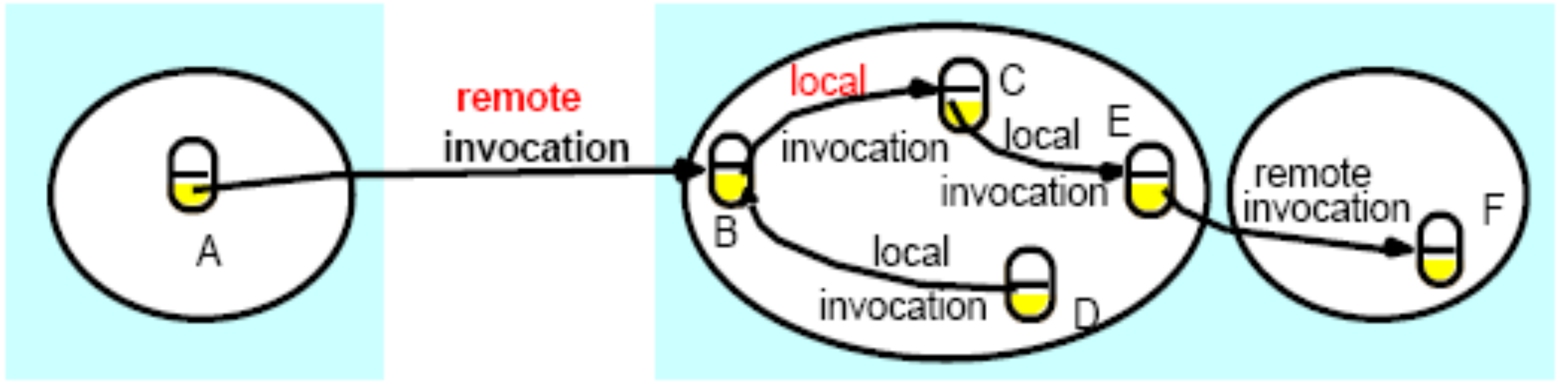
- Actions
  – via method invocation
  – effects ...
  – may lead to exceptions

- Garbage collection



- Objects = data + methods
  - logical and physical nearness
  - first class citizens, can be passed as arguments
- Interact via interfaces:
  - define types of arguments and exceptions of methods

The distributed object model



- Objects distributed (client-server models)
- Extend with
  - Remote object reference
  - Remote interfaces
  - Remote Method Invocation (RMI)

Advantages of distributed objects

- Objects
  – can act as clients, servers, etc
  – can be replicated for fault-tolerance and performance
  – can migrate,

- Data encapsulation gives better protection
  – concurrent processes, interference

Method invocations can be remote or local

Interfaces ...

- ... in distributed systems
  - Modules run in separate processes and different machines
  - No access to variables
  - No call-by-value/call-by-reference
  - No pointers

- Service interface
- Remote interface

Interface Definition Languages (IDL)

- They provide a notation that allow objects implemented in a variety of languages to invoke one another

The Distributed Object Model

- Remote object reference
- Remote interface

Remote object reference

- Object references
  – used to access objects which live in processes
  – can be passed as arguments, stored in variables,...

- Remote object references
  – object identifiers in a distributed system
  – must be unique in space and time
  – error returned if accessing a deleted object
  – can allow relocation (see CORBA)

Remote object reference

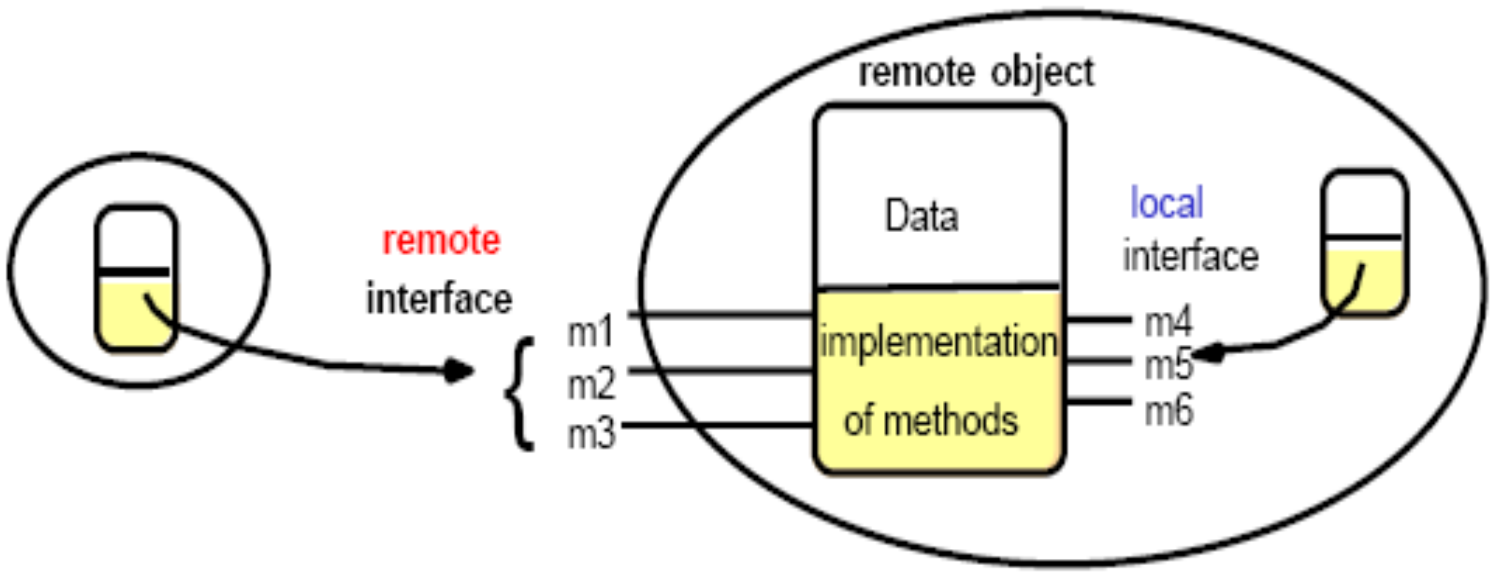
- Constructing unique remote object reference
  - IP address, port, interface name
  - time of creation, local object number (new for each object)
- Use the same as for local object references
- If used as addresses
  - cannot support relocation (alternative in CORBA)

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

Remote interfaces

- Specify externally accessed
  - – variables and procedures
  - – no direct references to variables (no global memory)
  - – local interface separate
- Parameters
  - – input, output or both,
  - – instead of call by value, call by reference
- No pointers
- No constructors

Remote object and its interfaces



- CORBA: Interface Definition Language (IDL)
- Java RMI: as other interfaces, keyword *Remote*

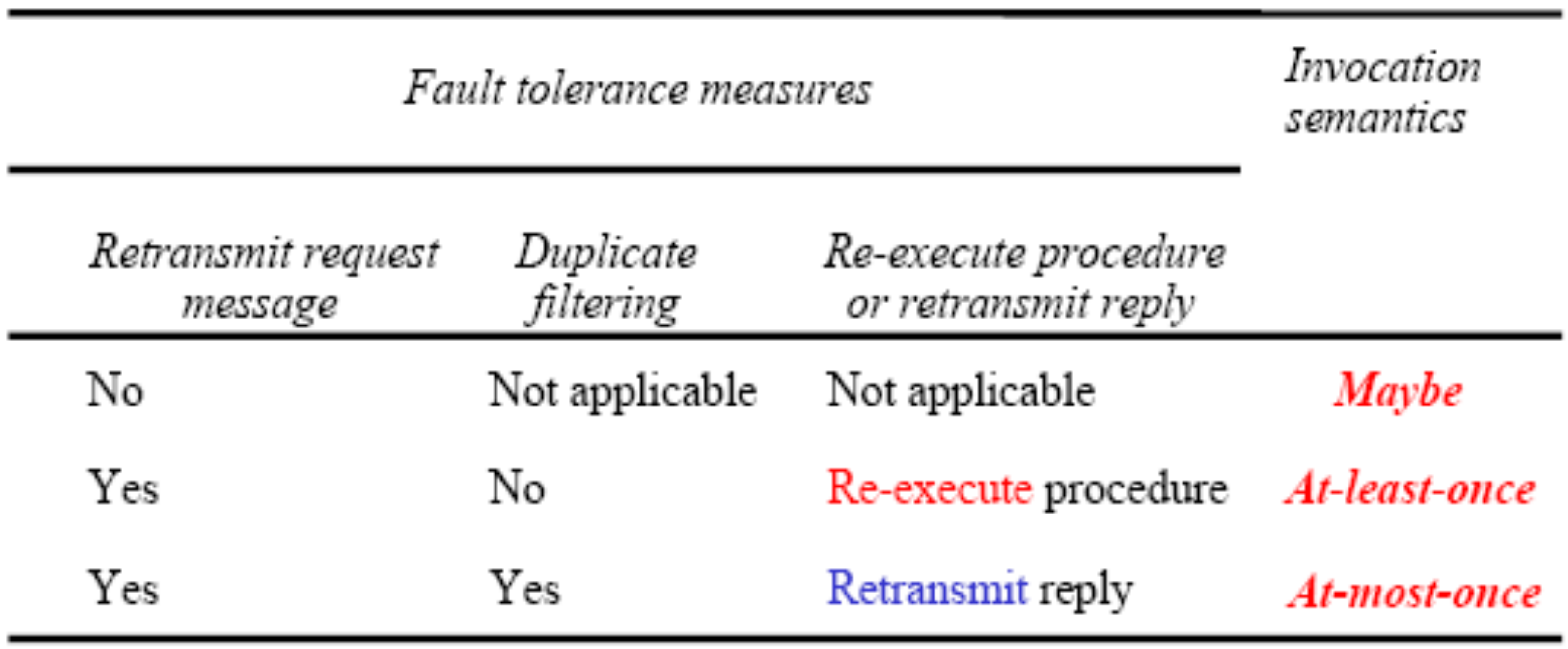
Handling remote objects

- Exceptions
  – raised in remote invocation
  – clients need to handle exceptions
  – timeouts in case server crashed or it is too busy
- Garbage collection
  – distributed garbage collection may be necessary
  – combined local and distributed collector
  – cfr Java reference counting

RMI design issues

- Local invocations
  - – executed exactly once
- Remote invocations
  - – via Request-Reply (see *DoOperation*)
  - – may suffer from communication failures!
- retransmission of request/reply
- message duplication, duplication filtering
  - – no unique semantics…

Invocation semantics summary



Fault tolerance measures			*Invocation semantics*
Retransmit request message	*Duplicate filtering*	*Re-execute procedure or retransmit reply*	
No	Not applicable	Not applicable	***Maybe***
Yes	No	Re-execute procedure	***At-least-once***
Yes	Yes	Retransmit reply	***At-most-once***

Re-executing a method sometimes dangerous...

Maybe invocation

- the remote method….
  - – may (or may not) been executed
- the invocation message was lost...
  - – method not executed
- the result was not received...
  - – was the method executed or was not?
- the server crashed
  - – before or after method executed?
  - – if timeout, result could be received after timeout...

At-least-once invocation

- the invoker receives a result or an exception

  – retransmission of request messages
- Invocation message retransmitted...

  – method may be executed more than once

  – arbitrary failure (wrong result)

  – method must be idempotent (repeated execution has the same effect as a single execution)
- Server crash...

  – dealt with by timeouts, exceptions

At-most-once invocation

- the invoker receives result (executed once) or exception (no result)
  – retransmission of reply & request messages
  – duplicate filtering
- Best fault-tolerance...
  – arbitrary failures can be prevented if methods are called at most once
- Used by CORBA and Java RMI

Transparency of RMI

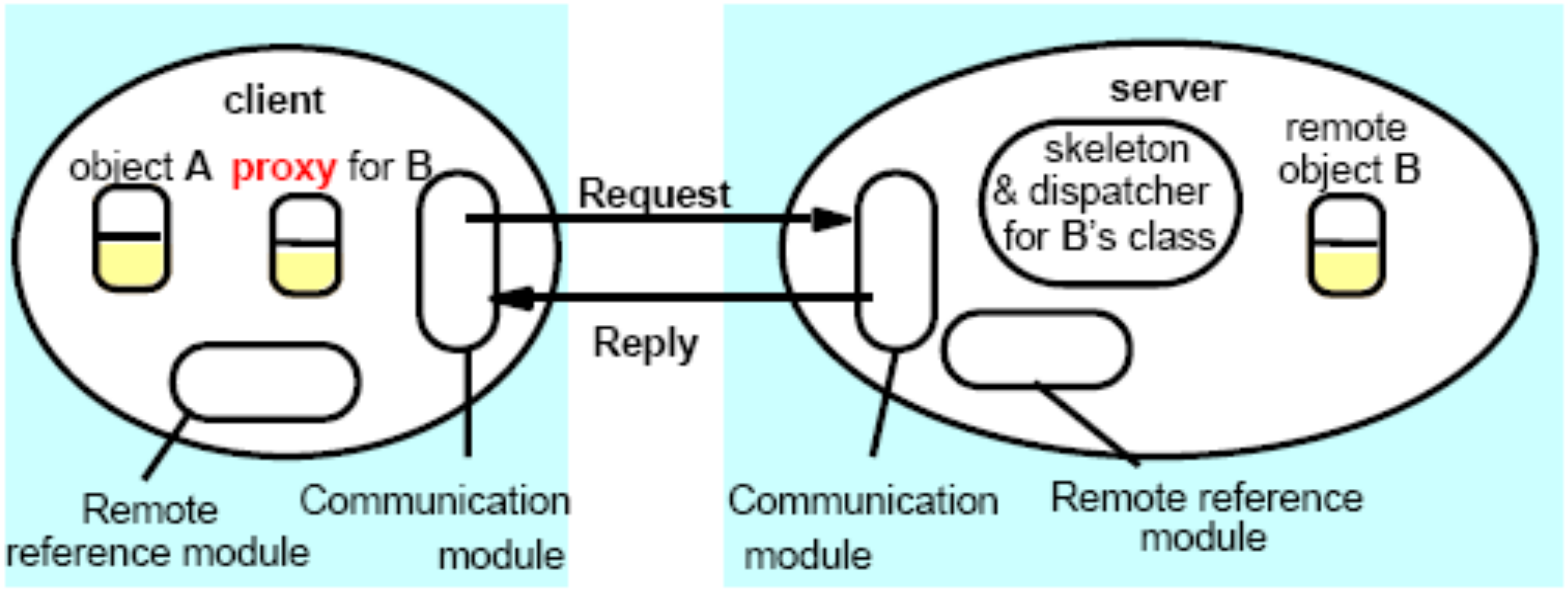
- Should remote method invocation look the same as local?
  - same syntax, see Java RMI (keyword *Remote*)
  - need to hide
- data marshalling
- IPC calls
- locating/contacting remote objects

Problems

  – different RMI semantics? susceptibility to failures?
  – protection against interference in concurrent scenario?

- Approaches (Java RMI)
  – transparent, but express differences in interfaces
  – provide recovery features

Implementation of RMI



Object A invokes a method in a remote object B: communication module, remote reference module, RMI software.

Communication modules

- Reside in client and server
- Carry out Request-Reply jointly
  – use unique message ids (new integer for each message)
  – implement a given RMI semantic
- Server's communication module
  – selects dispatcher within RMI software
  – converts remote object reference to local

Remote reference module

- Creates remote object references and proxies
- Translates remote to local references (object table):
  – correspondence between remote and local object references (proxies)
- Directs requests to proxy (if exists)
- Called by RMI software
  – when marshalling/unmarshalling

The core of middleware

- Proxy
  - behaves like a local object to client
  - forwards requests to the remote object
  - it hides:

    remote object reference

    argument marshalling

    send/receive

- How many proxies in a client ?
- Remote interface implementation

RMI software architecture

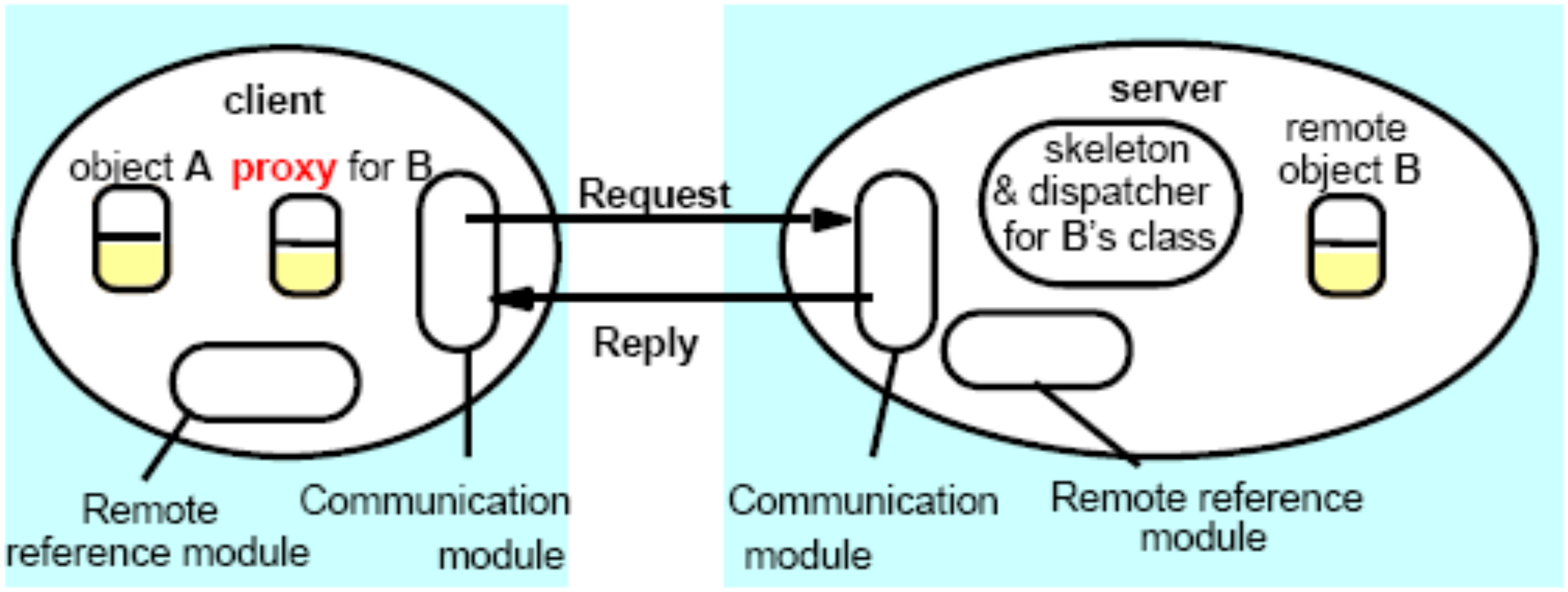
- Dispatcher
  - – receives request
  - – selects method and passes on request to skeleton
- Skeleton
  - – implements methods in remote interface
  - – unmarshals data, invokes remote object
  - – waits for result, marshals it and returns reply

- Proxy, dispatcher, skeleton are implemented “automatically” ...

  (generated automatically)

- Interface compiler ...

Implementation of RMI



Object A invokes a method in a remote object B: communication module, remote reference module, RMI software.

A server ...

- Dispatcher + skeleton classes
- Its own (remote) object classes
- Initialiazion (the mytical main ...)

A client

- Proxies classes

- A factory method ...
  - For creation of a remote object ...
- A factory object

Binding

- The binder

  – mapping from textual names to remote references (using a table)

  – used by clients as a look-up service (cf Java RMIregistry)

Activation

- Activation

  – objects active (available for running) and passive (implementation of methods + marshalled state)

  – activation = create new instance of class + initialise from stored state
- Activator

  – records location of passive and active objects

  – starts server processes and activates objects within them

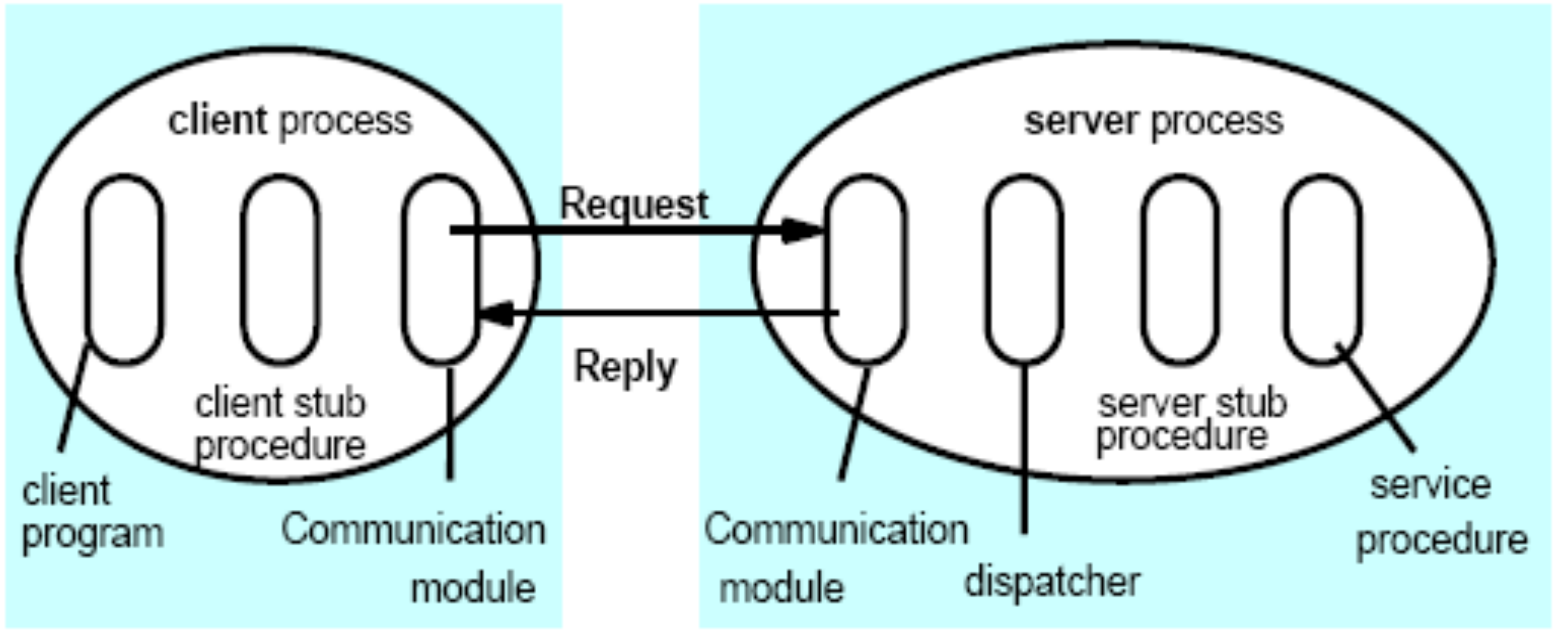
Persistent Objects

- ... Managed by a persisten object store that keeps their states on disk in marshalled form
- When an object is made “passive” ...

Location service

- It locates remote objects from their remote object reference
- It maps remote object references to probable locations
- It supports object migration

RPC client and server



Implemented over Request-Reply protocol.

Remote Procedure Call

- A client program calls a procedure in another program running in a server machine
- Available procedures are defined in server' s *service interface*
- Local procedures vs metods

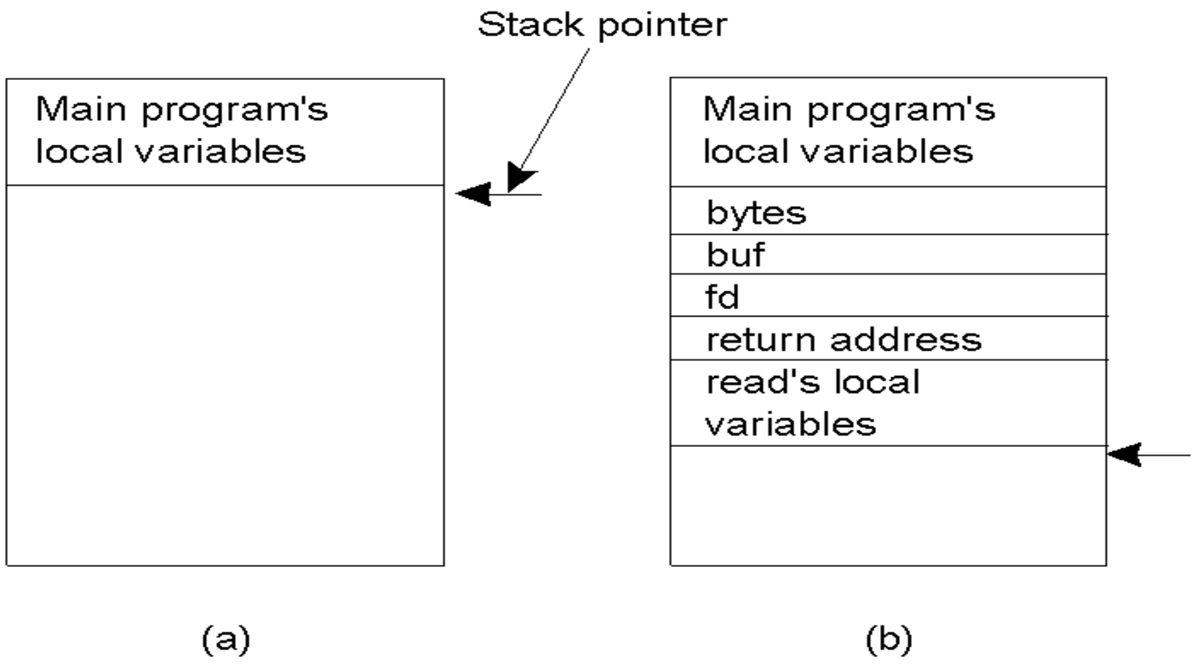
  RPC vs RMI

RPC

- RPC
  - historically first, now less used
  - implemented over the Request-Reply protocol
  - at-least-once or at-most-once semantics
  - can be seen as a restricted form of RMI

- RPC software architecture
  - similar to RMI (communication, dispatcher and stub in place of proxy/skeleton)

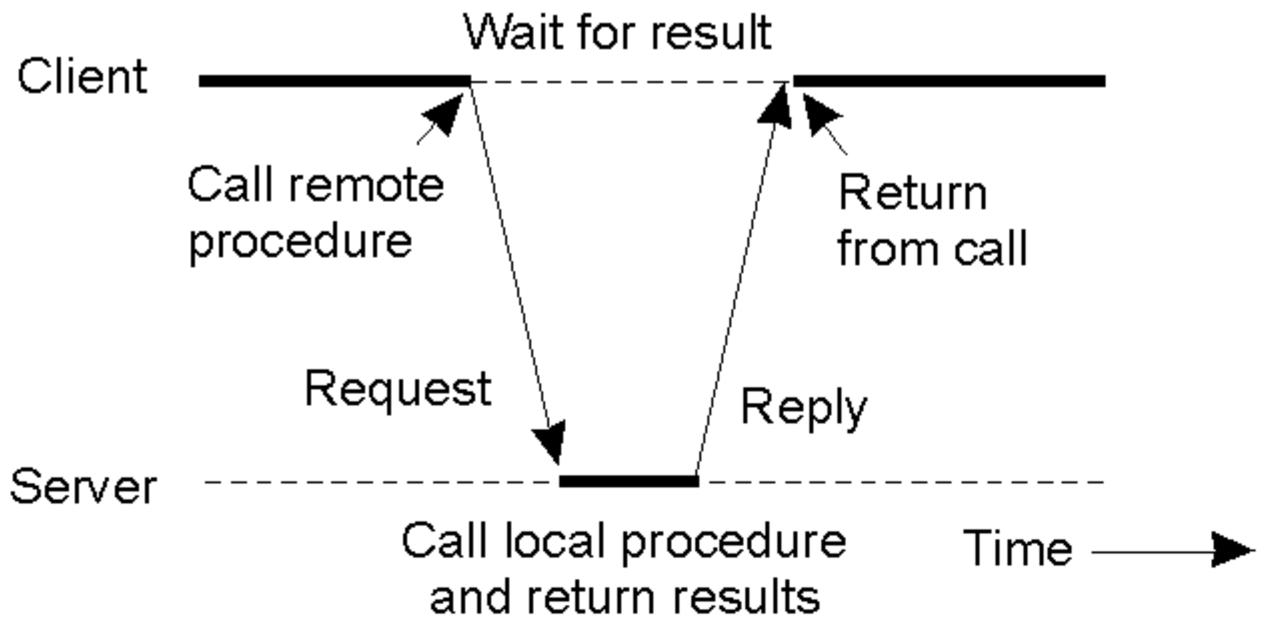
Conventional Procedure Call



a) Parameter passing in a local procedure call: the stack before the call to read

b) The stack while the called procedure is active

Client and Server Stubs

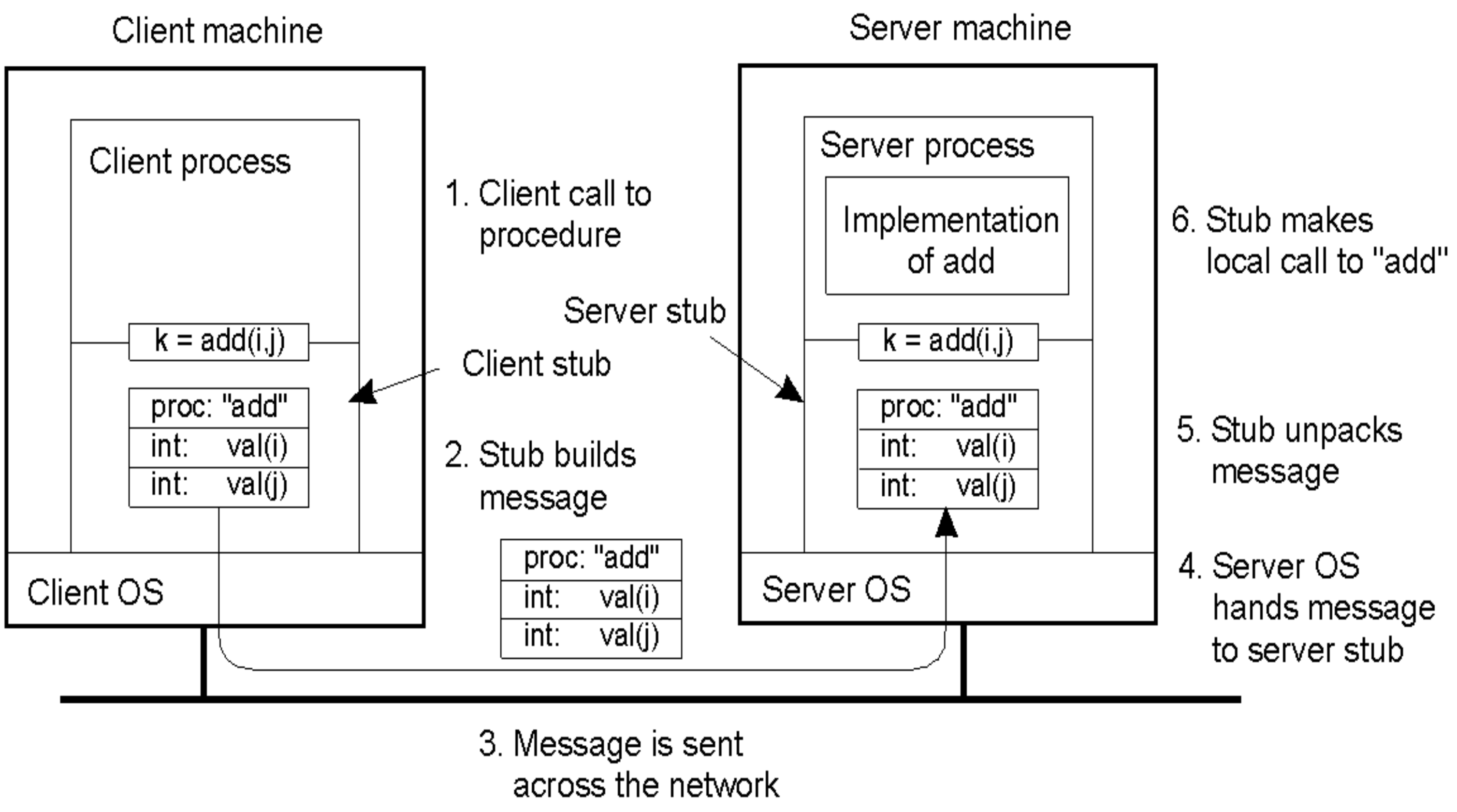


- Principle of RPC between a client and server program.

Steps of a Remote Procedure Call

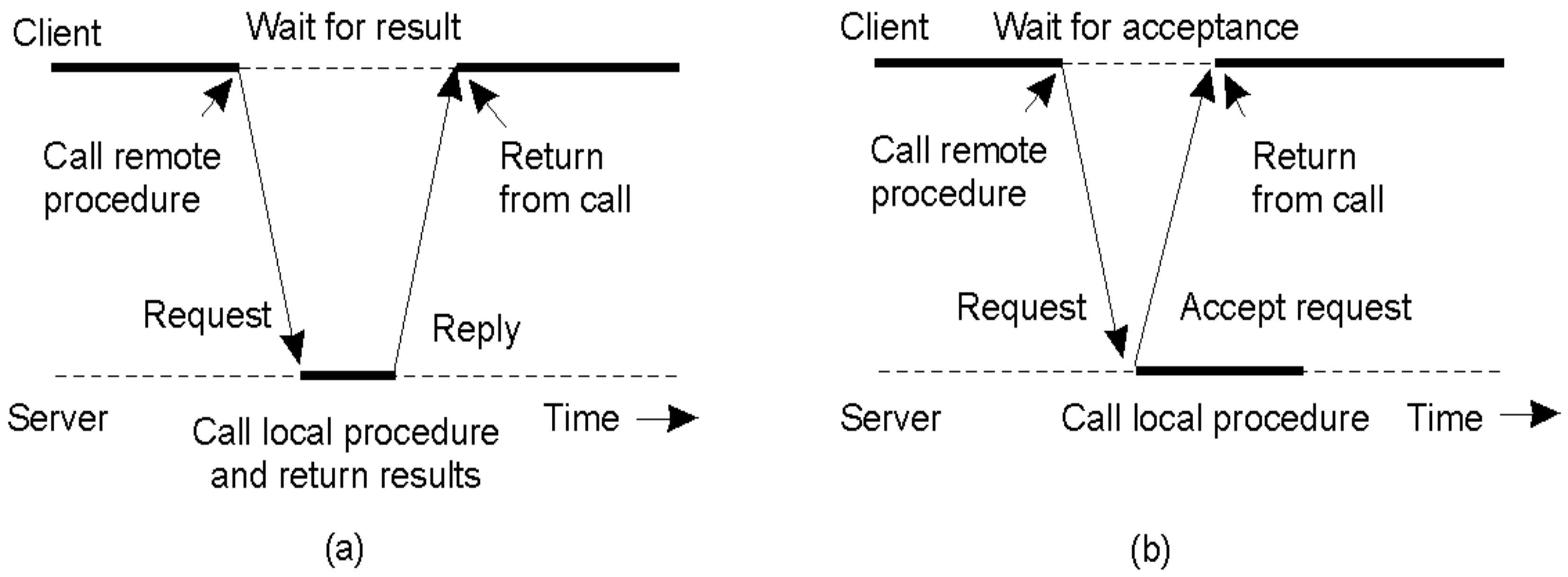
1. Client procedure calls client stub in normal way
2. Client stub builds message, calls local OS
3. Client's OS sends message to remote OS
4. Remote OS gives message to server stub
5. Server stub unpacks parameters, calls server
6. Server does work, returns result to the stub
7. Server stub packs it in message, calls local OS
8. Server's OS sends message to client's OS
9. Client's OS gives message to client stub
10. Stub unpacks result, returns to client

Passing Value Parameters (1)



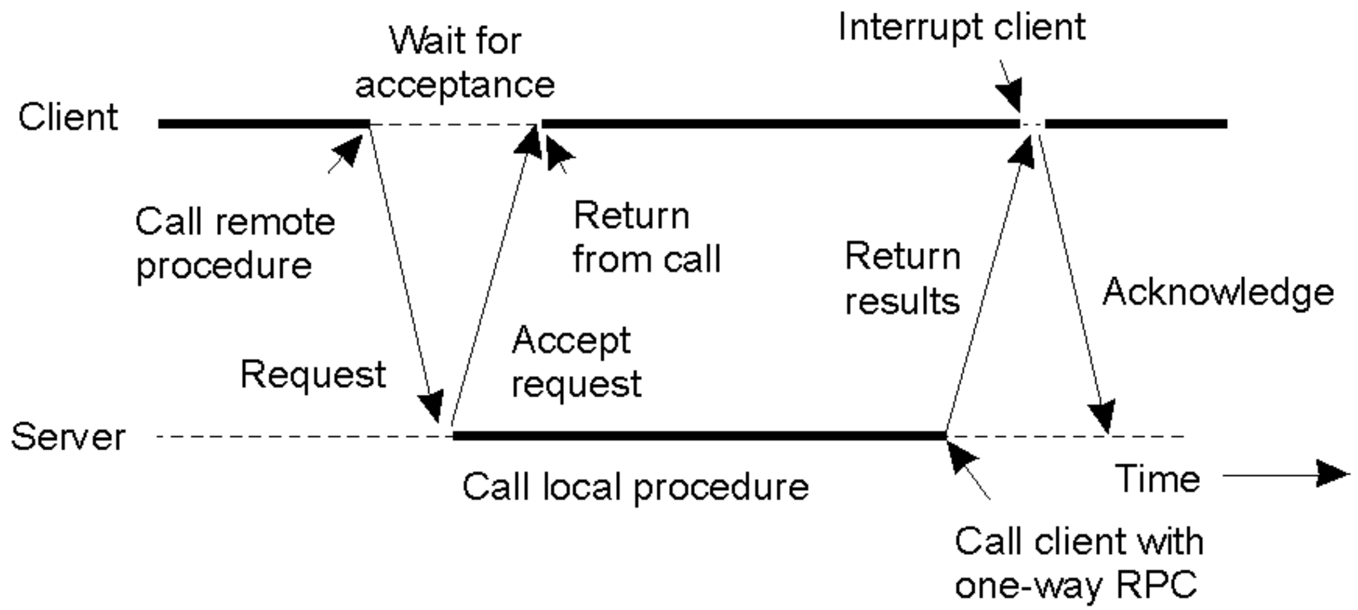
- Steps involved in doing remote computation through RPC

Asynchronous RPC (1)



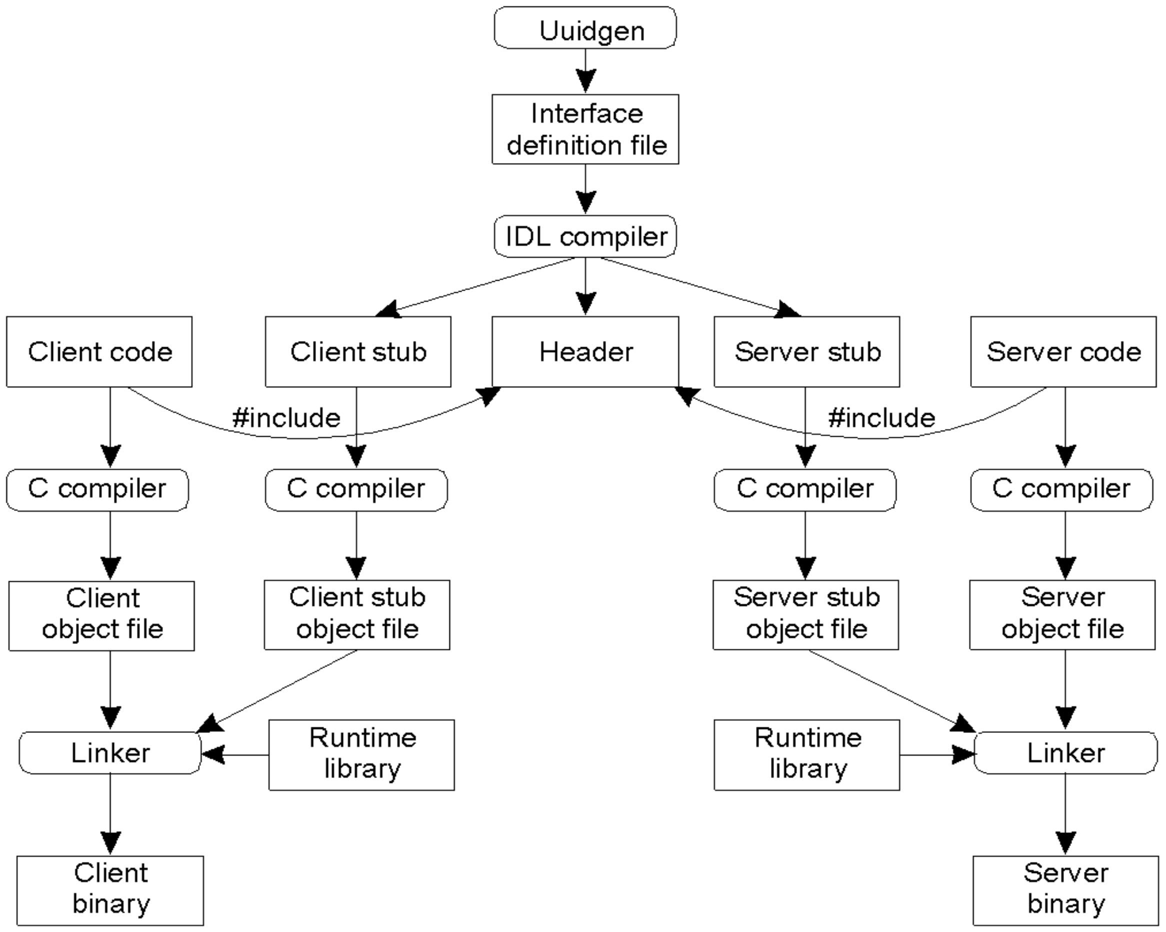
a) The interconnection between client and server in a traditional RPC
b) The interaction using asynchronous RPC

Asynchronous RPC (2)



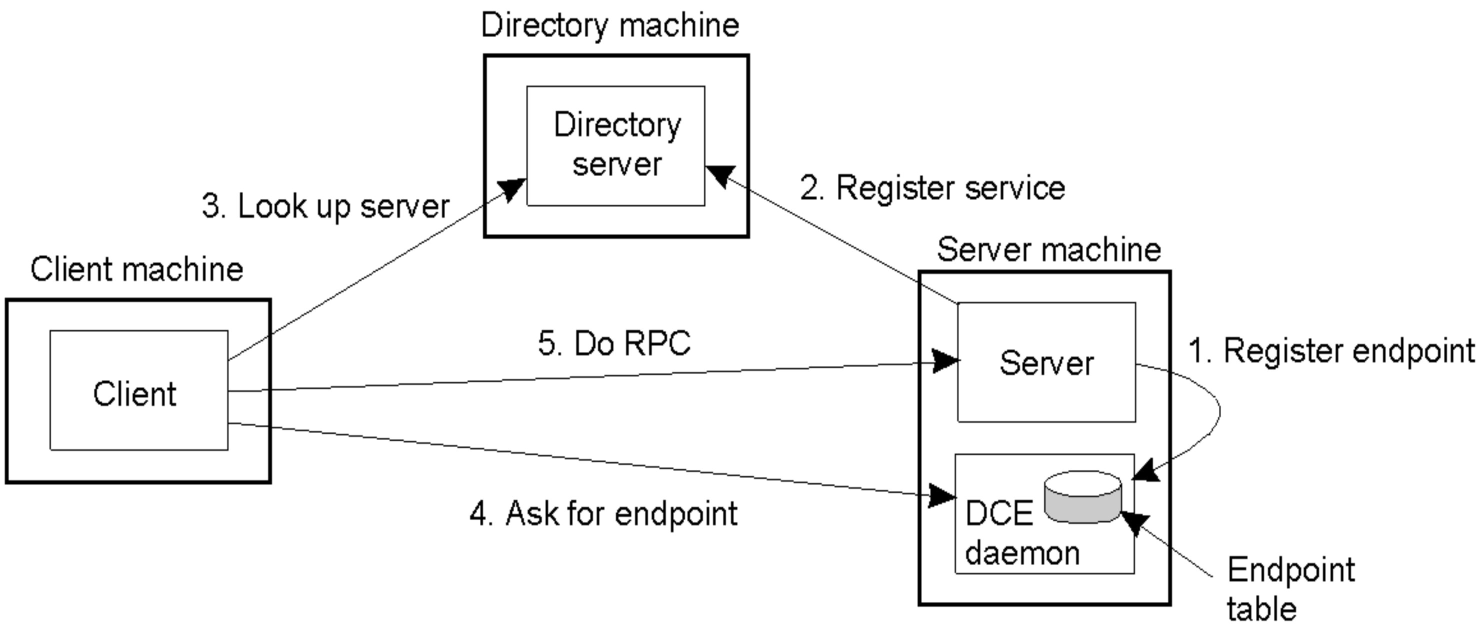
- A client and server interacting through two asynchronous RPCs

Writing a Client and a Server



- The steps in writing a client and a server in DCE RPC.

Binding a Client to a Server



- Client-to-server binding in DCE.

JAVA RMI

- Java RMI extends the Java Object Model
- Object making a remote invocation must handle a *RemoteException*
- The implementation of a remote object must implement the *Remote* interface

- Remote Interfaces in Java RMI
- Method parameters = input parameters
- Method result = single output parameter

- RMIregistry ... The Java RMI binder
  - The Naming class ...
  - Use a URL-formatted string like:
    //computerName:port/objectName

- The server side ...
  - Main
  - Servant classes
  - Servant classes as extension of *UnicastRemoteObject*

- The client side
  - It uses a binder to lookup a remote object reference ...

…Distributed Systems…

end of lectures