SRM UNIVERSITY
(Under section 3 of UGC Act 1956)

UNIT-III

REMOTE METHOD INVOCATION AND OBJECTS

1. Introduction

1. Request-reply protocols
2. RPC
3. RMI – in 1990s – RMI extension allowing a local object to invoke methods of remote objects

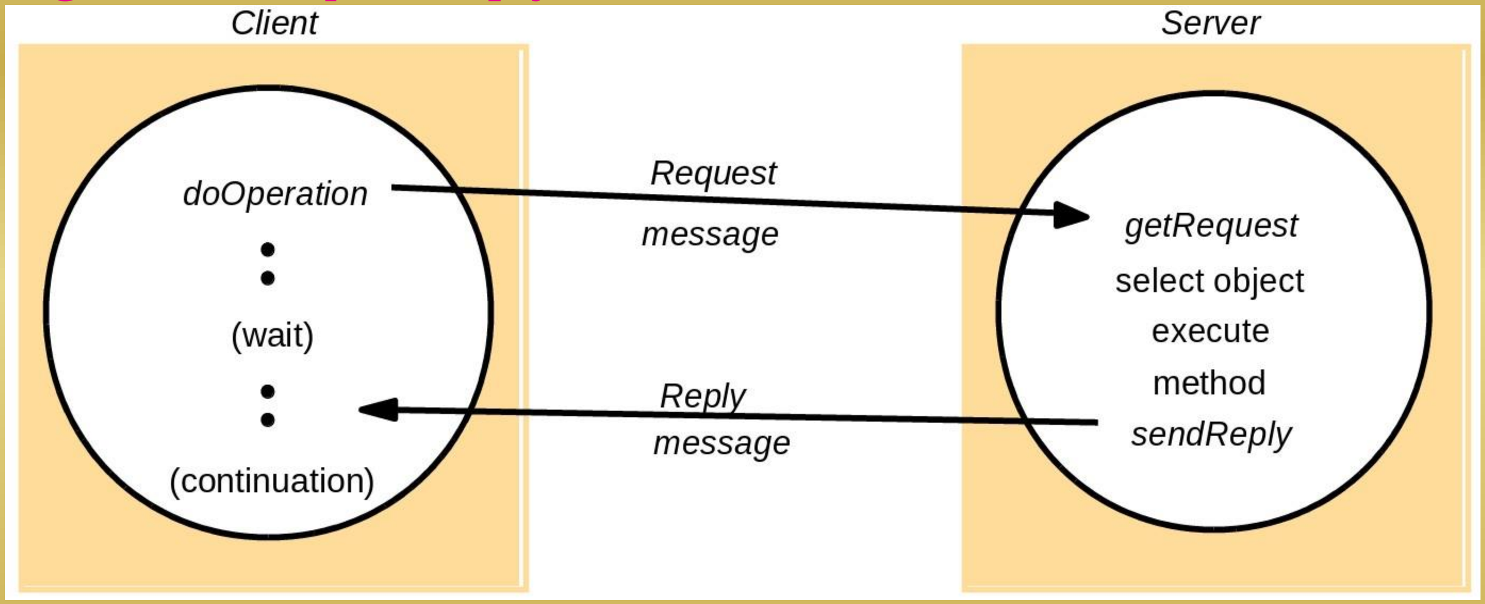
5.2 Request-reply protocols

- typical client-server interactions – request-reply communication is synchronous because the client process blocks until the reply arrives
- Asynchronous request-reply communication – an alternative that may be useful in situations where clients can afford to retrieve replies later

The request-reply protocol

doOperation, *getRequest* and *sendReply*

Figure 5.2 Request-reply



doOperation by clients to invoke remote op.; together with additional arguments; return a byte array. Marshaling and unmarshaling!

getRequest by server process to acquire service requests; followed by

sendReply send reply to the client

Figure 5.3 Operations of the request-reply protocol

public byte[] doOperation (RemoteRef s, int operationId, byte[] arguments)
sends a request message to the remote server and returns the reply.
The arguments specify the remote server, the operation to be invoked and the arguments of that operation.

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.

Figure 5.4 Request-reply message

messageType	*int (0=Request, 1= Reply)*
requestId	*int*
remoteReference	*RemoteRef*
operationId	*int or Operation*
arguments	*array of bytes*

Message identifiers

1. *requestID* – increasing sequence of integers by the sender 2.server process identifier – e.g. internet address and port

Failure model of the request-reply protocol

A. UDP datagrams

communication failures (omission failures; sender order not guaranteed) + possible crash failures

action taken when a timeout occurs depends upon the delivery guarantees being offered

Timeouts – scenarious for a client bahaviour

Discarding duplicate request messages – server filtering out duplicates

Lost reply messages

***idempotent* operation** – **an operation**

with the same effect as if it had been performed exactly once

History

retransmission by server ... problem with memory size ... ←– can be cured by the knowledge that the message has arrived, e.g.:

clients can make only one request at a time ⇒ server can interpret each request as an acknowledgement of its previous reply!

Styles of exchange protocols Three different types of protocols (Spector [1982]):

- the request (R) protocol
  - – No confirmation needed from server - client can continue right away – UDP implementation

•the request-reply (RR) protocol

– most client-server exchanges

•the request-reply-acknowledge reply (RRA)

protocol Figure 5.5 RPC exchange protocols

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

B. TCP streams to implement request-reply protocol

- TCP streams
  - transmission of arguments and results of any size
    - * flow-control mechanism
      - · ⇒ no need for special measures to avoid overwhelming the recipient
  - request and reply messages are delivered reliably
    - * ⇒ no need for
      - ·retransmission
      - ·filtering of duplicates
      - ·histories

Example: HTTP request-reply protocol

fixed set of methods (GET, PUT,POST, etc)

In addition to invoking methods on web resources:

- *Content negotiation*: information – what data representations client can accept (e.g, language, media type)
- *Authentication*: Credentials and challenges to support password-style authentication
  - When a client receives a challenge, it gets the user to type a name and password and submits the associated credentials with subsequent requests

HTTP – implemented over TCP

Original version of the protocol – client-server interaction steps:

•The client requests and the server accepts a connection at the default server

- The client sends a request message to the server
- The server sends a reply message to the client
- The connection is

closed Later version

- *persistent connections* – connections remain open ofer a series of request-reply exchanges
  - – client may receive a message from the server saying that the connection is closed while it is in the middle of sending another request or requests
    - * browser will resend the requests without user involvement, provided that the operations involved are *idempotent* (like GET-method)
    - * otherwise – consult with the user

– resources can be represented as byte sequences and may be compressed

- Multipurpose Internet Mail Extensions (MIME) – RFC 2045 – standard for sending multipart data containing, for example, text, images and sound

HTTP methods

- **GET**: Requests the resource whose URL is given as its argument. If the URL refers to data, then the web server replies by returning the data identified
  - Arguments may be added to the URL; for example, GET can be used to send the contents of a form to a program as an argument
- **HEAD**: identical to GET, but does not return any data but instead, all the infor- mation about the data
- **POST**: data supplied in the body of the request, action may change data on

- **PUT**: Requests that the data supplied in the request is stored with the given URL as its identifier, either as a modification of an existing resource or as a new resource

- **DELETE**: deletes the resource identified by the given URL

- **OPTIONS**: server supplies the client with a list of methods it allows to be applied to the given URL (for example GET, HEAD, PUT) and its special requirements

- **TRACE**: The server sends back the request message. Used for diagnostic purposes

operations PUT and DELETE – idempotent, but POST is not necessarily

Message contents

Figure 5.6 HTTP *Request* mes

method	*URL or pathname*	*HTTP version*	*headers*	*message body*
GET	//www.dcs.qmw.ac.uk/index.html	HTTP/ 1.1		

Figure 5.7 HTTP *Reply*

HTTP version	*status code*	*reason*	*headers*	*message body*
HTTP/1.1	200	OK		resource data

5.3 Remote procedure call (RPC)

- Concept by Birrell and Nelson [1984]

5.3.1 Design issues for RPC

Three issues we will look:

- the style of programming promoted by RPC – programming with interfaces
- the call semantics associated with RPC
- the key issue of transparency and how it relates to remote procedure calls

Programming with interfaces

Interfaces in distributed systems: In a distributed program, the modules can run in separate processes

service interface – specification of the procedures offered by a server, defining the types of the arguments of each of the procedures

(RPC) number of benefits to programming with interfaces in distributed

systems (separa-

tion between interface and implementation):

- programmers are concerned only with the abstraction offered by the service interface and need not be aware of implementation details
- not need to know the programming language or underlying platform used to implement the service (heterogeneity)
- implementations can change as long as long as the interface (the external view) remains the same

Distributed nature of the underlying infrastructure:

- not possible for a client module running in one process to access the variables in a module in another process
- parameter-passing mechanisms used in local procedure calls (e.g., call by

– parameters as input or output

- addresses cannot be passed as arguments or returned as results of calls to remote modules

Interface definition languages (IDLs)

designed to allow procedures implemented in different languages to invoke one another

- IDL provides a notation for defining interfaces in which each of the parameters of an operation may be described as for input or output in addition to having its type specified

Figure 5.8 CORBA IDL example

210 Remote invocation 5.3

```
// In file Person.idl
struct Person {
        string name; string place;
        long year;
};
interface PersonList {
        readonly attribute string listname;
        void addPerson(in Person p);
        void getPerson(in string name, out Person p);
        long number();
};
```

RPC call semantics

doOperation implementations with different delivery guarantees:

- Retry request message
- Duplicate filtering
- Retransmission of

resu

sem

Fault tolerance measures			*Call 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*

Maybe semantics – remote procedure call may be executed once or not at all

- when no fault-tolerance measures applied, can suffer from
  - omission failures (the request or result message lost)
  - crash failures

At-least-once semantics – can be achieved by retransmission of request messages

- types of failures
  - crash failures when the server containing the remote procedure fails
  - arbitrary failures – in cases when the request message is retransmitted, the remote server may receive it and execute the procedure more than once, possibly causing wrong values stored or returned
  - If the operations in a server can be designed so that all of the procedures in their service interfaces are idempotent operations, then at-least-once

At-most-once semantics – caller receives either a result or an exception

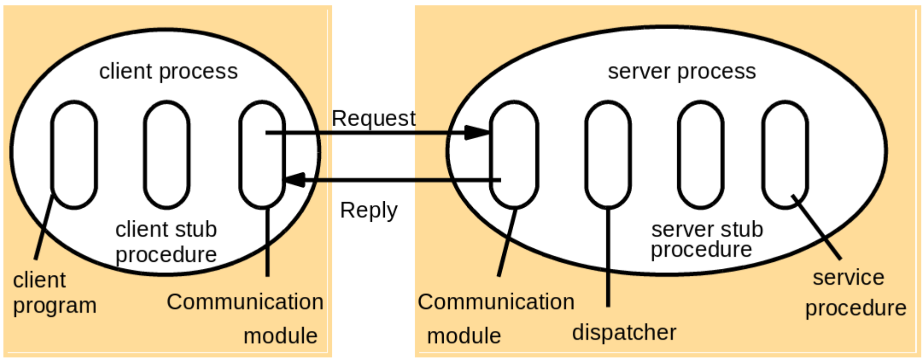
Transparency

at least location and access transparency

consensus is that remote calls should be made transparent in the sense that the syntax of a remote call is the same as that of a local invocation, but that the difference between local and remote calls should be expressed in their interfaces

End of week 5

5.3.2 Implementation of RPC



procedures in RPC

stub procedure behaves like a local procedure to the client, but instead of executing the call, it marshals the procedure identifier and the arguments into a request message, which it sends via its communication module to the server

- RPC generally implemented over request-reply protocol
- general choices:
  - *at-least-once* or
  - *at-most-once*

5.3.3 Case study: Sun RPC

- designed for client-server communication in Sun Network File System (NFS)
- interface language called XDR
  - instead of interface names – program number (obtained from central au-

– single input parameter

– procedure definition specifies a procedure signature and a procedure number

Figure 5.11 Files interface in Sun XDR

```
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;
}; //...
```

```
//...continued:
struct readargs {
    FileIdentifier f;
    FilePointer position;
    Length length;
};

program FILEREADWRITE {
  version VERSION {
    void WRITE(writeargs)=1;          //1
    Data READ(readargs)=2;            //2
  }=2;  //version nunber=2
} = 9999;  //program number=999
```

SRM UNIVERSITY (Under section 3 of UGC Act 1956)

- interface compiler *rpcgen* can be used to generate the following from an inter-face definition:
- client stub procedures
- server main procedure, dispatcher and server stub procedures
- XDR marshalling and unmarshalling procedures for use by the dispatcher and client and server stub procedures

Further on Sun RPC:http://www.cdk5.net/rmi

5.4 Remote method invocation (RMI)

Remote method invocation (RMI) closely related to RPC but extended into the world of distributed objects

- a calling object can invoke a method in a potentially remote object. As with RPC, the underlying details are generally hidden from the user

Similarities between RMI and RPC, they both:

- support programming with interfaces
- typically constructed on top of request-reply protocols
- can offer a range of call semantics, such as
  - *at-least-once*
  - *at-most-once*

- local and remote calls employ the same syntax
- remote interfaces
  - typically expose the distributed nature of the underlying call e.g. sup- porting remote exceptions

RMI added expressiveness for programming of complex distributed applications and services:

- full expressive power of object-oriented programming
  - use of objects, classes and inheritance
  - objectoriented design methodologies and associated tools
- all objects in an RMI-based system have unique object references (independent of they are local or remote)
  - object references can also be passed as parameters ⇒ offering significantly richer parameter passing semantics than in RPC

1. Design issues for RMI

Transition from *objects* to *distributed objects*

The object model

some langueages allow accessing object instance variables directly (C++, Java) – in distributed object system, object’s data can be accessed only with the help of its methods

Object references: to invoke a method object’s reference and method name are given

Interfaces: definition of the signatures of a set of methods without their implementation

Actions: initiated by an object invoking a method in another object three effects of invocation of a method:

1.The state of the receiver may be changed

2. A new object may be instantiated, for example, by using a constructor in Java or C++

3.Further invocations on methods in other objects may take place

Exceptions: a block of code may be defined to *throw* an exception; another block *catches* the exception

Garbage collection: ...Java *vs* C++ case...

Distributed objects

Distributed object systems – different possible architectures

•client-server architecture ... but also possibly:

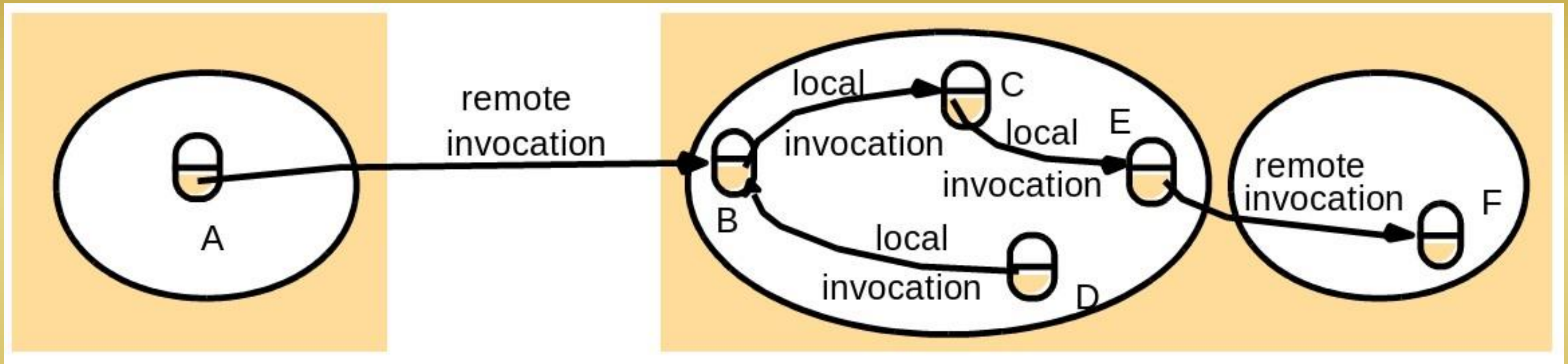
•replicated objects – for enhanced performance and fault-tolerance

•migrated objects – enhanced availability and performance

The distributed object model

Each process contains a collection of objects

objects that can receive remote invocations – ***remote objects***

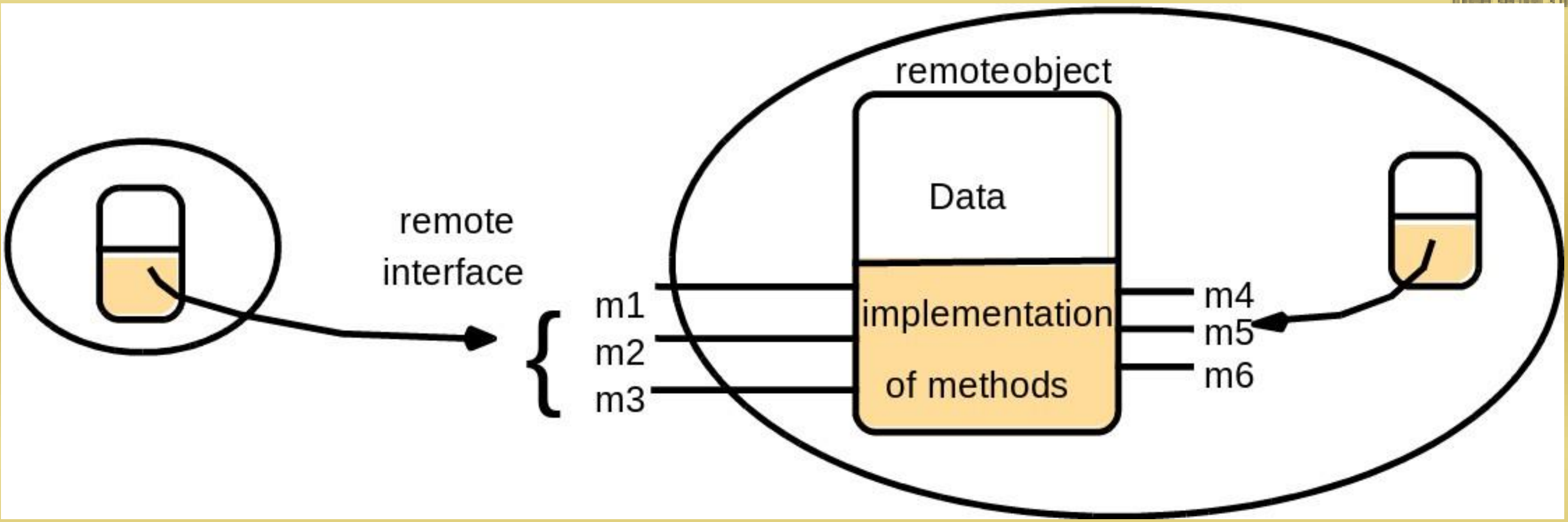


Remote object reference: identifier that can be used throughout a distributed sys- tem to refer to a particular unique remote object

- Remote object references may be passed as arguments and results of remote method invocations

Remote interfaces: which of the object methods can be invoked remotely

Figure 5.12 A remote object and its remote



•CORBA interface definition language (IDL)

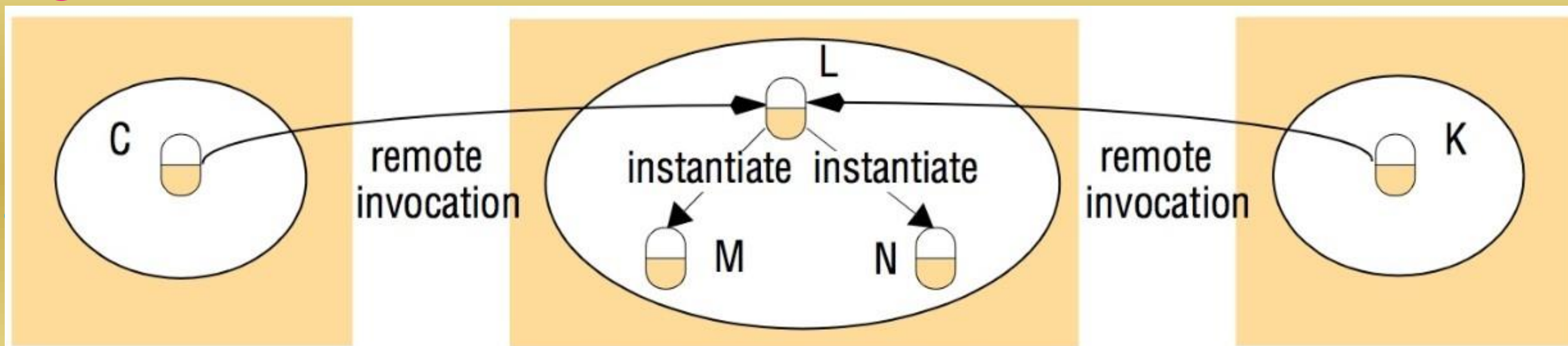
•Java RMI – keyword: *Remote*

NB! Remote interfaces cannot contain constructors!

Actions in a distributed object system

•remote reference of the object must be available to the invoker

Figure 5.14 Instantiation of remote



22

invocation (RMI) Remote object references may be obtained as the results of remote method invocations

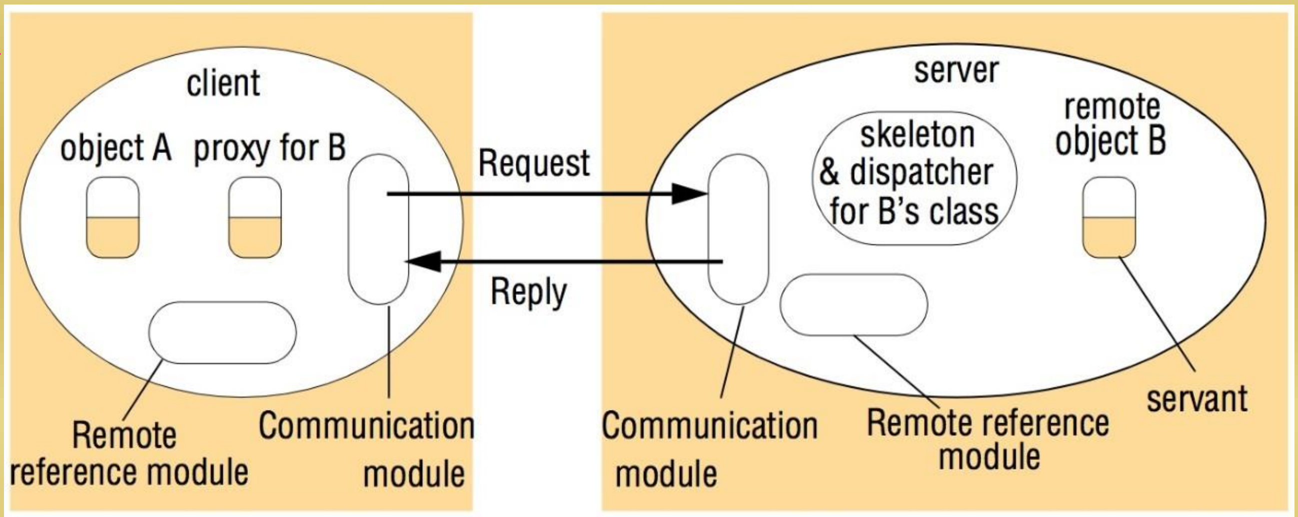
Garbage collection in a distributed-object system:

if garbage collection supported by the language (*e.g.* Java) – also RMI should allow it + a module for distributed reference counting

Exceptions: usual exceptions + *e.g.* timeouts

5.4.2 Implementation of RMI

Figure 5.15 The role of proxy and skeleton in remote method invocation



We will discuss:

- What are the roles of each of the com-ponents?
- What are communication and remote reference modules?
- What is the role of RMI software that runs over them?
- What is generation of proxies and why is it needed?
- What is binding of names to their re-mote object references?
- What is the activation and passivation of objects?

Communication module

– responsible for transfering *request* and *reply* messages between the client and server uses only 3 fields of the messages: *message type*, *requestId* and *remote reference* (Fig. 5.4)

communication modules are together responsible for providing a specified invoca- tion semantics, for example *at-most-once*

Remote reference module

– responsible for translating between local and remote object references and for cre- ating remote object references

using *remote object table* – correspondence between local object references in that process and remote object references

- An entry for all the remote objects held by the process
- An entry for each local proxy

(RMI) Actions of the remote reference module:

- When a remote object is to be passed as an argument or a result for the first time, the remote reference module creates a remote object reference, and adds it to its table

- When a remote object reference arrives in a request or reply message, the remote reference module is asked for the corresponding local object reference, which may refer either to a proxy or to a remote object

  – In the case that the remote object reference is not in the table, the RMI software creates a new proxy and asks the remote reference module to add it to the table

Servants

– instance of a class providing the body of a remote object

- handles the remote requests passed on by the corresponding skeleton

- living within a server process
- created when remote objects instantiated
- remain in use until they are no longer needed (finally being garbage collected or deleted)

The RMI software

Proxy: making remote method invocation transparent to clients – behaving like a local object to the invoker

- forwards invocation in a message to a remote object
- hides the details of:
  - remote object reference
  - marshalling of arguments, unmarshalling of results

– sending and receiving of messages from the client

•just one proxy for each remote object for which a process holds a remote object reference

•implements:

– the methods in the remote interface of the remote object it represents

– each method of the proxy marshals:

* a reference to the target object

* its own *operationId* and its arguments

– ... into a request message and sends it to the target

•then awaits the reply message

– unmarshals it and returns the results to the invoker

(RMI) server has one dispatcher and one skeleton for each class representing a remote object

Dispatcher: receives request messages from the communication module

- uses the *operationId* to select the appropriate method in the skeleton, passing on the request message

Skeleton: implements the methods in the remote interface

- unmarshals the arguments in the request message
- invokes the corresponding method in the servant
- waits for the invocation to complete
- marshals the result (together with any exceptions in a reply message to the send- ing proxy's method)

Generation of the classes for proxies, dispatchers and skeletons

Dynamic invocation: An alternative to proxies

–useful in applications where some of the interfaces of the remote objects cannot be predicted at design time

- dynamic downloading of classes to clients (available in Java RMI) – an alterna- tive to dynamic invocation

- Dynamic skeletons
  - Java RMI generic dispatcher and the dynamic downloading of classes to the server
  - (book Chapter 8 on CORBA)

Server and client programs

Server program : classes for

- dispatchers, skeletons, supported servants +

- initialization section
  - creating and initializing at least one of the hosted servants, which can be used to access the rest
  - may also register some of its servants with a binder

Client program: classes for proxies for all of the remote objects that it will invoke

- can use a binder to look up remote object references

Factory methods:

remote object interfaces cannot include constructors ⇒ servants cannot be created this way

- Servants created either in
  - the initialization section or by
  - factory methods – methods that create servants

•factory object – an object with factory methods

Any remote object that needs to be able to create new remote objects on demand for clients must provide methods in its remote interface for this purpose.

> Such methods are called **factory methods**

The binder in a distributed system

binder – a separate service that maintains a table containing mappings from textual names to remote object references

•binder used by:

- servers to register their remote objects by name
- clients to look them up

CORBA Naming Service – Chapter 8

(RMI) The Java binder – RMIregistry, see case study on Java RMI in Section 5.5

Server threads

– each remote invocation executed on a separate thread – (to avoid blocking)

... programmer has to take it into account...

Activation of remote objects

active-passive modes of service objects – to economise on resources

- *active* object - available for invocation
- *passive* object -

1. the implementation of its methods
2. its state in the marshalled form

Activation: creating an active object from the corresponding passive object by

- creating a new instance of its class
- initializing its instance variables from the stored

state An *activator* is responsible for:

- registering passive objects that are available for activation (involves recording the names of servers against the URLs or file names of the corresponding passive objects)
- starting named server processes and activating remote objects in them
- keeping track of the locations of the servers for remote objects that it has already activated

- Java RMI – the ability to make remote objects activatable [java.sun.com IX]

- CORBA case study in Chapter 8 describes the implementation repository
  - – a weak form of activator that starts services containing objects in an initial state

Persistent object stores

An object that is guaranteed to live between activations of processes is called a persis- tent object

- generally managed by persistent object stores, which store their state in a mar- shalled form on disk

Object location

remote object reference – Internet address and port number of the process that created the remote object – to quarantee uniqueness

(RMI) some remote objects exist in series of different processes, possibly on different computers, throughout their lifetime

location service – helping clients to locate remote objects from their remote object references

- using database: remote object reference ⟶ probable current location

5.4.3 Distributed garbage collection

Java distributed garbage collection algorithm

- server keeping track, which of its objects are proxied at which clients
  - – protocol for creation and removal of proxies with notifications to the server
- based on no client proxies to an object exist and no local references either,

Leases in Jini

•references to a certain object are *leased* to other (outside) processes

•leases have a certain pre-negotiated time period

•before the lease is about to expire, the client must request a renewal if needed

5.5 Case study: Java RMI

Example: *shared whiteboard* www.cdk5.net/rmi

Remote interfaces in Java RMI

- extending an interface *Remote* in *java.rmi* package
- must throw *RemoteException*

Figure 5.16 Java Remote interfaces Shape and ShapeList

```
import java.rmi.*;
import java.util.Vector;
public interface Shape extends Remote {  // i.e. Shape is a remote interface
    int getVersion() throws RemoteException;
    GraphicalObject getAllState() throws RemoteException; // 1
}
public interface ShapeList extends Remote {
    Shape newShape(GraphicalObject g) throws RemoteException; // 2
    Vector allShapes() throws RemoteException;
    int getVersion() throws RemoteException;
```

Parameter and result passing

In Java RMI:

- parameters of a method – *input* parameters
- result of a method – single *output* parameter

Any object that is serializable – implements the *Serializable* interface – can be passed as an argument or result in Java RMI.

- All primitive types and remote objects are serializable

***Passing remote objects*:** When the type of a parameter or result value is defined as a remote interface, the corresponding argument or result is always passed as a remote object reference

Passing non-remote objects: All serializable non-remote objects are copied and passed by value

RMI The arguments and return values in a remote invocation are serialized to

a stream

using the method described in Section 4.3.2, with the following modifications:

1. Whenever an object that implements the Remote interface is serialized, it is replaced by its remote object reference, which contains the name of its (the remote object's) class

2. When any object is serialized, its class information is annotated with the location of the class (as a URL), enabling the class to be downloaded by the receiver

Downloading of classes

- If the recipient does not already possess the class of an object passed by value, its code is downloaded automatically

- if the recipient of a remote object reference does not already possess the

RMI 1.There is no need for every user to keep the same set of classes in their working
environment

2.Both client and server programs can make transparent use of instances of new classes whenever they are added

RMIregistry

– binder for Java RMI

- on every server computer that hosts remote objects
- maintains a table mapping textual, URL-style names to references to remote objects hosted on that computer
- accessed by methods of the Naming class
  - methods take as an argument a URL-formatted string of the form:

//computerName:port/objectName

Figure 5.17 The Naming class of Java RMIregistry

void rebind (String name, Remote obj)

This method is used by a server to register the identifier of a remote object by name, as shown in Figure 15.18, line4.

void bind (String name, Remote obj)

This method can alternatively be used by a server to register a remote object by name, but if the name is already bound to a remote object reference an exception is thrown.

void unbind (String name, Remote obj)

This method removes a binding.

Remote lookup(String name)

This method is used by clients to look up a remote object by name, as shown in Figure 5.20 line 1. A remote object reference is returned.

String [] list()

This method returns an array of Strings containing the names bound in the registry.

5.5.1 Building client and server programs

Server program

Figure 5.18 Java class ShapeListServer with main method

```
import java.rmi.*;
import java.rmi.server.UnicastRemoteObject;
public class ShapeListServer {
    public static void main(String args[]){
        System.setSecurityManager(new RMISecurityManager());
        try{
            ShapeList aShapeList = new ShapeListServant();          //1
            ShapeList stub =                                         //2
                (ShapeList) UnicastRemoteObject.exportObject(aShapeList,0);   //3
            Naming.rebind("//bruno.ShapeList", stub);               //4
            System.out.println("ShapeList server ready");
        }catch(Exception e) {
            System.out.println("ShapeList server main " + e.getMessage());}
    }
}
```

Figure 5.19 Java class ShapeListServant implements interface ShapeList

```java
import java.util.Vector;
public class ShapeListServant implements ShapeList {
    private Vector theList; // contains the list of Shapes
    private int version;
    public ShapeListServant(){...}
    public Shape newShape(GraphicalObject g) {          //1
        version++;
        Shape s = new ShapeServant( g, version);        //2
        theList.addElement(s);
        return s;
    }
    public Vector allShapes(){...}
    public int getVersion() { ... }
}
```

SRM UNIVERSITY (Under section 3 of UGC Act 1956)

Client program

Figure 5.20 Java client of ShapeList

```
import java.rmi.*;
import java.rmi.server.*;
import java.util.Vector;
public class ShapeListClient{
    public static void main(String args[]){
        System.setSecurityManager(new RMISecurityManager());
        ShapeList aShapeList = null;
        try{
            aShapeList = (ShapeList) Naming.lookup("//bruno.ShapeList"); //1
            Vector sList = aShapeList.allShapes();                        //2
        } catch(RemoteException e) {System.out.println(e.getMessage());
        }catch(Exception e) {System.out.println("Client: " + e.getMessage());
        }
    }
}
```

5.5 Callbacks

server should inform its clients whenever certain event occurs

callback – server's action of notifying clients about an event

- client creates a remote object – *callback object* – that implements an interface containing a method for the server to call
- server provides an operation allowing interested clients to inform it of the remote object references of their *callback objects*
- Whenever an event of interest occurs, the server calls the interested

clients Problems with polling solved, but at the same time, attention is

needed because:

- server needs to have up-to-date lists of the clients' callback objects, but clients may not always inform the server before they exit, leaving the server with in-

– leasing technique can be used to overcome this problem

- server needs to make a series of synchronous RMIs to the *callback objects* in the list

– TextBook Chapter 6 gives some ideas on solving this issue

⇒WhiteboardCallback interface could be defined as:

```
public interface WhiteboardCallback implements Remote {
    void callback(int version) throws RemoteException;
};
```

– implemented as a remote object by the client

- client needs to inform the server about its callback object

ShapeList interface requires additional methods such as register and deregister, defined as follows:

```
int register(WhiteboardCallback callback) throws RemoteException;
void deregister(int callbackId) throws RemoteException;
```

5.5.2 Design and implementation of Java RMI

Use of reflection

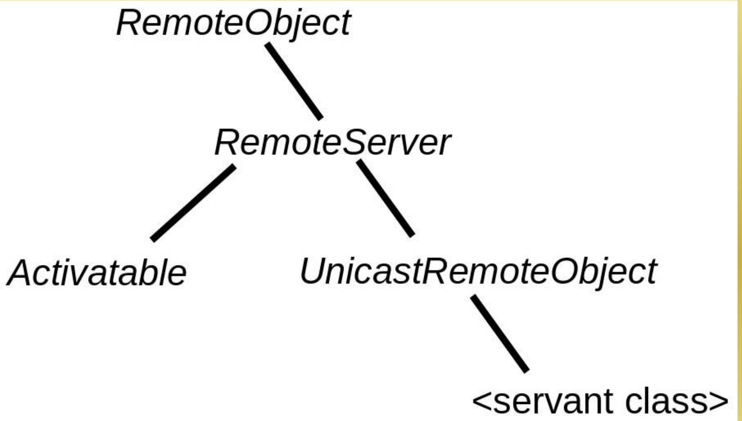
Reflection used to pass information in request messages about the method to be invoked.

– with the help of the Method class in reflection package

Java classes supporting RMI

Inheritance structure of the classes supporting Java RMI servers:

Figure 5.21 Classes supporting Java



End of week 6

6 Indirect communication

6.1 Introduction

Roger Needham, Maurice Wilkes and David Wheeler: *“All problems in computer science can be solved by an- other level of indirection”*

Indirect communication –What does it mean? communication be- tween entities in a distributed system through an intermediary with no direct coupling be- tween the sender and the receiver(s)

2 key properties stemming from the use of an intermediary:

1. ***Space uncoupling***

   •the sender does not know or need to know the identity of the receiver(s)

   •participants (senders or receivers) can be replaced, updated, replicated or migrated

2. ***Time uncoupling***

   •the sender and receiver(s) can have independent lifetimes

   – ⇒ more volatile environments where senders and receivers may come and go

Figure 6.1 Space and time coupling in distributed systems

	Time-coupled	*Time-uncoupled*
Space coupling	*Properties:* Communication directed towards a given receiver or receivers; receiver(s) must exist at that moment in time *Examples:* Message passing, remote invocation (see Chapters 4 and 5)	*Properties:* Communication directed towards a given receiver or receivers; sender(s) and receiver(s) can have independent lifetimes *Examples:* See Exercise 6.3
Space uncoupling	*Properties:* Sender does not need to know the identity of the receiver(s); receiver(s) must exist at that moment in time *Examples:* IP multicast (see Chapter 4)	*Properties:* Sender does not need to know the identity of the receiver(s); sender(s) and receiver(s) can have independent lifetimes *Examples:* Most indirect communication paradigms covered in this chapter

Introduction

The relationship with asynchronous communication

- In asynchronous communication, a sender sends a message and then continues (without blocking) ⇒ no need to meet in time with the receiver to communicate
- Time uncoupling adds the extra dimension that the sender and receiver(s) can have independent existences

6.2 Group communication

Group communication – a message is sent to a group ⟶ message is delivered to all members of the group

- the sender is not aware of the identities of the receivers

Key areas of application:

- the reliable dissemination of information to potentially large numbers of

•support for a range of fault-tolerance strategies, including the consistent update of replicated data

•support for system monitoring and management

JGroups toolkit

1. **The programming model**

group & *group* membership < processes may join or leave the group

aGroup.send(aMessage))

process groups

• *e.g.* RPC

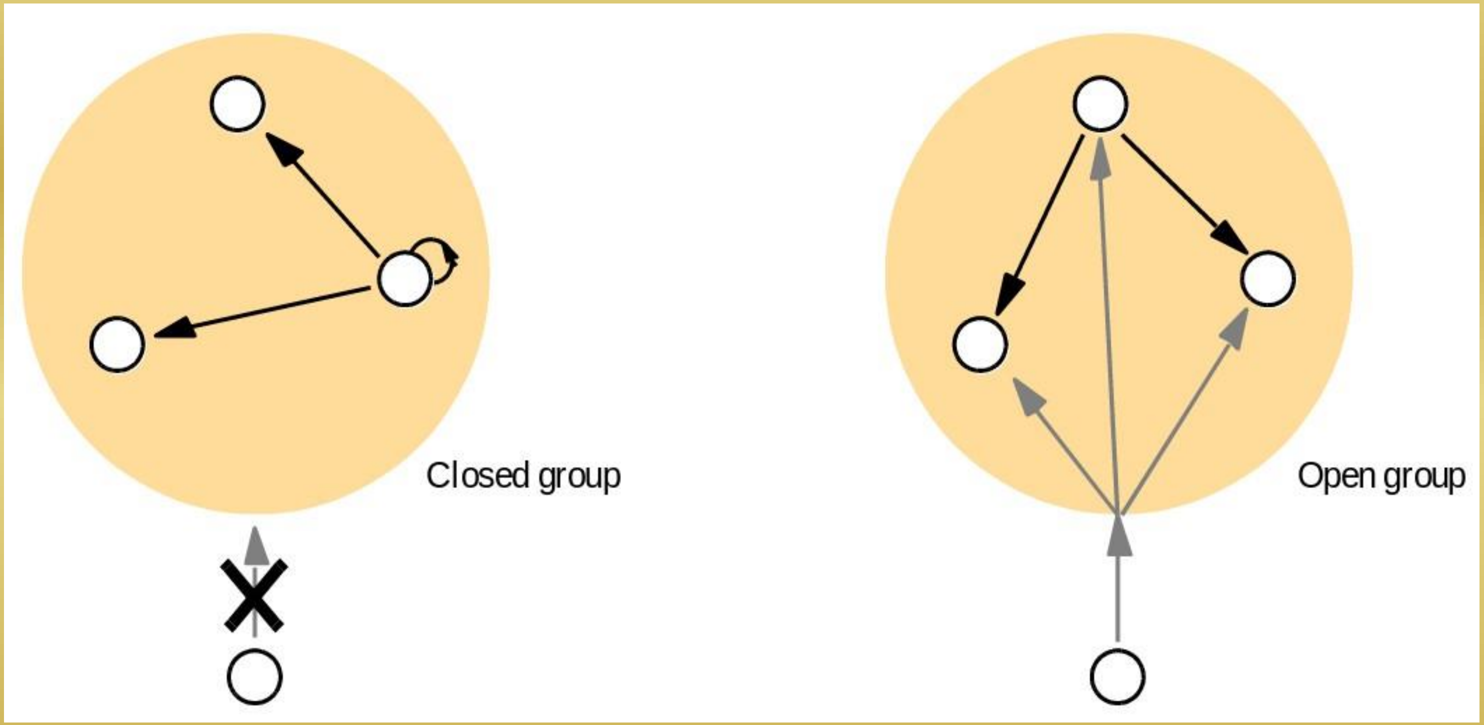
object groups

•marshalling and dispatching as in RMI

•Electra – CORBA-compliant system supporting object groups

closed and open groups

Figure 6.2 Open and closed



overlapping and non-overlapping groups

synchronous and asynchronous systems

2. Implementation issues

Reliability and ordering in multicast

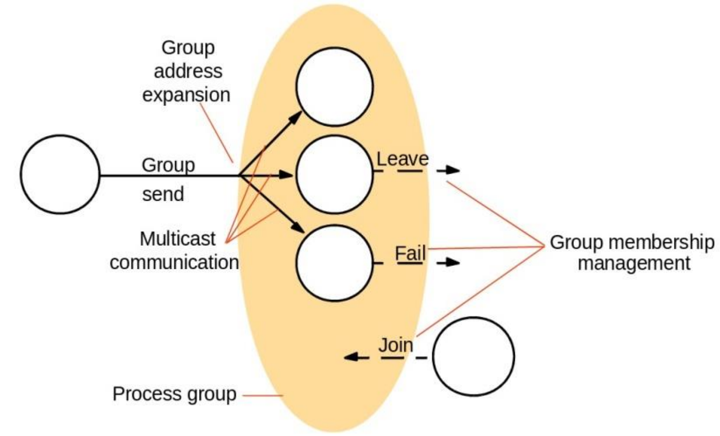
integrity, validity + agreement

ordered multicast possibilities (hybrid solutions also possible) :

- FIFO ordering: First-in-first-out (FIFO) (or source ordering) – if sender sends one before the other, it will be delivered in this order at all group processes
- Casual ordering: – if a message happens before another message in the dis- tributed system, this so-called casual relationship will be preserved in the deliv- ery of the associated messages at all processes
- Total ordering: – if a message is delivered before another message at one pro- cess, the same order will be preserved at all processes

Group membership management

Figure 6.3 The role of group member- ship management



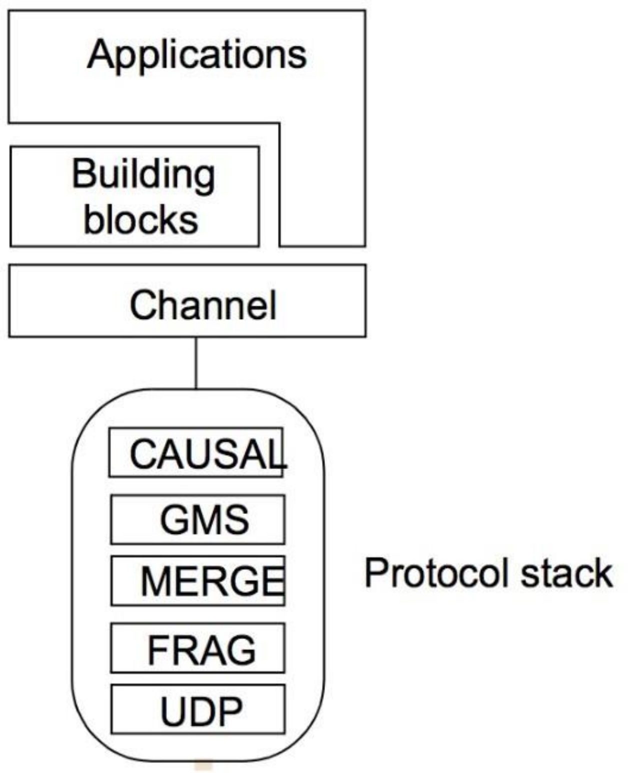
- Providing an interface for group membership changes
- Failure detection
- Notifying members of group mem- bership changes
- Performing group address expansion

IP multicast as a weak case of a group membership service

- IP multicast itself does not provide group members with information about cur- rent membership delivery; is not coordinated with membership changes

6.2.3 Case study: the JGroups toolkit

Figure 6.4 The architecture



- **Channel** – acts as a handle onto a group

core functions of joining, leaving, sending and receiving

- connect – to a particular named group
  if the named group does not exist, it is implicitly created at the time of the first connect
- disconnect – to leave a group
- getView – returns the current member list
- getState – historical application state associated with the

SRM UNIVERSITY (Under section 3 of UGC Act 1956)

261 Indirect communication 6.2

Figure 6.5 Java class FireAlarmJG

```java
import org.jgroups.JChannel;
public class FireAlarmJG {
    public void raise() {
        try {
            JChannel channel = new JChannel();
            channel.connect("AlarmChannel");
            Message msg = new Message(null, null, "Fire!"); //destination, source, payload
                //destination=null — distribute to whole group; source null — source
                added automatically by the system anyway
            channel.send(msg);
        }
        catch(Exception e) {
        }
    }
}
```

```java
FireAlarmJG alarm = new FireAlarmJG(); //create a new instance of the FireAlarmJG class
alarm.raise(); //raise an alarm
```

262 Indirect communication 6.2

Figure 6.6 Java class FireAlarmConsumerJG

```
importorg .jgroups .JChannel ;
publicclassFireAlarmConsumerJG
    publicStringawait () {
        try {
            JChannelchannel = newJChannel ();
            channel.connect("AlarmChannel");
            Messagemsg = (Message) channel.receive(0);
//parameter:timeoutzero – thereceivemessagewillblockuntilamessageisreceived
//incomingmessagesarebufferedandreceivereturnsthetopelementinthebuffer
            return (String) msg.GetObject();
        }
        catch (Exceptione) {
            returnnull ;
        }
    }
}
```

```
FireAlarmConsumerJGalarmCall = newFireAlarmConsumerJG () //(...receivercode...)
;
Stringmsg = alarmCall.await();
System.out.println("Alarm␣received:␣" + msg);
```

- **Building blocks**
  – higher-level abstractions, building on the underlying service offered by channels

  – *MessageDispatcher*

    * e.g. castMessage method that sends a message to a group and blocks until a specified number of replies are received

  – *RpcDispatcher* – invokes specified method on all objects associated with a group

  – *NotificationBus* – implementation of a distributed event bus, in which an event is any serializable Java object

- **The protocol stack**
  – underlying communication protocol, constructed as a stack of composable protocol layers

bidirectional stack of protocol layers

```
publicObjectup    (Eventevt );
publicObjectdown    (Eventevt );
```

- UDP most common transport layer in JGroups (IP multicast for sending to all members in a group; TCP layer may be preferred; PING for member- ship discovery etc.)
- FRAG – message packetization to maximum message size (8,192 bytes by default)
- MERGE – unexpected network partitioning and the subsequent merging of subgroups after the partition
- GMS implements a group membership protocol to maintain consistent views of membership across the group

– CAUSAL implements causal ordering (Section 6.2.2 Chapter 15)

6.3 Publish-subscribe systems

also referred to as *distributed event-based* systems

- publishers publish structured events to an event service and subscribers express interest in particular events through subscriptions which can be arbitrary pat- terns over the structured events
- event notifications
- one-to-many communications paradigm

Applications of publish-subscribe systems

application domains needing large-scale dissemination of events Examples:

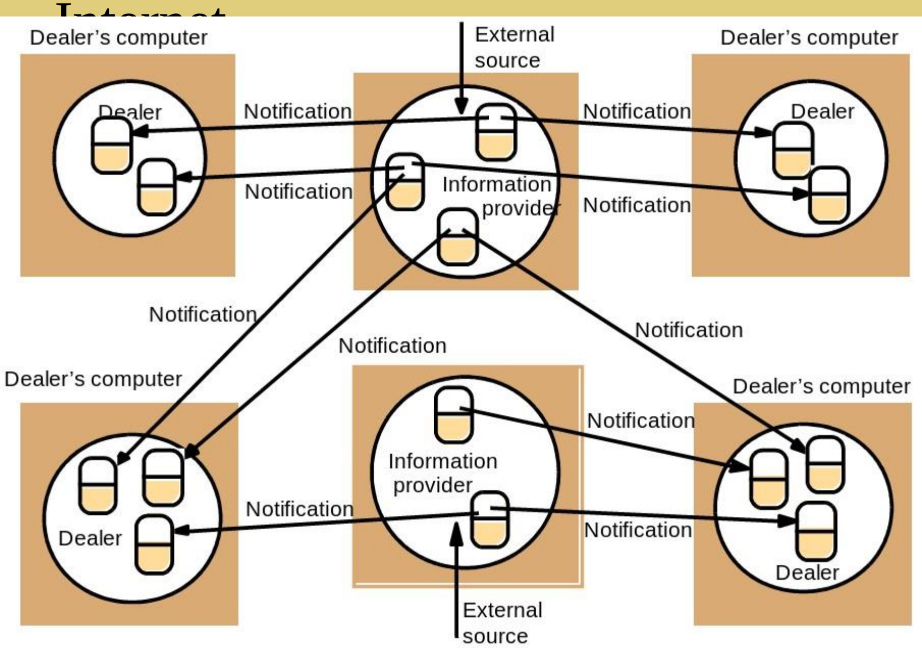
- financial information systems
- other areas with live feeds of real-time data (including RSS feeds)

•support for cooperative working, where a number of participants need to be informed of events of shared interest

•support for ubiquitous computing, including the management of events emanat- ing from the ubiquitous infrastructure (for example, location events)

•a broad set of monitoring applications, including network monitoring in the Internet

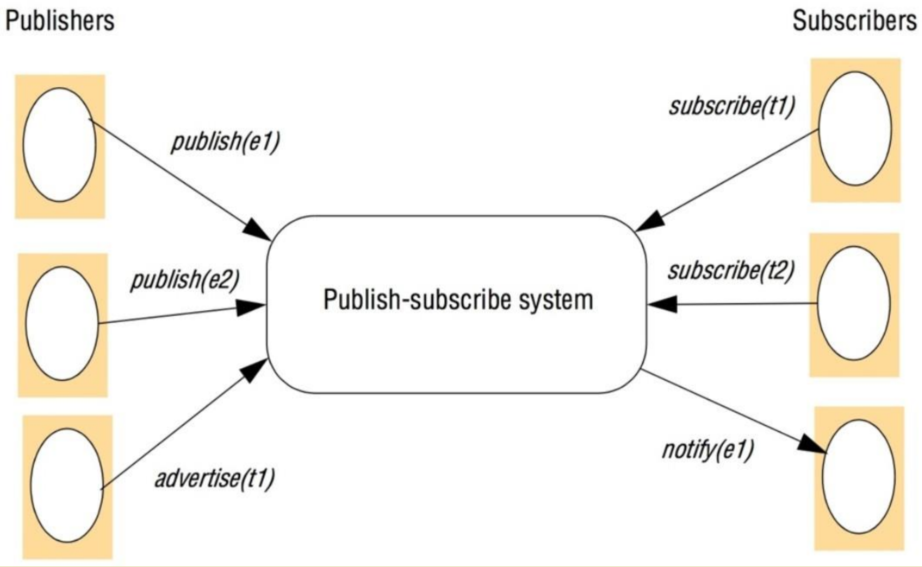
Fig



Characteristics of publish-subscribe systems

- *Heterogeneity*
- *Asynchronicity*
- *different delivery guarantees*

6.3.1 The programming model



(un)publish(event); (un)subscribe(filter); advertise(filter); notify(event)

Expressiveness of publish-subscribe system determined by the **subscription (filter) model**:

- ***Channel-based***
  - publishers publish events to named channels
  - subscribers then subscribe to one of these named channels to receive all events sent to that channel
    - CORBA Event Service (see Chapter 8)
- ***Topic-based*** (also referred to as subject-based):
  - each notification is expressed in terms of a number of fields, with one field denoting the topic
  - Subscription defined in terms of topic of interest

- generalization of topic-based approaches allowing the expression of sub- scriptions over a range of fields in an event notification

- **Type-based**
  - subscriptions defined in terms of types of events
  - matching is defined in terms of types or subtypes of the given filter
- + *concept-based subscription models*
  - filters are expressed in terms of the semantics as well as the syntax of events
- + *complex event processing (or composite event detection)*
  - allows the specification of patterns of events as they occur in the distributed environment

6.3.2 Implementation issues

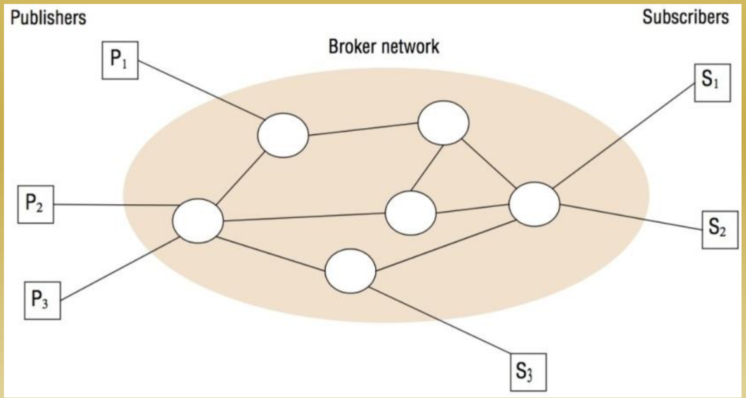
task of a publish-subscribe system: ensure that events are delivered efficiently to all subscribers that have filters defined that match the event

additional requirements in terms of security, scalability, failure handling, concur- rency and quality of service

Centralized versus distributed implementations

centralised broker vs. *network of brokers*

Figure 6.9 A network of brokers

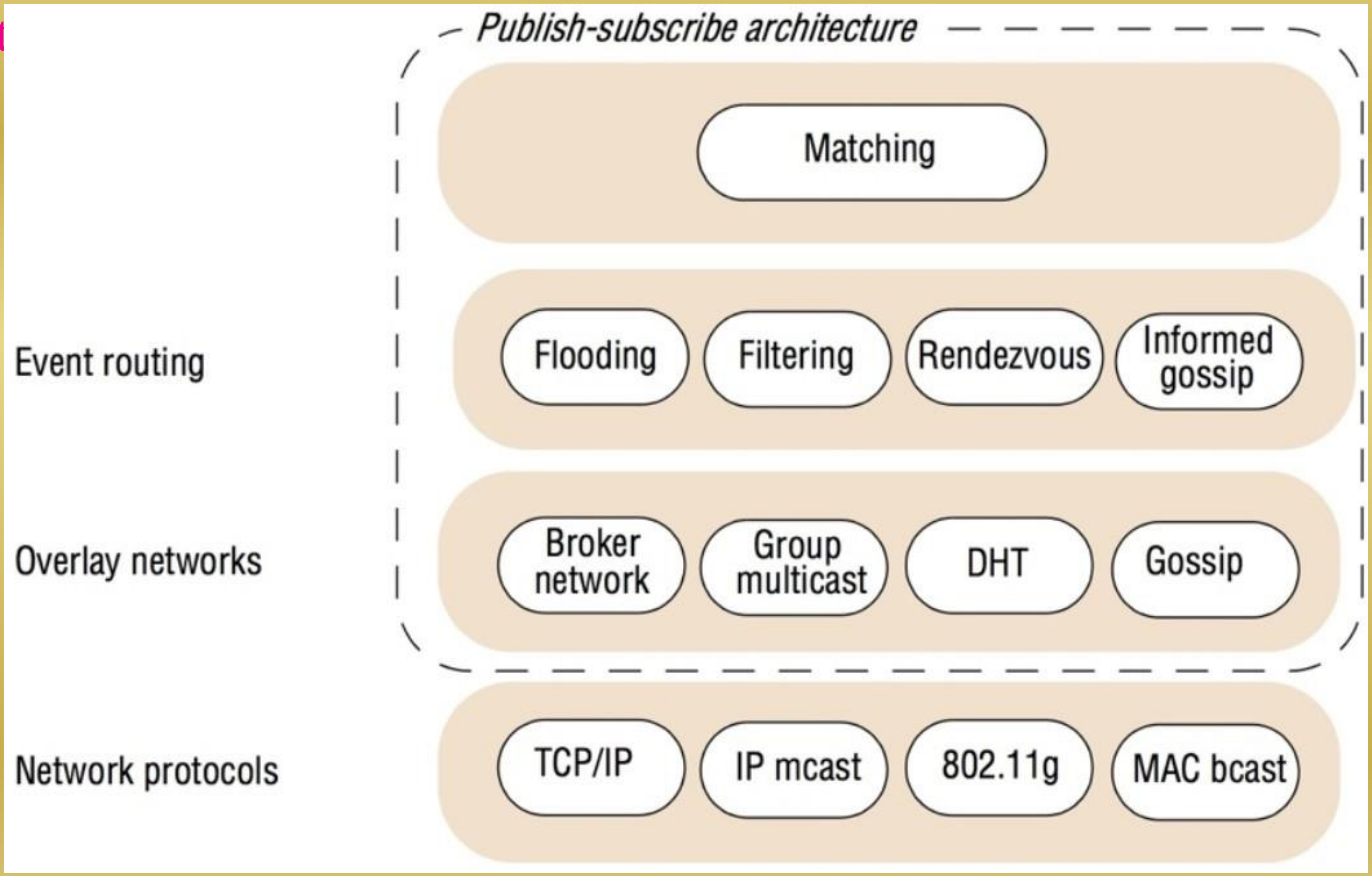


A step further:

fully peer-to-peer implementation of a publish-subscribe system – no distinc- tion between publishers, subscribers and brokers; all nodes act as brokers, coop- eratively implementing the required event routing functionality

Overall systems architecture

Figure 6.10 The architecture of publish-subscribe syst



Implementation approaches:

- ***Flooding***:
  - sending an event notification to all nodes in the network and then carrying out the appropriate matching at the subsciber end
  - alternative – send subscriptions back to all possible publishers, with the matching carried out at the publishing end
  - can be implemented
    - * using an underlying broadcast or multicast facility
    - * brokers can be arranged in an acyclic graph in which each forwards incoming event notifications to all its neighbours
  - benefit of simplicity but can result in a lot of unnecessary network traffic
- ***Filtering*** (filtering-based routing)
  - Brokers forward notifications through the network only where there is a

– each node must maintain

* neighbours list containing a list of all connected neighbours in the network of brokers
* subscription list containing a list of all directly connected subscribers serviced by this node
* routing table

Figure 6.11 Filtering-based routing

```
upon receive publish (event e) from node x
    matchlist := match(e, subscriptions)
    send notify(e) to matchlist; fwdlist := match(e, routing);
    send publish(e) to fwdlist - x;
upon receive subscribe (subscription s) from node x
    if x is client then
        add x to subscriptions
    ;
    else add (x, s) to routing ;
    send subscribe (s) to neighbours - x;
```

∗ subscriptions essentially using a flooding approach back towards all possible publishers

- ***Advertisements***: propagating the advertisements towards subscribers in a simi- lar (actually, symmetrical) way to the propagation of subscriptions
- ***Rendezvous***: rendezvous nodes, which are broker nodes responsible for a given subset of the event space
  - $SN(s)$ – given subscription, *s* –> one or more rendezvous nodes that take responsibility for that subscription
  - $EN(e)$ – given event *e* –> one or more rendezvous nodes responsible for matching e against subscriptions in the system

Figure 6.12 Rendezbous-based routing

```
upon receive publish(event e) from node x at node i
    rvlist := EN(e);
    if i in rvlist then begin
        matchlist <- match(e, subscriptions);
        send notify(e) to matchlist;
    end
    send publish(e) to rvlist - i;
upon receive subscribe(subscription s) from node x at node i
    rvlist := SN(s);
    if i in rvlist then
        add s to subscriptions;
    else
        send subscribe(s) to rvlist - i;
```

- distributed hash table (DHT) – can be used
  - hash table distributed over a set of nodes in P2P manner

6.3.3 Examples of publish-subscribe systems

Figure 6.13 Example publish-subscribe

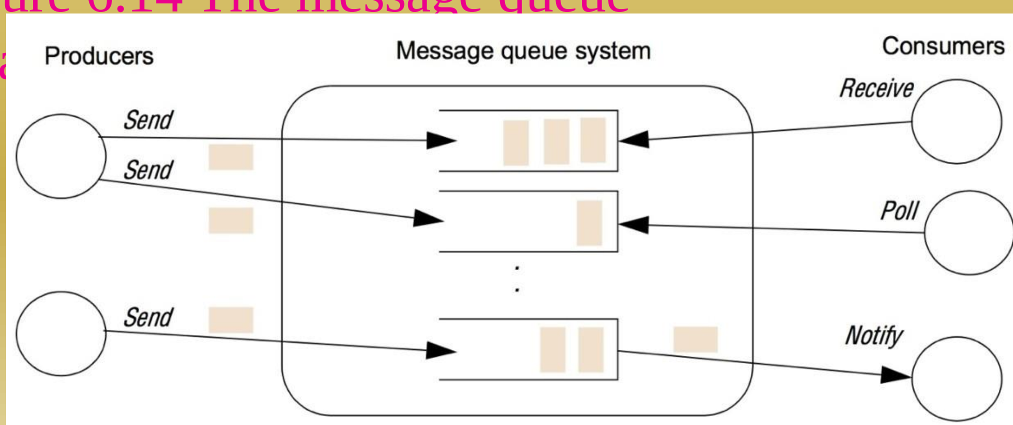
System (and further reading)	*Subscription model*	*Distribution model*	*Event routing*
CORBA Event Service (Chapter 8)	Channel-based	Centralized	-
TIB Rendezvouz [Oki *et al*. 1993]	Topic-based	Distributed	Ffiltering
Scribe [Castro *et al*. 2002b]	Topic-based	Peer-to-peer (DHT)	Rendezvous
TERA [Baldoni *et al*. 2007]	Topic-based	Peer-to-peer	Informed gossip
Siena [Carzaniga *et al*. 2001]	Content-based	Distributed	Filtering
Gryphon [www.research.ibm.com]	Content-based	Distributed	Filtering
Hermes [Pietzuch and Bacon 2002]	Topic- and content-based	Distributed	Rendezvous and filtering
MEDYM [Cao and Singh 2005]	Content-based	Distributed	Flooding
Meghdoot [Gupta *et al*. 2004]	Content-based	Peer-to-peer	Rendezvous
Structure-less CBR [Baldoni *et al*. 2005]	Content-based	Peer-to-peer	Informed gossip

4. Message queues

1. The programming model

- Types of receive operations:
  - blocking receive
  - non-blocking receive
  - notify operation

Figure 6.14 The message queue para



- A number of processes can send messages to the same queue
- a number of receivers can remove messages from a queue
- queuing policy
  - (normally) first-in-first-out (FIFO) but most message queue implementa- tions also support the
  - concept of priority
    - * higher-priority messages delivered first
- Consumer processes

  can select messages from the queue based on message properties
  - destination (a unique identifier designating the destination queue)
  - metadata associated with the message
    - * priority of the message

* the delivery mode
* body of the message (though body – normally opaque and untouched by the message queue system)

•message content serialized

•length of a message varying (can be 100s megabytes...)

messages are persistent – system preserves messages indefinitely (or until they are consumed)

also system can commit messages to disk – for reliable delivery:

•any message sent is eventually received (validity)

•the message received is identical to the one sent, and no messages are delivered twice (integrity)

- support for the sending or receiving of a message to be contained within a trans- action (all or nothing)
- support for message transformation (e.g. in heterogeneous environments)
- support for security

difference with message-passing systems (MPS):

- MPS have implicit queues associated with senders and receivers (for example, the message buffers in MPI),

message queuing systems have explicit queues that are third-party entities, separate from the sender and the receiver – making it into indirect communication paradigm with the crucial properties of space and time uncoupling

2. Implementation issues

Case study: WebSphere MQ (textbook pp.272-274)

3. Case study: The Java Messaging Service (JMS)

JMS – specification of a standardized way for distributed Java programs to communicate indirectly

- unifies the publish-subscribe and message queue paradigms at least superficially by supporting topics and queues as alternative destinations of messages

implementations:

Joram from OW2

Java Messaging from JBoss

Sun's Open MQ

Apache ActiveMQ

OpenJMS

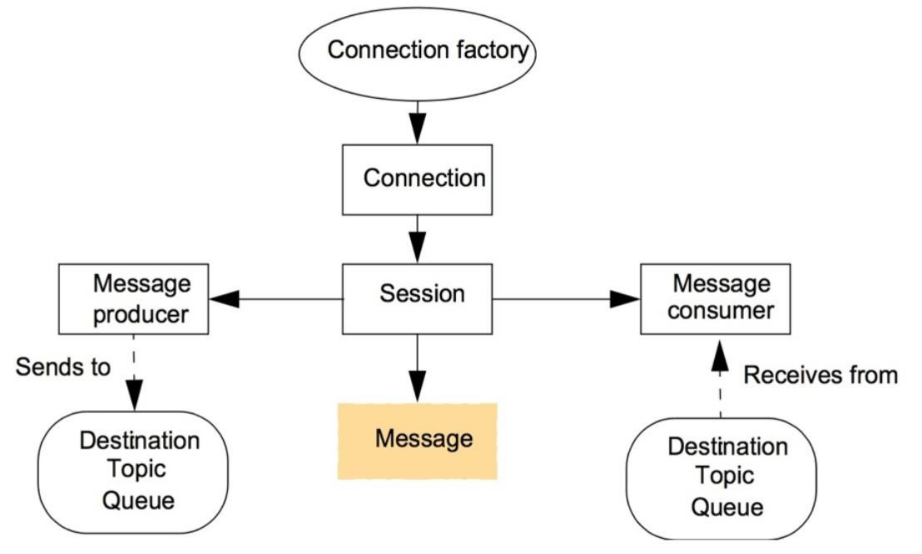
WebSphere MQ provides a JMS interface

Key roles:

- ***JMS client*** – Java program or component that produces or consumes messages
  - JMS producer – program that creates and produces messages
  - JMS consumer – program that receives and consumes messages
- ***JMS provider*** – any of the multiple systems that implement the JMS specification
- ***JMS message*** – object that is used to communicate information between JMS clients (from producers to consumers)
- ***JMS destination*** – object supporting indirect communication in JMS – either:
  - JMS topic

Programming with JMS

Figure 6.16 The programming model offered by JMS



•two types of connection can be established:

- TopicConnection
- QueueConnection

Connections can be used to create one or more sessions

•session – series of operations involving the creation, production and consump- tion of messages related to a logical task

•session object also supports operations to create transactions, supporting all-or- nothing execution of a series of operations

- TopicConnection can support one or more topic sessions
- QueueConnection can support one or more queue sessions (but it is not possible to mix session styles)

session object – central to the operation of JMS – methods for creation of messages, message producers and message consumers:

- ***message*** consists of three parts:

  –header

  * destination – reference to:

SRM UNIVERSITY (Under section 3 of UGC Act 1956)

·topic

·queue

* priority

* expiration date

* message ID

* timestamp

–**properties** – user-defined

–**body** – text message, byte stream, serialized Java object, stream of primi- tive Java values, structured set of name/value pairs

- ***message producer*** – object to publish messages under particular topic or to send messages to a queue
- ***message consumer*** – object to subscribe to messages with given topic or receive messages from a queue

– filters: *message selector* (over header or properties)

  * subset of SQL used to specify properties

– can block using a *receive* operation

– can establish *message listener* object

  * – has to establish method onMessage

A simple example

Figure 6.17 Java class FireAlarmJMS

```java
import javax.jms.*;
import javax.naming.*;
public class FireAlarmJMS {
    public void raise() {
    try {
        Context ctx = new InitialContext();
        TopicConnectionFactory topicFactory = //find factory
            (TopicConnectionFactory) ctx.lookup("TopicConnectionFactory");
        Topic topic = (Topic) ctx.lookup("Alarms"); //topic
        TopicConnection topicConn = //connection
            topicConnectionFactory.createTopicConnection();
        TopicSession topicSess = topicConn.createTopicSession(false,
                            Session.AUTO_ACKNOWLEDGE); //session
        TopicPublisher topicPub = topicSess.createPublisher(topic);
        TextMessage msg = topicSess.createTextMessage(); //create message
        msg.setText("Fire!");
        topicPub.publish(message); //publish it
        } catch (Exception e) {
    }
}
```

```
//create a new instance of the FireAlarmJMS class and then raise an alarm is:
    alarm = new FireAlarmJMS();
    alarm.raise();
```

Figure 6.18 Java class FireAlarmConsumerJMS

```
import javax.jms.*;
import javax.naming.*;
public class FireAlarmConsumerJMS {
    public String await() {
    try {
        Context ctx = new InitialContext();
        TopicConnectionFactory topicFactory =
            (TopicConnectionFactory) ctx.lookup("TopicConnectionFactory")
        ; Topic topic = (Topic) ctx.lookup("Alarms");
        TopicConnection topicConn =
            topicConnectionFactory.createTopicConnection();
        TopicSession topicSess = topicConn.createTopicSession(false,
            Session.AUTO_ACKNOWLEDGE); // ... identical up to here
        TopicSubscriber topicSub = topicSess.createSubscriber(topic)
        ; topicSub.start(); // topic subscriber created and started
        TextMessage msg = (TextMessage) topicSub.receive(); // receive
        return msg.getText(); // return message as string
```

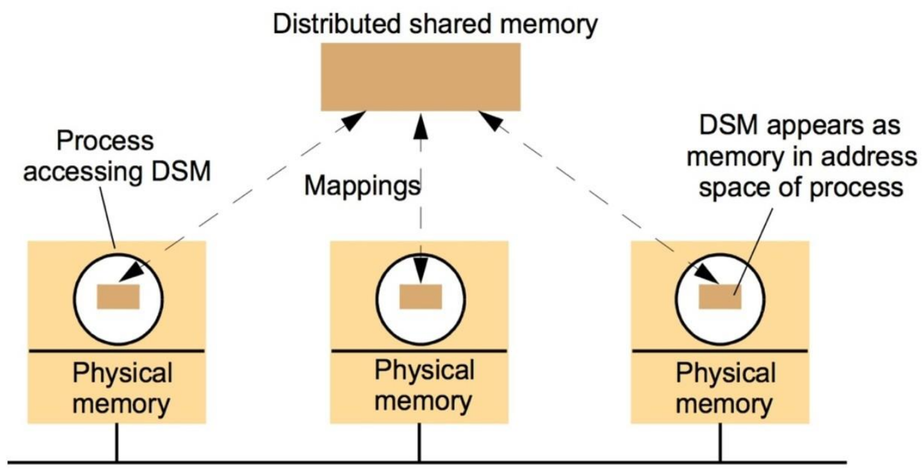
```
    } catch (Exception e) {
        return null;
    }
}
```

```
//class usage by a consumer:
FireAlarmConsumerJMS alarmCall = new FireAlarmConsumerJMS();
String msg = alarmCall.await();
System.out.println("Alarm received: " + msg);
```

6.5 Shared memory approaches

6.5.1 Distributed shared memory (DSM)

Figure 6.19 The distributed shared memory



- DSM – tool for parallel applications
- shared data items available for access directly
- DSM runtime – sends messages with updates between computers
- managed replicated data for faster access

approaches One of the first examples: Apollo Domain file system [1983] – DSM

can be persistent

Non-Uniform Memory Access (NUMA) architecture

- processors see a single address space containing all the memory of all the boards
- access latency for on-board memory less than for a memory module on a differ- ent board

Message passing versus DSM

- ***service offered***
  - message passing: variable marshalled-unmarshalled into variable on other processor
  - DSM – not possible to run on heterogeneous architectures

- via message model
- locks and semaphores in DSM implementations

- DSM can be made persistant

- message-passing systems: processes have to coexist in time

- ***Efficiency*** – very problem-dependent
  - message-passing: suitable for hand-tuning on supercomputer-sized clusters
  - DSM – can be made to perform as well at least for small numbers of pro- cessors

2. Tuple space communication

- David Gelernter [1985], Yale University
- *generative communication*
  - processes communicate indirectly by placing tuples in a tuple space
  - from which other processes can read or remove them
  - Tuples
    - do not have an address
    - are accessed by pattern matching on content (*content-addressable memory*)
    - consist of a sequence of one or more typed data fields such as
      - <"fred", 1958>
      - <"sid", 1964>
      - <4, 9.8, "Yes">

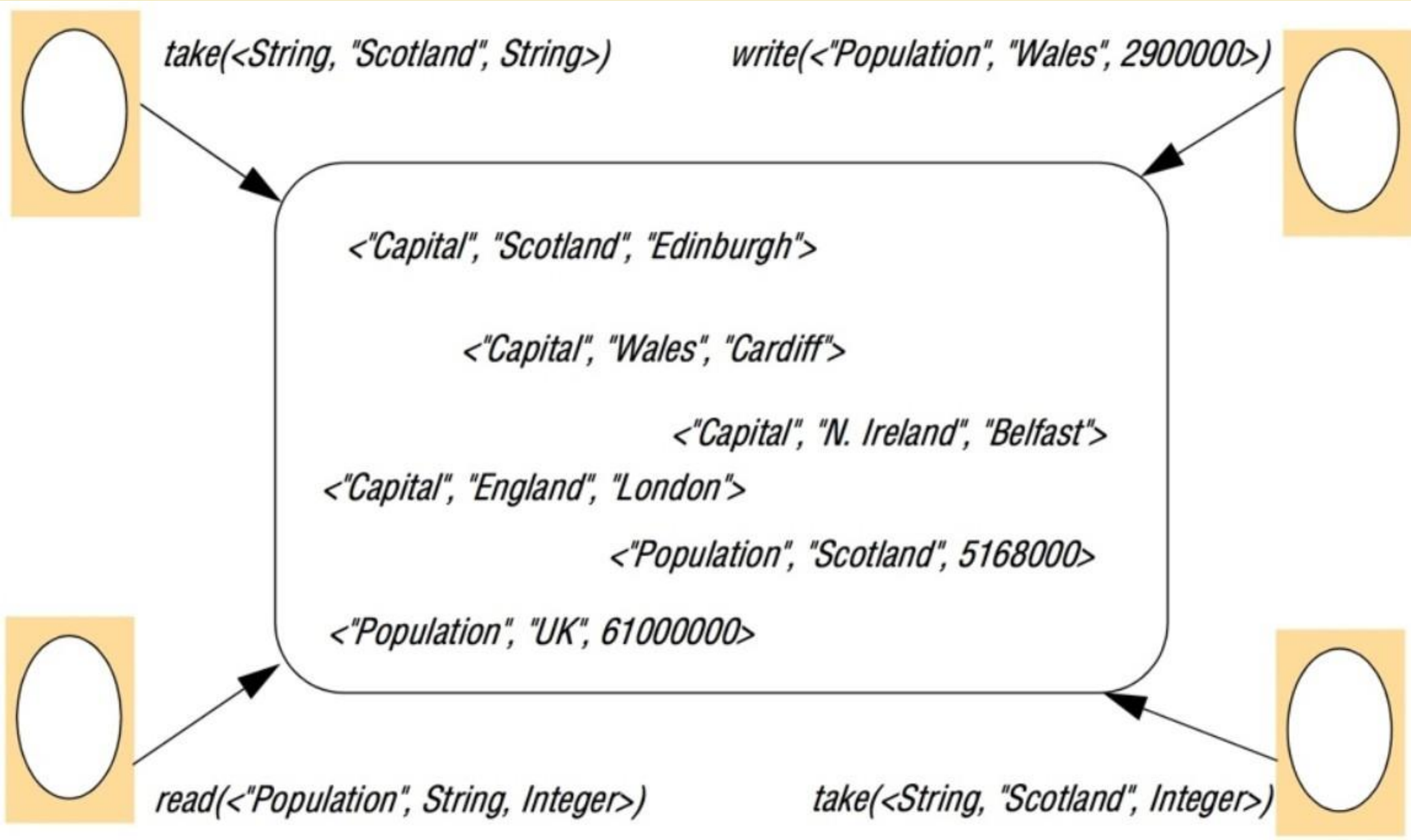
- * tuples are immutable
- – Tuple space (TS)
  - * any combination of types of tuples may coexist in the same tuple space
  - * processes share date through it
    - · write operation
    - · read (or take) operation

      read – TS not affected

      take – returns tuple and removes it from TS

      both blocking operations until there is a matching tuple in TS
- – associative addressing – processes provide for read and take operation a specification – any tuple with a matching specification is returned
- – Linda programming model – Linda programming language

Figure 6.20 The tuple space abst



Properties associated with tuple spaces

- *Space uncoupling*:

– A tuple placed in tuple space may originate from any number of sender processes and may be delivered to any one of a number of potential recipients

– also referred to as *distributed naming* in Linda

- *Time uncoupling:*

– A tuple placed in tuple space will remain in that tuple space until removed (potentially indefinitely) ⇒ hence the sender and receiver do not need to overlap in time

a form of *distributed sharing* of shared variables via the tuple space Variations:

•multiple tuple spaces

•distributed implementation

- Bauhaus Linda:
  - modelling everything as (unordered) sets – that is, tuple spaces are sets of tuples and tuples are sets of values, which may now also include tuples
- turning the tuple space into an *object space*
  - e.g. in JavaSpaces

Implementation issues

centralized vs distributed

- Replication or *state machine* approach (read more in textbook)
- peer-to-peer approaches

Case study: JavaSpaces

tool for tuple space communication developed by Sun

- Sun provides specification, third-party developers offer implementations:
  - GigaSpaces
  - Blitz
- strongly dependent on Jini (Sun's discovery service)
  - Jini Technology Starter Kit includes
    - Outrigger (JavaSpaces implementation)

goals of the JavaSpaces technology are:

- to offer a platform that simplifies the design of distributed applications and services

•to be simple and minimal in terms of the number and size of associated classes

•to have a small footprint

•to allow the code to run on resource-limited devices (such as smart phones)

•to enable replicated implementations of the specification

– (although in practice most implementations are centralized)

Programming with JavaSpaces

programmer can create any number of instances *space* – shared, persistant repository of objects

an item in JavaSpace – referred to as an *entry:* a group of objects contained in a class that implements net.jini.core.entry.Entry

Figure 6.23 The JavaSpaces

Operation	*Effect*
Lease write(Entry e, Transaction txn, long lease)	Places an entry into a particular JavaSpace
Entry read(Entry tmpl, Transaction txn, long timeout)	Returns a copy of an entry matching a specified template
Entry readIfExists(Entry tmpl, Transaction txn, long timeout)	As above, but not blocking
Entry take(Entry tmpl, Transaction txn, long timeout)	Retrieves (and removes) an entry matching a specified template
Entry takeIfExists(Entry tmpl, Transaction txn, long timeout)	As above, but not blocking
EventRegistration notify(Entry tmpl, Transaction txn, RemoteEventListener listen, long lease, MarshalledObject handback)	Notifies a process if a tuple matching a specified template is written to a JavaSpace

- placing an entry with write operation
  - entry can have an associated *lease*.
    - numerical value in milliseconds or Lease.FOREVER
  - write returns granted *Lease* value

302 Indirect communication 6.5

- read or take
  - matching specified by a *template*
  - matching entry has the same class or subclass
- notify
  - uses Jini distributed event notification
  - notification via a specified *RemoteEventListener* interface

operations in JavaSpaces can take place in the context of a transaction, ensuring that either all or none of the operations will be executed

Figure 6.24 Java class AlarmTupleJS

```
import net.jini.core.entry.*;
public class AlarmTupleJS implements Entry {
    public String alarmType;
    public AlarmTupleJS() {
    }
    public AlarmTupleJS(String alarmType) {
        this.alarmType = alarmType;
    }
}
```

Figure 6.25 Java class FireAlarmJS

```
import net.jini.space.JavaSpace;
public class FireAlarmJS {
    public void raise() {
        try {
            JavaSpace space = SpaceAccessor.findSpace("AlarmSpace");
            AlarmTupleJS tuple = new AlarmTupleJS("Fire!");
            space.write(tuple, null, 60*60*1000);
        catch (Exception e) {
        }
    }
}
```

c

```
//thecodecanbecalledusing:
FireAlarmJSalarm = newFireAlarmJS();
alarm.raise();
```

Figure 6.26 Java class FireAlarmReceiverJS

```
importnet.jini.space.JavaSpace;
publicclassFireAlarmConsumerJS {
    publicStringawait () {
        try {
            JavaSpacespace = SpaceAccessor.findSpace();
            AlarmTupleJStemplate = newAlarmTupleJS ("Fire!");
            AlarmTupleJSrecvd = (AlarmTupleJS) space.read(template, null
            ,
                                                    Long.MAX_VALUE);
        }   returnrecvd.alarmType;
        catch (Exceptione) {
            returnnull;
        }
    }
}
```

```
//consumer:
FireAlarmConsumerJSalarm Call = newFireAlarmConsumerJS ()
;
Stringmsg = alarm Call . await() ;
System . out . println ("Alarm received: " + msg) ;
```

End of week 7

7 Operating systems support

End of week 8

8 Distributed objects and components

8.1 Introduction

Distributed object middleware

- *encapsulation* in object-based solutions – well suited to distributed programming
- *data abstraction* – clean separation between the specification of an object and its implementation ⇒ programmers to deal solely in terms of interfaces and not concern with implementation details
- ⇒ more dynamic and extensible solutions

Examples of distributed objects middleware: Java RMI and CORBA

Component-based middleware

– to overcome a number of limitations with distributed object middleware:

Implicit dependencies: Object interfaces do not describe what the implementation of an object depends on

Programming complexity: need to master many low-level details

Lack of separation of distribution concerns: Application developers need to con- sider details of security, failure handling and concurrency – largely similar from one application to another

No support for deployment: Object-based middleware provides little or no support for the deployment of (potentially complex) configurations of objects

8.2 Distributed objects

- DS started as client-server architecture
- with emergence of highly popular OO languages (C++, Java) the OO concept spreading to DS
- Unified Modelling Language (UML) in SE has its role too in middleware devel- opments (e.g. CORBA and UML standards developed by the same organisation)

Distributed object (DO) middleware

- Java RMI and CORBA – quite common
- but CORBA – language independent

in DO he term class is avoided – instead factory instantiating new objects from a given template

- in Smalltalk – implementational inheritance
- in DO – interface inheritance:
  - new interface inherits the method signatures of the original interface
    - + can add extra ones

Distributed objects and components 8.2

Figure 8.1 Distributed obj

Objects	*Distributed objects*	*Description of distributed object*
Object references	Remote object references	Globally unique reference for a distributed object; may be passed as a parameter.
Interfaces	Remote interfaces	Provides an abstract specification of the methods that can be invoked on the remote object; specified using an interface definition language (IDL).
Actions	Distributed actions	Initiated by a method invocation, potentially resulting in invocation chains; remote invocations use RMI.
Exceptions	Distributed exceptions	Additional exceptions generated from the distributed nature of the system, including message loss or process failure.
Garbage collection	Distributed garbage collection	Extended scheme to ensure that an object will continue to exist if at least one object reference or remote object reference exists for that object, otherwise, it should be removed. Requires a distributed garbage collection algorithm.

istribut
d
bjects

Distributed objects OO: objects + class + inheritance ⟷ DO: encapsulation + data abstraction + design methodologies

The added complexities with DO:

- *Inter-object communication*
  - remote method invocation
  - + often other communications paradigms
    - (e.g. CORBA's event service + associated notification service)
- *Lifecycle management*
  - creation, migration and deletion of DO

- *Activation and deactivation*
  - # DOs may be very large...
  - node availabilities
- *Persistence*

  state of DO need to be preserved across all cycles (like [de]activation, system failures etc.)
- *Additional services*
  - e.g. naming, security and transaction services

8.3 Case study: CORBA

Object Management Group (OMG)

•formed in 1989

•designed an interface language

– independent of any specific implementation language

object request broker (ORB)

– to help a client to invoke a method on an object

Common Object Request Broker Architecture (CORBA)

CORBA 2 specification

CORBA3 – introduction of a component model

8.3.1 CORBA RMI

CORBA's object model

CORBA objects refer to remote objects

wide range of types PL support ⇒ no classes ⇒instances of classes cannot be passed as arguments

CORBA IDL

Figure 8.2 IDL interfaces *Shape* and *ShapeList*

```
struct Rectangle {   //1
    long width ;
    long height ;
    long x ;
    long y ;
};
struct GraphicalObject {   //2
    string type ;
    Rectangle enclosing ;
```

```
    booleanisFilled ;
};
interfaceShape { //3
    longgetVersion () ;
    GraphicalObjectgetAllState () ; //returnsstateoftheGraphicalObject
};
typedefsequence <Shape, 100> All; //4
interfaceShapeList { //5
    exceptionFullException { }; //6
    ShapenewShape (inGraphicalObjectg ) raises (FullException); /
    /7 AllallShapes (); //returnssequenceofremoteobjectreferences/ /8
    longgetVersion () ;
};
```

- same lexical rules as C++
    - + distribution keywords
        - – e.g. interface, any, attribute, in, out, inout, readonly, raises
- grammar of IDL – subset of ANSI C++ + constructs to support method signatures

IDL modules:

module defines a naming scope

Figure 8.3 IDL module *Whiteboard*

```
module Whiteboard {
    struct Rectangle {
    ...} ;
    struct GraphicalObject {
    ...};
    interface Shape {
    ...};
    typedef sequence <Shape, 100> All;
    interface ShapeList {
    ...};
};
```

IDL interfaces

- *IDL interface* describes the methods that are available in CORBA objects that implement that interface

•Clients of a CORBA object may be developed just from the knowledge of its IDL interface

IDL methods

The general form of a method signature is:

```
[oneway] <return_type> <method_name> (parameter1,...,
parameterL)  [raises (except1,..., exceptN)] [context
(name1,..., nameM)];
```

Example:

```
void getPerson(in string name, out Person p);
```

•parameters: in, out, inout

•return value acting as if additional out parameter

– return type may be void

Passing CORBA objects:

•Any parameter specified by the name of an IDL interface – a reference to a CORBA object

– the value of a remote object reference is passed

Passing CORBA primitive and constructed types:

•Arguments of primitive and constructed types are copied and passed by

value Invocation semantics

remote invocation call semantics defaults to: *at-most-once*

•to specify method invocation with *maybe semantics*: keyword oneway

– non-blocking call on the client side

– ⇒ method should not return a result

Exceptions in CORBA IDL

- optional *raises* expression indicates user-defined exceptions

- exceptions may be defined to contain variables, e.g:

  - exception **FullException** {**GraphicalObjectg** };

IDL types:

- 15 primitive types, *const* keyword

- object - remote object references

Figure 8.4 IDL constructed types

Case study: CORBA

Type	Examples	Use
sequence	*typedef sequence <Shape, 100> All;* *typedef sequence <Shape> All;* Bounded and unbounded sequences of Shapes	Defines a type for a variable-length sequence of elements of a specified IDL type. An upper bound on the length may be specified.
string	*string name;* *typedef string<8> SmallString;* Unbounded and bounded sequences of characters	Defines a sequence of characters, terminated by the null character. An upper bound on the length may be specified.
array	*typedef octet uniqueId[12];* *typedef GraphicalObject GO[10][8];*	Defines a type for a multi-dimensional fixed-length sequence of elements of a specified IDL type.

Type	Examples	Use
record	*struct GraphicalObject {* *string type; Rectangle enclosing; boolean isFilled;* *};*	Defines a type for a record containing a group of related entities.
enumerated	enum Rand (Exp, Number, Name);	The enumerated type in IDL maps a type name onto a small set of integer values.
union	*union Exp switch (Rand) {* *case Exp: string vote; case Number: long n; case Name: string s;* *};*	The IDL discriminated union allows one of a given set of types to be passed as an argument. The header is parameterized by an enum, which specifies which member is in use.

•All arrays or sequences used as arguments must be defined in typedefs

- None of the primitive or constructed data types can contain references

- passing non-CORBA objects (nCO) by value – CORBA's *valuetype*
  - nCO operations cannot be invoked remotely
  - makes it possible to pass a copy of a nCO between client and server

- *valuetype* – struct with additional method signatures (like those of an interface)

- *valuetype* arguments and results – passed by value
  - the state is passed to the remote site and used to produce a new object at the destination
  - if the client and server are both implemented in Java, the code can be downloaded
  - common C++ implementation – the necessary code to be present at

Attributes

IDL interfaces can have methods and **attributes**

- like public class fields in Java
- may be readonly
- private to CORBA objects
  - pair of attribute value set-retrieve generated by IDL compiler automatically

Inheritance

IDL interfaces may be extended through interface inheritance Example:

- interface B extends interface A ⇒

– B may add new types, constants, exceptions, methods and attributes to those of A

- + can redefine types, constants and exceptions
- not allowed to redefine methods

IDL interface may extend more than one interface

```
interface A { };
interface B: A{ };
interface C {};
interface Z : B, C {};
```

(but inheriting common names from two different interfaces not alowed)

<u>IDL type identifiers</u>

- generated by the IDL compiler

IDL:Whiteboard/Shape:1.0

- has three parts – the IDL prefix, a type name and a version number
- programmers have to provide a unique mapping to the interfaces – may use
  pragma prefix for this

IDL pragma directives

- for specification of additional non-IDL properties in IDL

interface for example,

- specifying that an interface will be used only locally
- supplying the value of an interface repository

ID Example:

```
#pragma version Whiteboard 2.3
```

CORBA language mappings

primitive types in IDL ⟶ corresponding primitive types in that language

structs, enums, unions ⟶ Java classes

IDL allows to have multiple return values... can be solved like this:

```
void getPerson(in string name, out Person p);
//IDL  void getPerson(String name, PersonHolder
p); //java
```

Asynchronous RMI

CORBA RMI allows clients to make non-blocking invocation requests on CORBA objects

- intended to be implemented in the client - server unaware on invocation synchronous or asynchronous (except e.g. Transaction Service)

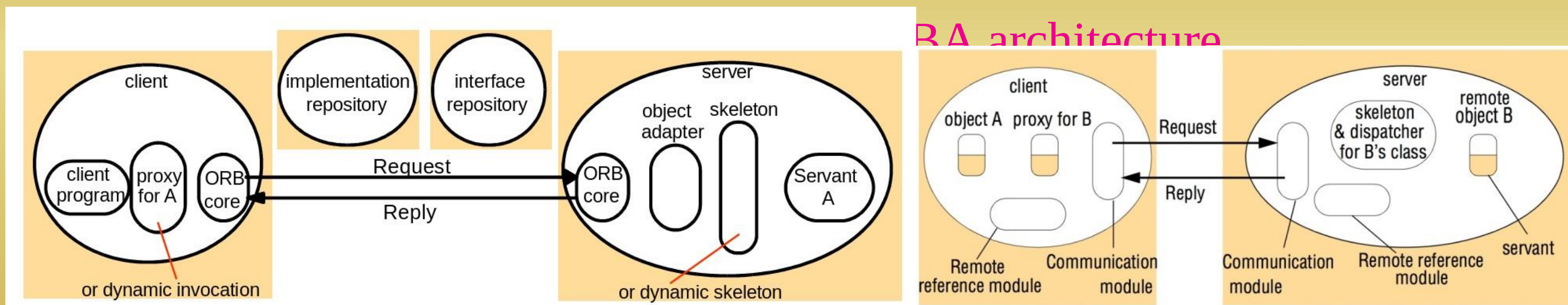
Asynchronous RMI invocation semantics:

- callback – client passes an extra parameter with a reference to a callback

⇒ server can call back with the results

- polling – server returns a valuetype object that can be used to poll or wait for the reply

8.3.2 The architecture of CORBA



- 3 additional components compared to Figure 5.15 (at right)...

- object adaptor; implementation repository; interface repository

a) Static invocation – object interfaces known at compile time – skeleton can be used

b) Dynamic invocation

ORB core

– role of [Fig. 5.15 communication module] + an interface which includes the following:

- operations enabling it to be started and stopped
- operations to convert between remote object references and strings
- operations to provide argument lists for requests using dynamic invocation

Object adapter (OA)

– role of [Fig. 5.15 reference and dispatcher modules]

CORBA objects with IDL interfaces ⟷ the programming language interfaces of the corresponding servant classes

OA tasks:

- creates remote object references for CORBA objects (Section 8.3.3)

•dispatches each RMI via a skeleton to the appropriate servant

•activates and deactivates servants

-gives each CORBA object a unique object name (forms part of its remote object reference)

- keeps a remote object table that maps the names of CORBA objects to their servants
- also has its own name (forms part of the remote object references of all of the CORBA objects it manages)

Portable Object Adapter (POA)

allows applications and servants to be run on ORBs produced by different developers supports CORBA objects with two different sorts of lifetimes:

•those whose lifetimes are restricted to that of the process in which their servants are instantiated (transient object references)

...for further details the textbook Section 8.3.2...

•those whose lifetimes can span the instantiations of servants in multiple pro- cesses (resistant object references)

Skeletons

Chapter 5.4.2:

Skeleton: implements the methods in the remote interface

- unmarshals the arguments in the request message
- invokes the corresponding method in the servant
- waits for the invocation to complete
- marshals the result (together with any exceptions in a reply message to the sending proxy's method)

Client stubs/proxies

The class of a proxy (for OO languages) or a set of stub procedures (for procedural languages) is generated from an IDL interface by an IDL compiler for the client language

Implementation repository

– responsible for:

- activating registered servers on demand
- locating servers that are currently running
- stores a mapping from the names of object adapters to the pathnames of files containing object implementations

When object implementations are activated in servers, the hostname and port number of the server are added to the mapping

object adapter name	pathname of object implementation	hostname and port number of server

Some objects (e.g. callback) created by clients, run once and cease to exist when they are no longer needed – do not use the implementation repository

Interface repository

– information about registered IDL interfaces to clients and servers that require it

- adds a facility for reflection to CORBA

Every CORBA remote object reference includes a slot that contains the type identi- fier of its interface, enabling clients that hold it to enquire its type of the interface repository

- applications using static (ordinary) invocation with client proxies and IDL skele- tons do not require an interface repository

- Not all ORBs provide an interface repository

SRM UNIVERSITY (Under section 3 of UGC Act 1956)

Dynamic invocation interface

CORBA does not allow classes for proxies to be downloaded at runtime (as in Java RMI) – The dynamic invocation interface is CORBA's alternative

- used when it is not practical to employ proxies
- The client can obtain from the interface repository the necessary information about the methods available for a given CORBA object
- The client may use this information to construct an invocation with suitable arguments and send it to the server

Dynamic skeletons

- Consider CORBA object whose interface was unknown when the server was compiled

with dynamic skeletons, server can accept invocations on the interface of a CORBA object for which it has no skeleton

- When a dynamic skeleton receives an invocation, it inspects the request contents to discover its
  - target object
  - the method to be invoked
  - the arguments
  - then invokes the target

Legacy code

- The term legacy code refers to existing code that was not designed with dis- tributed objects in mind

A piece of legacy code may be made into a CORBA object by defining an IDL inter- face for it and providing an implementation of an appropriate object adapter and the necessary skeletons

3. CORBA remote object reference

called: **interoperable object references (IORs)**

IOR format

1. IDL interface type ID2. Protocol and address details3. Object key

interface repository identifier type	IIOP	host domain name	port number	adapter name	object name

1. Note that IDL interface type ID is also identifier for the ORB interface reposi- tory (if it is existing)

2. Transport protocol: Internet InterORB protocol (IIOP) – uses TCP May be repeated to allow possible replications

3.Used by ORB to identify a CORBA object

Transient IOR last only as long as the process that hosts object

Persistant IOR last between activations of the CORBA objects

Case study: CORBA

8.3.4 CORBA services

specification of commen services includes in CORBA:

CORBA Service	Role
Naming service	Supports naming in CORBA, in particular mapping names to remote object references within a given naming context (see Chapter 9).
Trading service	Whereas the Naming service allows objects to be located by name, the Trading service allows them to be located by attribute; that is, it is a directory service. The underlying database manages a mapping of service types and associated attributes onto remote object references.
Event service	Allows objects of interest to communicate notifications to subscribers using ordinary CORBA remote method invocations (see Chapter 6 for more on event services generally).

SRM UNIVERSITY
(Under section 3 of UGC Act 1956)

Case study: CORBA

CORBA Service	Role
Notification service	Extends the event service with added capabilities including the ability to define filters expressing events of interest and also to define the reliability and ordering properties of the underlying event channel.
Security service	Supports a range of security mechanisms including authentication, access control, secure communication, auditing and nonrepudiation (see Chapter 11).
Transaction service	Supports the creation of both flat and nested transactions (as defined in Chapters 16 and 17).
Concurrency control service	Uses locks to apply concurrency control to the access of CORBA objects (may be used via the transaction service or as an independent service).

Case study: CORBA

CORBA Service	Role
Persistent state service	Offers a persistent object store for CORBA, used to save and restore the state of CORBA objects (implementations are retrieved from the implementation repository).
Lifecycle service	Defines conventions for creating, deleting, copying and moving CORBA objects; for example, how to use factories to create objects.

8.3.5 CORBA client and server example

compiler *idlj* generates the following items:

Figure 8.7 Java interfaces generated by *idlj* from CORBA interface *ShapeList*

•2 Java interfaces per IDL interface:

```
public interface ShapeListOperations {
    Shape newShape(GraphicalObject g) throws ShapeListPackage.FullException;
    Shape[] allShapes();
    int getVersion();
}

public interface ShapeList extends ShapeListOperations, org.omg.CORBA.Object,
    org.omg.CORBA.portable.IDLEntity { }
```

•server skeletons

– The names of skeleton classes end in POA – for example, ShapeListPOA

– The names of these classes end in Stub – for example, _ShapeListStub

•A Java class to correspond to each of the structs defined with the IDL interfaces

– In our example, classes Rectangle and GraphicalObject are generated.

– Each of these classes contains a declaration of one instance variable for each field in the corresponding struct and a pair of constructors, but no other methods.

•Classes called helpers and holders, one for each of the types defined in the IDL interface.

– A helper class contains the narrow method, which is used to cast down from a given object reference to the class to which it belongs, which is lower down the class hierarchy.

* For example, the narrow method in ShapeHelper casts down to

Server program

– The holder classes deal with out and inout arguments, which cannot be mapped directly onto Java.

CORBA objects – instances of servant classes.

When a server creates an instance of a servant class, it must register it with the POA (Portable Object Adaptor), which makes the instance into a CORBA object and gives it a remote object reference

Figure 8.8 *ShapeListServant* class of the Java server program for CORBA inter- face *ShapeList*

```
import org.omg.CORBA.*;
import org.omg.PortableServer.POA;
class ShapeListServant extends ShapeListPOA {
    private POA theRootpoa;
    private Shape theList[];
    private int version;
    private static int n=0;
    public ShapeListServant(POA rootpoa) {
```

```
        theRootpoa = rootpoa;
        // initialize the other instance variables
    }
    public Shape newShape(GraphicalObject g) throws ShapeListPackage.FullException { //1
        version++;
        Shape s = null;
        ShapeServant shapeRef = new ShapeServant( g, version);
        try {
            org.omg.CORBA.Object ref = theRootpoa.servant_to_reference(shapeRef); //2
            s = ShapeHelper.narrow(ref);
        } catch (Exception e) {}
        if(n >=100) throw new ShapeListPackage.FullException();
        theList[n++] = s;
        return s;
    }
    public Shape[] allShapes(){ ... }
    public int getVersion() { ... }
}
```

Main method in Server class:

Figure 8.9 Java class *ShapeListServer*

```
import org.omg.CosNaming.*;
import org.omg.CosNaming.NamingContextPackage.*;
```

```java
import org.omg.CORBA.*;
import org.omg.PortableServer.*;
public class ShapeListServer {
    public static void main(String args[]) {
        try{
            ORB orb = ORB.init(args, null); //1
            POA rootpoa = POAHelper.narrow(orb.resolve_initial_references("RootPOA")); //2
            rootpoa.the_POAManager().activate(); //3
            ShapeListServant SLSRef = new ShapeListServant(rootpoa); //4
            org.omg.CORBA.Object ref = rootpoa.servant_to_reference(SLSRef); //5
            ShapeList SLRef = ShapeListHelper.narrow(ref);
            org.omg.CORBA.Object objRef = orb.resolve_initial_references("NameService")
            ; NamingContext ncRef = NamingContextHelper.narrow(objRef); //6 NameComponent nc
            = new NameComponent("ShapeList", ""); //7
            NameComponent path[] = {nc}; //8
            ncRef.rebind(path, SLRef); //9
            orb.run(); //10
        } catch (Exception e) { ... }
    }
}
```

The client program

Figure 8.10 Java client program for CORBA interfaces *Shape* and *ShapeList*

```
import org.omg.CosNaming.*;
import org.omg.CosNaming.NamingContextPackage.*;
import org.omg.CORBA.*;
public class ShapeListClient {
    public static void main(String args[]) {
        try{
            ORB orb = ORB.init(args, null); //1
            org.omg.CORBA.Object objRef =
            orb.resolve_initial_references("NameService");
            NamingContext ncRef = NamingContextHelper.narrow(objRef);
            NameComponent nc = new NameComponent("ShapeList", "");
            NameComponent path [] = { nc };
            ShapeList shapeListRef =
            ShapeListHelper.narrow(ncRef.resolve(path)); //2
            Shape[] sList = shapeListRef.allShapes(); //3
            GraphicalObject g = sList[0].getAllState(); //4
        } catch(org.omg.CORBA.SystemException e) {...} //5
    }
}
```

Callbacks

Similar to JavaRMI

```
interface WhiteboardCallback {
    oneway void callback (in int version);
};
```

- implemented by client enabling the server to send version number whenever objects get added
- for this the ShapeList interface requires additional methods:

```
int register (in WhiteboardCallback callback);
void deregister (in int callbackId);
```

8.4 From objects to components

Component-based approaches – a natural evolution from distributed object computing

Issues with object-oriented middleware

Implicit dependencies – internal (encapsulated) behaviour of an object is hidden

– think remote method invocation or other communication paradigms... – not apparent from the interface

•there is a clear requirement to specify not only the interfaces offered by an object but also the dependencies that object has on other objects in the distributed configuration

Interaction with the middleware – too many relatively low-level details associated with the middleware architecture

•clear need to:

- simplify the programming of distributed applications
- to present a clean separation of concerns between code related to operation in a middleware framework and code associated with the application
- to allow the programmer to focus exclusively on the application code

Lack of separation of distribution concerns: Application developers need to deal explicitly with non-functional concerns related to issues such as security, trans- actions, coordination and replication – largely repeating concerns from one ap- plication to another

•the complexities of dealing with such services should be hidden wherever pos- sible from the programmer

No support for deployment: objects must be deployed manually on individual

→ component based middleware

Essence of components

- Middleware platforms should provide intrinsic ...re ...ies of deployment hidden from the user

> **software component** – unit of composition with contractually spec- ified interfaces and explicit context dependencies only

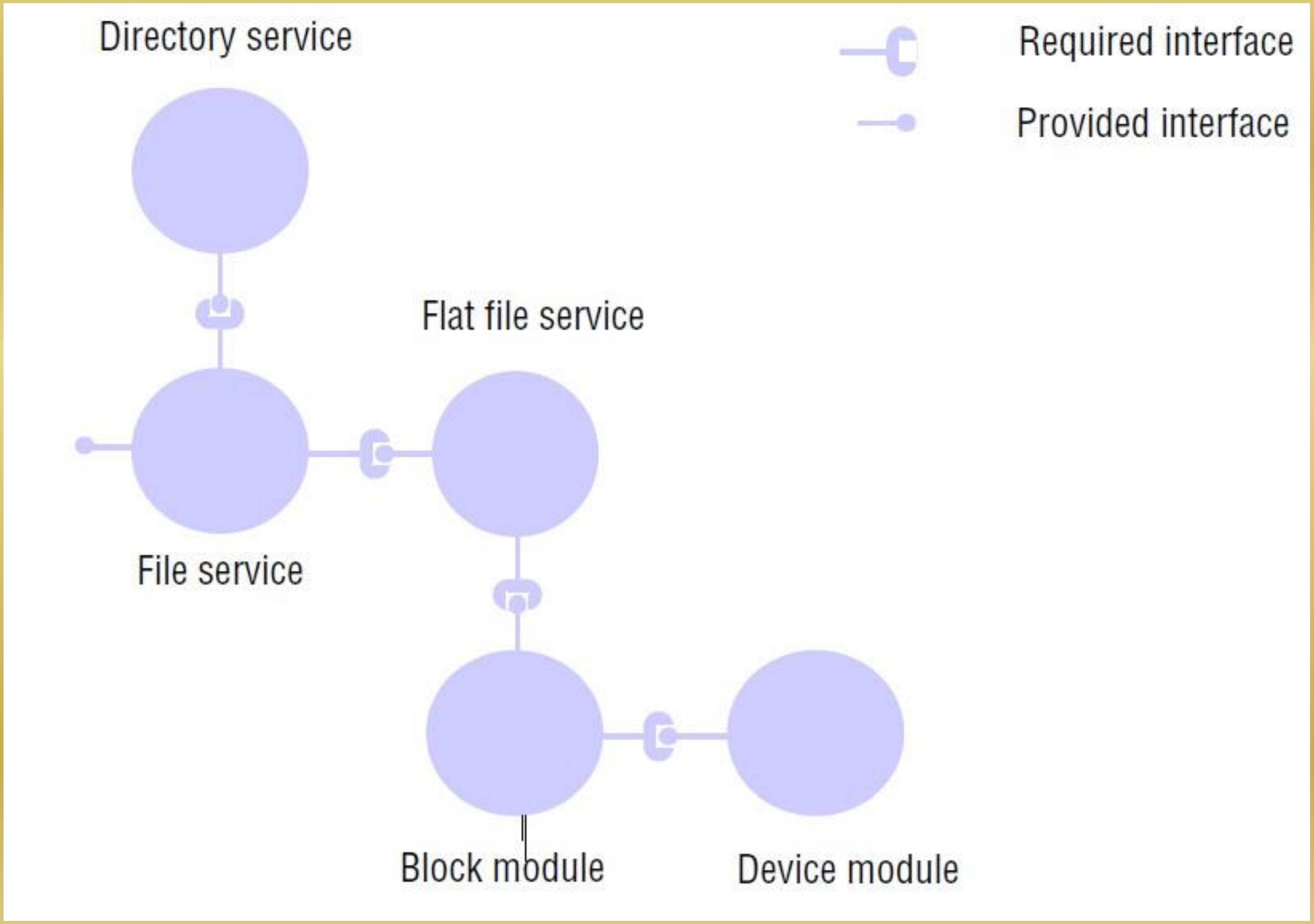
- dependencies are also represented as interfaces

SRM UNIVERSITY (Under section 3 of UGC Act 1956)

- component is specified in terms of a contract, which includes:
  - a set of provided interfaces
    - * – interfaces that the component offers as services to other components
  - a set of required interfaces
    - * – the dependencies that this component has in terms of other compo- nents that must be present and connected to this component for it to function correctly
- every required interface must be bound to a provided interface of another component
- →software architecture consisting of components, interfaces and connections between interfaces

Example: Architecture of a simple file system

Figure 8.11 An example software



From objects to components Many component-based approaches offer two styles of interface:

- interfaces supporting remote method invocation, as in CORBA and Java RMI
- interfaces supporting distributed events (as discussed in Chapter

6) Component-based system programming concerned with

- development of components
- composition of components

Moving from software development to software assembly

Components and distributed systems

Containers:

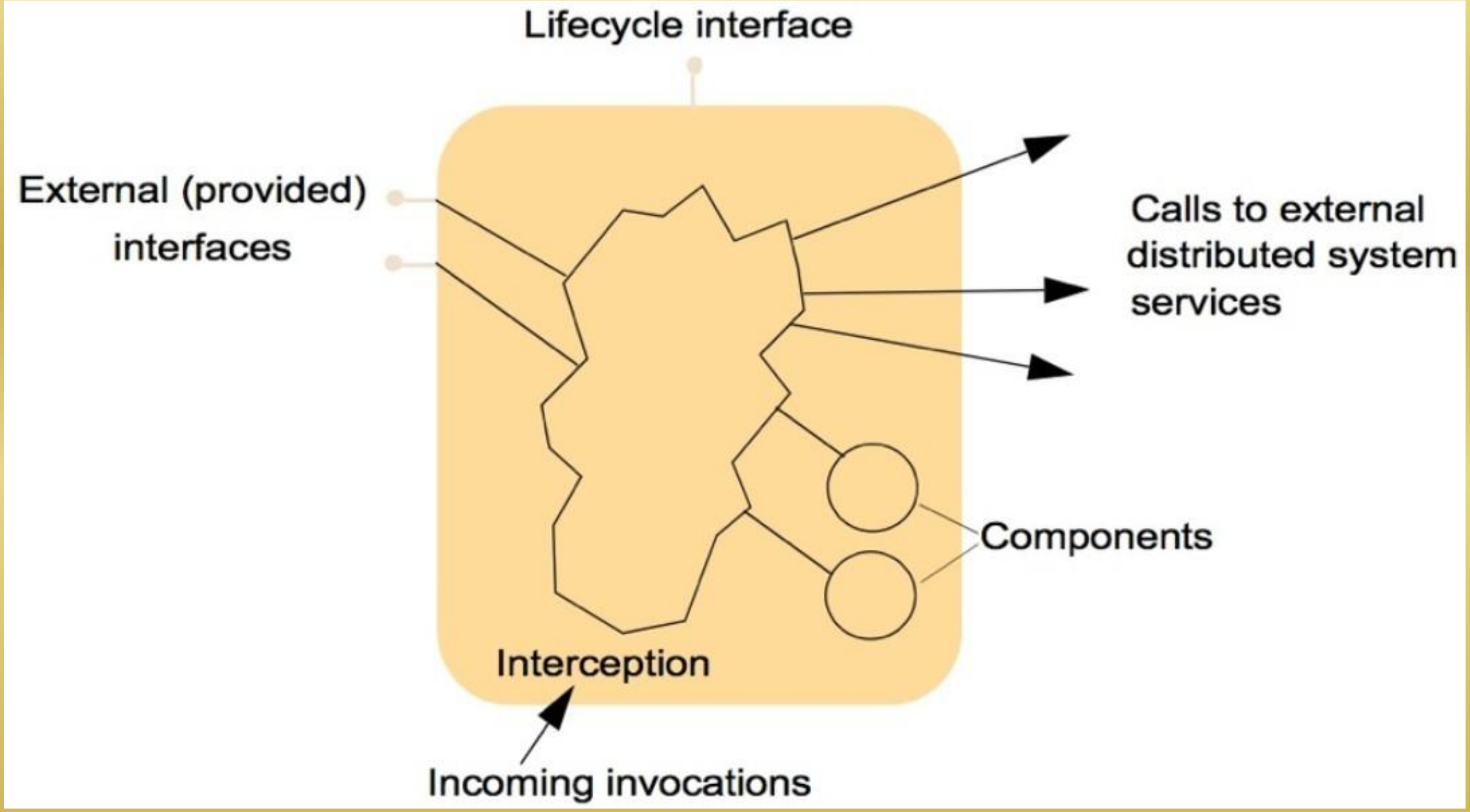
Containers support a common pattern often encountered in distributed applications, which consists of:

- a front-end (perhaps web-based) client
- a container holding one or more components that implement the application or business logic
- system services that manage the associated data in persistent

storage components deal with application concerns

container deals with distributed systems and middleware issues (ensuring that non-functional properties are achieved)

Figure 8.12 The structure of a



the container does not provide direct access to the components but rather intercepts incoming invocations and then takes appropriate actions to ensure the desired properties of the distributed application are maintained

SRM UNIVERSITY

components Middleware supporting the container pattern and the separation of concerns im-

plied by this pattern is known as an ***application server***

This style of distributed programming is in widespread use in industry today: –

ran

Technology	*Developed by*	*Further details*
WebSphere Application Server	IBM	www.ibm.com
Enterprise JavaBeans	SUN	java.sun.com
Spring Framework	SpringSource (a division of VMware)	www.springsource.org
JBoss	JBoss Community	www.jboss.org
CORBA Component Model	OMG	[Wang *et al.* 2001JOnAS]
JOnAS	OW2 Consortium	jonas.ow2.org
GlassFish	SUN	glassfish.dev.java.net

SRM UNIVERSITY (Under section 3 of UGC Act 1956)

Support for deployment

Component-based middleware provides support for the **deployment of component** ***configuration***

- components are deployed into containers
- deployment descriptors are interpreted by containers to establish the required policies for the underlying middleware and distributed system services

container therefore includes

- a number of components that require the same configuration in terms of distributed system support

Deployment descriptors are typically written in XML with sufficient information to ensure that:

- components are correctly connected using appropriate protocols and

- the associated distributed system services are set up to provide the right level of security, transaction support and so on
- the underlying middleware and platform are configured to provide the right level of support to the component configuration

8.5 Case study: Enterprise JavaBeans

End of week 9