Distributed Systems – CS249

Distributed Objects and Components

Agenda

- Distributed Objects
  - Introduction
  - Case Study:
    - CORBA Services
    - Java RMI
  - From Objects to Components: J2EE/EJB
- Web Services
- Peer-to-Peer Systems

Distributed Objects

1. Introduction

Distributed Objects

- **Middle-tier** in a multi-tier solution need to provide a high-level programming abstraction hiding the complexity of the underlying distributed infrastructure
- **Distributed objects** provides such abstraction while maintaining the benefits of the object-oriented approach to distributed programming
- **Two relevant abstraction are:** CORBA (Common Object Request Broker Architecture) and Java RMI (Remote Method Invocation)
- **Distributed objects programming** is more complex than traditional Object-Oriented programming. While class is fundamental concept in object-oriented, in distributed objects we care more about interfaces.

Distributed Objects

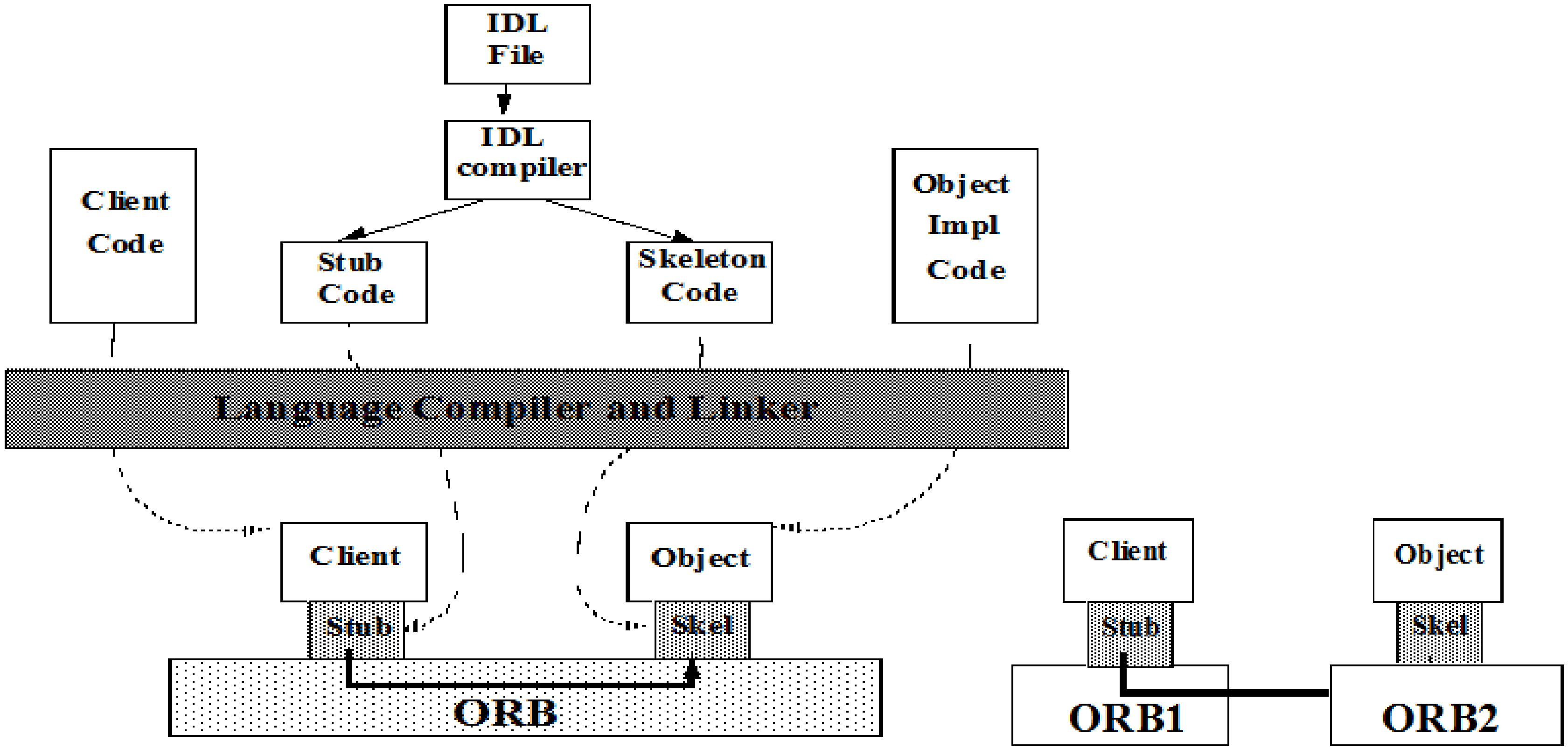
- **Distributed Object needed functionality:**
  - **Inter-object communication:** need to support remote method invocation capability
  - **Lifecycle management:** this is related to creation, migration, and deletion of objects
  - **Activation and deactivation:** as the number of objects is very large we cannot assume that all objects will be active all the time
  - **Persistence:** it is important to persist/maintain the object state across activation and deactivation operations
  - **Additional services:** most notable additional services needed in distributed object include: name service, additional security and transaction services

Distributed Objects

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.

2.1 Case Study: CORBA

Distributed Objects:
Case Study: CORBA overview

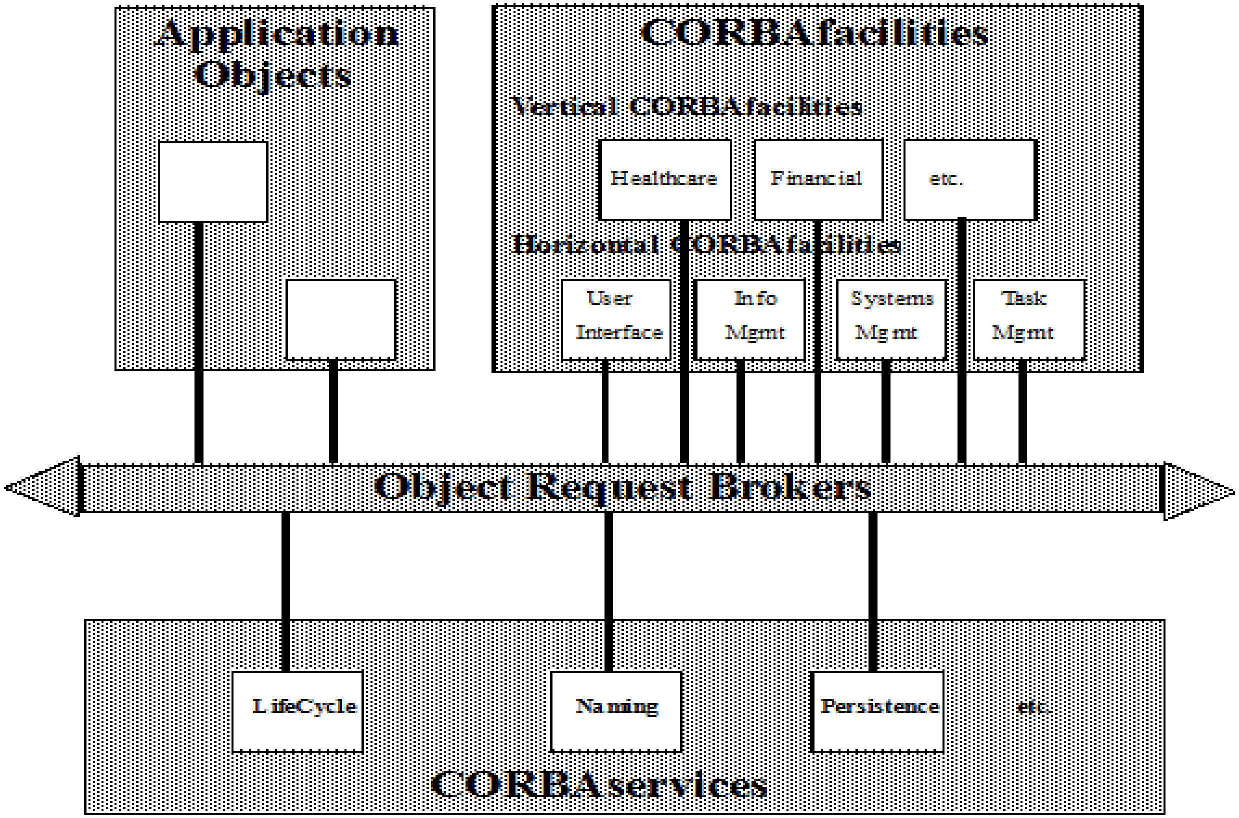


Client and Object Use Single ORB

Client and Object on Different ORBs

Distributed Objects:

Case Study: CORBA architecture



Object Management Architecture

Distributed Objects:
Case Study: CORBA – common object services

- **Naming Service:** allow objects to locate other components
- **Event Service:** allow objects to dynamically register/unregister their interest in a specific event
- **Life Cycle Service:** define operations for creating, copying, moving, and deleting objects
- **Object Transaction Service:** provides 2PC coordination between objects using either flat or nested transactions
- **Concurrency Control Service:** lock manager can be used by transactions/threads
- **Persistence Service:** an API to store the object state in different persistent stores including RDBMA, ODBMS, and files
- **Query Service:** superset of SQL based on SQL3 and ODMG/OQL
- **Externalization Service:** streaming object state In/Out service
- **Licensing Service:** charging per session, per node, per object instance, etc.

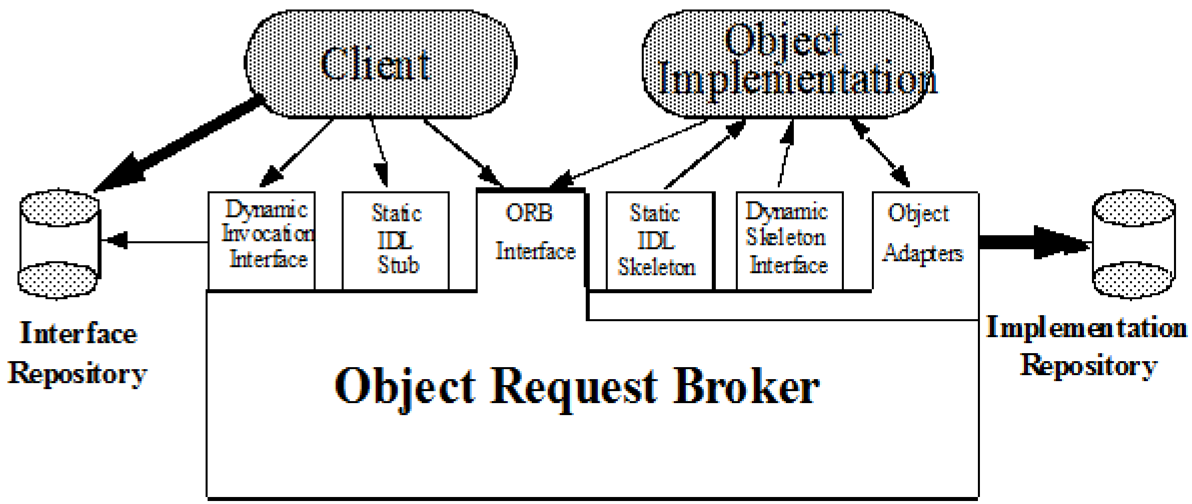
Distributed Objects:

Case Study: CORBA – common object services

- **Property Service:** associate name-value pair (property) with any object
- **Common Facilities:** It is collection of IDL-defined objects that provide services of direct use to application objects:
  - **Horizontal CORBA Facilities:** include user interface (UIs), information mgmt, system mgmt, and task mgmt such as workflow
  - **Vertical CORBA Facilities:** provide IDL defined interfaces for vertical market segments such as healthcare, retail, Telco, etc.

Distributed Objects:

Case Study: CORBA – the distributed object bus



- **CORBA** provides other services in addition to remote method invocation across heterogeneous languages, platforms and networks.
- **CORBA** provides both static and dynamic interfaces to its services

Distributed Objects:
Case Study: CORBA – the distributed object bus

- **I**n addition to supporting remote method invocation across heterogeneous languages, platforms and networks. CORBA provides both static and dynamic interfaces to its services
- **Client Static IDL stubs:** provides static interface to server objects. Client stub acts like a local call from the client perspective. Client stub perform marshaling. It also includes header files and enable invoking method on the server from high-level language like C, C++. Java, etc.
- **Client Dynamic IDL stubs**: let you discover at runtime the server interface (methods, parameters, exceptions,…) by looking up the metadata defined in the Interface Repository (IR), issuing the remote call and getting the result back.
- **Interface Repository (IR):** allows you to obtain/modify registered interfaces, methods they support, and parameters they require. IR is a runtime database that contains readable version of the IDL interfaces. An interface is defined in terms of global ID.

Distributed Objects:
Case Study: CORBA – the distributed object bus

- **Client/Server ORB Interface:** set of APIs to convert an object reference to a string and vice versa. Other operations that you can invoke on any object reference include: *get_interface*, *get_implementation*, and *is_nil* (to test if the object exists!).
- **Server Side:** can's tell the difference between client static or dynamic invocations; they both have the same message semantics. In both cases, the ORB locates an object adapter, transmits the parameters, and perform local method invocation on the real server object.
- **Server Static IDL Skeleton**: like the client stub generated by the IDL compiler. Skeleton provide static interfaces to services exported by the server to remote clients.
- **Server Dynamic Skeleton Interface (DSI):** provides runtime binding to server objects that need to handle incoming method invocations that don't have IDL-based compiled skeletons.

Distributed Objects:
Case Study: CORBA – the distributed object bus

- **Server Side Object Adapter:** sits on top of the ORB to accept request for services on behalf of the server's object. It provides runtime environment to instantiate the server object and pass requests to them, assign object an object-ID (object reference), and register the runtime object instance with the implementation Repository.
- **Server Side Implementation Repository (IR):** provides runtime repository of information about classes a sever support, objects that are instantiated, and their IDs.

Distributed Objects:
Case Study: CORBA – the distributed object bus

- **Interface Definition Language (IDL):** it describes the server object's attributes (public variables), super classes, exceptions it raises, typed events, programs for generating globally unique identifier, and methods supported including In/Out arguments. The IDL grammar is a subset of C++ with additional keywords, and it supports C++ preprocessing features.
- **ID: Structure:**

```
module  <identifier>                                   Define a NamingContext
{

        <type declarations>;
        <constant declarations>;
        <exception declarations>;
```

Distributed Objects:

Case Study: CORBA – the distributed object bus

```
interface   <identifier>   [:<inheritance>]          Defines a CORBA class
{

          <type declarations>;
          <constant declarations>;
          <exception declarations>;

          [<op_type>] <identifier> (<parameters>)    Defines a method
                [raises exception] [context];
          ....
          [<op_type>] <identifier> (<parameters>)    Defines a method
                [raises exception] [context];
          ....

}
```

Distributed Objects:
Case Study: CORBA – the distributed object bus

interface <identifier> [:<inheritance>] **Defines a CORBA class**

....

}

- **Module:** provides namespace to group a set of interfaces. A module has a scoped name that consists of one or more identifiers separated by the character "::".
- **Interface:** defines a set of methods that a client can invoke on an object. An interface may have attributes. These are values for which the implementation automatically creates *get*()/*set*() methods. An attribute can be declared read-only in which case the implementation provides the *get*() method only. An interface can be derived from one or more interfaces, i.e., multiple inheritance.

Distributed Objects:

Case Study: CORBA – the distributed object bus

- **Operation/Method:** in addition to defining signatures (in, out, inout), an optional *context* expression contains a set of attributes values that describe client context. It lets a client pass information to the server that describe local environment.
- **Data Types:** CORBA supports 2 categories: basic and constructed. Basic types include: short, long, unsigned long, unsigned short, float, double, char, Boolean, and octet. Constructed types include: enum, string, struct, array, union, sequence (variable size array), and any.
- Object Management Group (OMG) Home page: http://www.omg.org/
- OMG CORBA Download: http://www.corba.org/corbadownloads.htm
- The IDL compiler is called "idlj" and refer to this page for more details: https://docs.oracle.com/javase/7/docs/technotes/tools/share/idlj.html

Example

Distributed Objects:

Case Study: CORBA – the distributed object bus

```
module    MyAnimals
{
        interface    Dog  :Pet, Animal                  Defines class Dog
        {
                attribute integer age;
                exception NotInterested {string explanation};

                void Bark(in short how_long) raises (NotInterested);
                void Sit(in string where) raises (NotInterested);
                void Growl(in string at_whom) raises (NotInterested);
        }
        interface    Cat   :Animal                      Defines class Cat
        {
                void Eat();
                void HerKitty();
                void Bye();
        }
}
```

Distributed Objects:
IDL interfaces Shape and ShapeList

```
struct Rectangle{                          1
    long width;
    long height;
    long x;
    long y;
} ;

struct GraphicalObject {                   2
    string type;
    Rectangle enclosing;
    boolean isFilled;
};

interface Shape {                                                        3
    long getVersion() ;
    GraphicalObject getAllState() ;        // returns state of the GraphicalObject
};

typedef sequence <Shape, 100> All;                                       4
interface  ShapeList {                                                   5
    exception FullException{ };                                          6
    Shape newShape(in GraphicalObject g) raises (FullException);         7
    All     allShapes();                   // returns sequence of remote object references 8
    long  getVersion() ;
};
```

Distributed Objects:
IDL module whiteboard

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

Distributed Objects:
IDL constructed types - 1

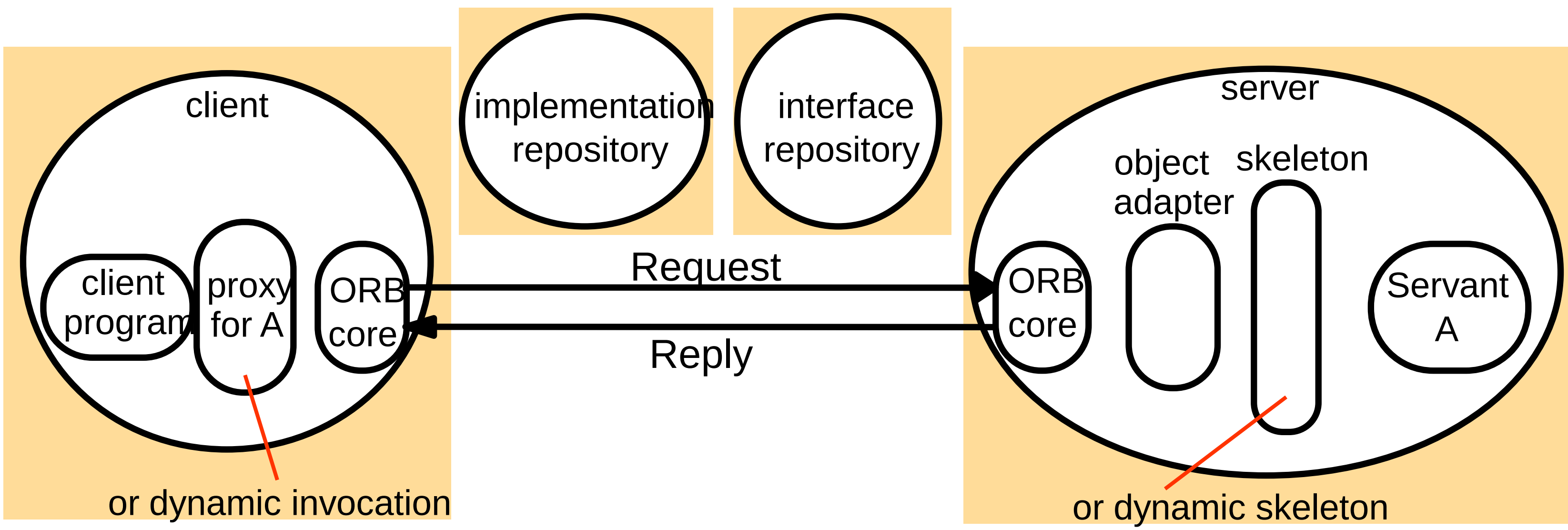
Type	*Examples*	*Use*
sequence	*typedef sequence <Shape, 100> All;* *typedef sequence <Shape> All* bounded and unbounded sequences of *Shapes*	Defines a type for a variable-length sequence of elements of a specified IDL type. An upper bound on the length may be specified.
string	*String name;* *typedef string<8> SmallString;* unbounded and bounded sequences of characters	Defines a sequences 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.

this figure continues on the next slide

Distributed Objects:
IDL constructed types - 2

Type	*Examples*	*Use*
record	*struct GraphicalObject { string type; Rectangle enclosing; Boolean isFilled; };*	Defines a type for a record containing a group of related entities. *Structs* are passed by value in arguments and results.
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.

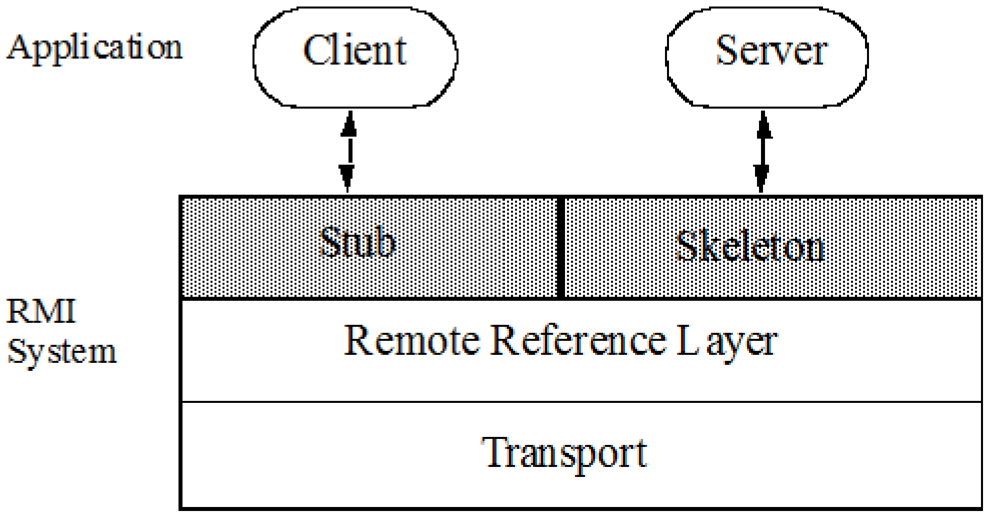
Distributed Objects: The main components of the CORBA architecture



2.2 Case Study: Java RMI

Distributed Objects:
Case Study: Java RMI overview

- **The Remote Method Invocation (RMI)** is designed to support remote method invocation on objects across VMs or across the network.
- RMI support object serialization – marshaling and unmarshaling
- RMI takes advantage of Java's ability to dynamically download byte code (client stub) across the network
- Stubs/Skeleton are generated by the RMI stub compiler (rmic)

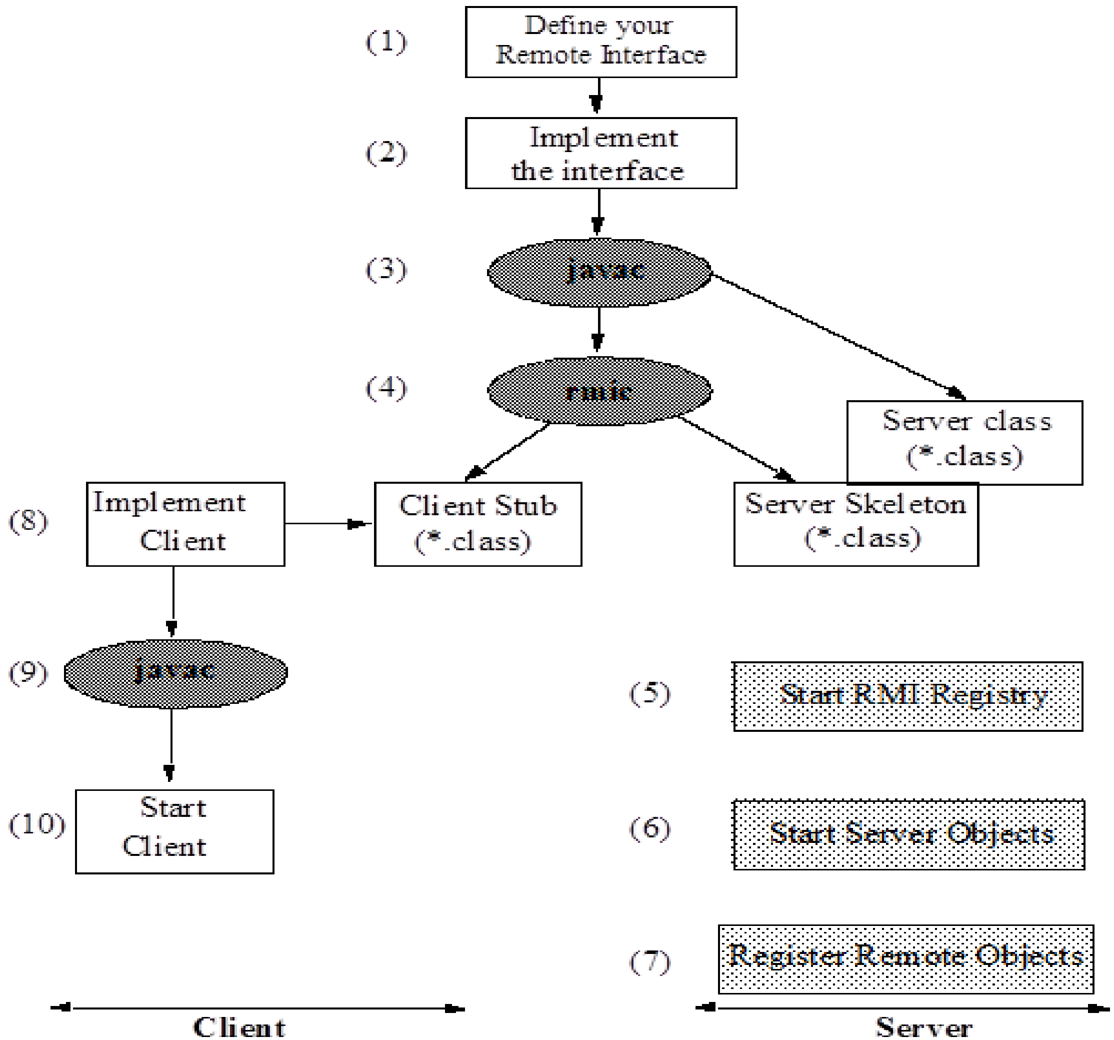


Distributed Objects:
Case Study: Java RMI overview

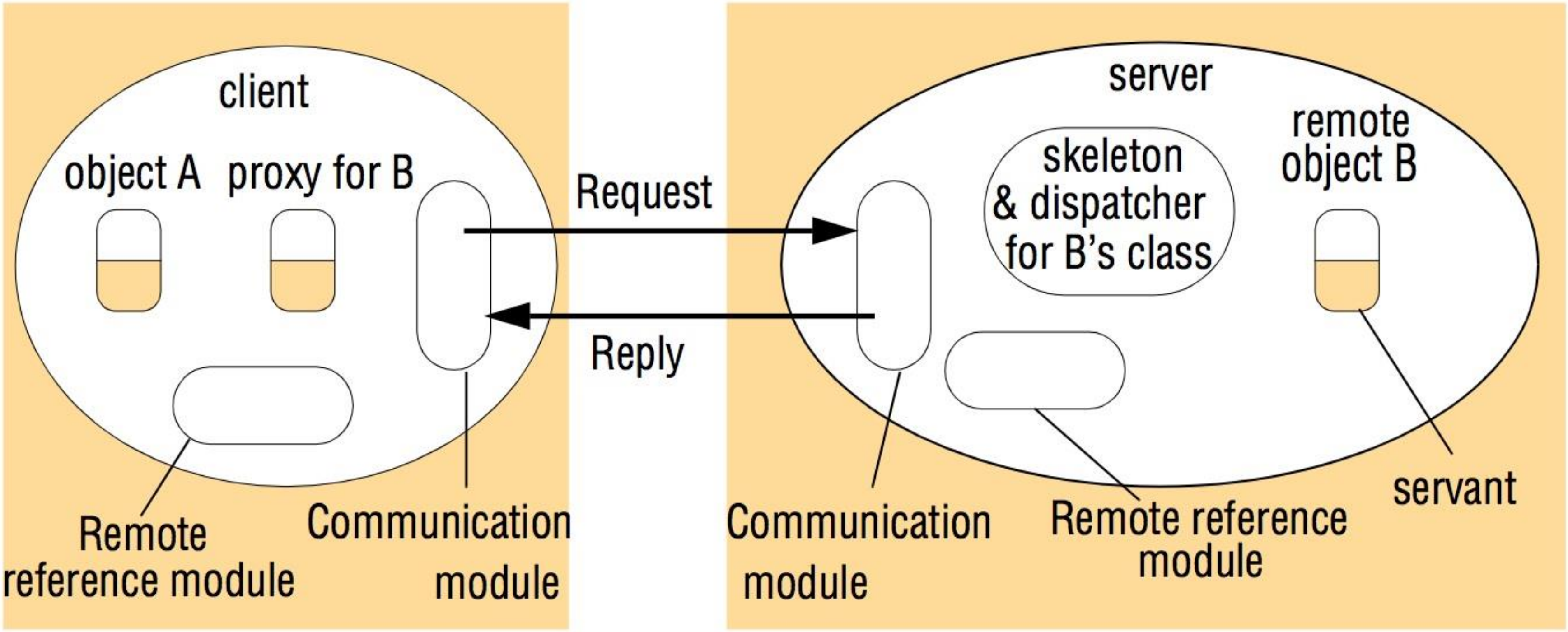
- **The Remote Reference layer** is responsible for carrying out the semantics of the invocation, i.e., whether the server is a single object or replicated object
- **The transport layer** is responsible for connection setup, connection management, and keeping track of the server object
- **Method calls** originating from the same client VM may execute by the same thread in the server object VM. Method calls originating from different client VMs will execute by different threads in the server object VM.

Distributed Objects:

Case Study: Java RMI development process



Distributed Objects:
Case Study: Java RMI architecture



Example

Remote Interfaces

```
package    class8;

import     java.rmi.*;
import     java.io.OutputStream;

public     interface  RMISolver     extends   java.rmi.Remote {
           public     boolean       solve ( )                                        throws RemoteException;
           public     boolean       solve (RMIProblemSet s, int numIters )           throws RemoteException;
           public     RMIProblemSet getProblem ( )                                   throws RemoteException;
           public     boolean       setProblem ( RMIProblemSet s )                   throws RemoteException;
           public     int           getIterations ( )                                throws RemoteException;
           public     int           setIterations ( int numIter )                    throws RemoteException;
}
```

```
package    class8;
import     java.rmi.*;

public     interface  RMIProblemSet  extends   java.rmi.Remote {
           public     double         getValue ( )                                    throws RemoteException;
           public     double         getSolution ( )                                 throws RemoteException;
           public     void           setValue ( double v )                           throws RemoteException;
           public     void           setSolution ( double s )                        throws RemoteException;
}
```

Server class must implement the remote interface and extends the class java.rmi.unicastRemoteObject

```
package     class8;

import      java.rmi.*;
import      java.rmi.server.UnicastRemoteObject;
import      java.io.*;

public      class       RMISolverImpl  extends   UnicastRemoteObject     implements  RMISolver {
            // Protected implementation variables.
            protected   int              numIterations = 1;   // not used for the Solver
            protected   RMIProblemSet    currProblem = null;

            // Constructors
            public      RMISolverImpl ( )  throws           RemoteException {
                        super ( );
            }

            public      RMISolverImpl ( int numIter )   throws     RemoteException {
                        super ( );
                        numIterations = numIter;
            }
```

```
// Public Methods.
public      boolean    solve ( )                  throws                 RemoteException {
            System.out.println ( "Solving current problem .... ");
            return solve ( curreProblem, numIterations );
}
public      boolean    solve (RMIProblemSet s, int numIters )    throws    RemoteException {
            boolean    success = true;
            if ( s == null ) {
                    System.out.println ( "No problem to solve!");
                    return false;
            }
            System.out.println ( "Problem value = " + s.getValue ( ) );
            numIterations = numIters;

            // Solve the problem here.
            try {
                    s.setSolution ( Math.sqrt ( s.getValue ( ) ) );
            }
            catch (ArithmaticException e ) {
                    System.out.println ( "Badly formed problem" ); success = false;
            }

            System.out.println ( "Problem solution = " + s.getSolution ( ) );
            return success;
}
```

```
public    RMIProblemSet    getProblem ( )    throws    RemoteException {
          return currProblem;
}

public    boolean   setProblem (RMIProblemSet s )      throws    RemoteException {
          currProblem = s;
          return true;
}

public    int       getIterations ( )                  throws    RemoteException {
          return numIterations;
}
public    boolean   setIterations ( int numIter )      throws    RemoteException {
          numIterations=numIter;
          return true;
}
```

```
        public      static void main ( String argv[] ) {
                    // Create and install a security manager
                    System.setSecurityManager ( new RMISecurityManager ( ) );

                    try {
                          // register an instance of RMISolverImpl with the RMI Naming Service
                          String name = "TheSolver";
                          RMISolverImpl solver = new RMISolverImpl ( );
                          Naming.rebind ( name, solver );
                          System.out.println ( "Remote solver is ready.. " );
                    }
                    catch ( Exception e ) {
                          System.out.println ( "Caught an exception while registering: + e );
                          e.printStackTrace ( );
                    }
        }
}
```

Server class must implement the remote interface and extends the class java.rmi.unicastRemoteObject

```
package     class8;

import      java.rmi.*;
import      java.rmi.server.UnicastRemoteObject;


public      class   RMIProblemSetImpl  extends   UnicastRemoteObject   implements  RMIProblemSet {
            // Protected implementation variables.

            protected   double          value;              // not used for the Solver
            protected   double          solution;

            // Constructors
            public      RMIProblemSetImpl  ( )     throws     RemoteException {
                        value = 0.0;
                        solution = 0.0;
            }


            public      double getValue ( )        throws     RemoteException {
                        return value;
            }
```

```
public    double getSolution ( )    throws    RemoteException {
          return solution;
}

public    double setValue (double v )  throws    RemoteException {
          value = v;
}

public    double setSolution (double s )  throws    RemoteException {
          solution = s;
}
}
```

- Compile your server class: using javac
- Run the stub compiler: using rmic against the className
- Rmic   className <cr>
- Start the RMI registry on your server: shell> rmic rmiregistry   <cr>
- Start your server objects
- Register your remote objects with the registry
- Write the client code…

Client class: use naming class to locate the remote object. Invoke methods via a stub (generated by rmic) that serves as a proxy for the remote object.

```
package    class8;

import     java.rmi.*;
import     java.rmi.server.*;

public     class   RMISolverClient {

    public     static void main ( String argv[] ) {
               // Create and install a security manager
               System.setSecurityManager ( new RMISecurityManager ( ) );
               // get a remote reference to the RMISolver class
               String name = "rmi://nest.us.oracle.com/TheSolver";
               RMISolverImpl solver = null;
```

```
try {
        solver = (RMISolver) Naming.lookup ( name );
}
catch ( Exception e ) {
        System.out.println ( "Caught an exception while  looking up the server:" );
         e.printStackTrace ( ); System.exit ( );
}

// make a problem set for the solver.
         RMIProblemSetImpl s = null;

try {
         s = new RMIProblemSetImpl ( );
         s.getValue ( Double.valueOf ( argv[0] ).doubleValue( ) );
}
catch ( Exception e ) {
        System.out.println ( "Caught exception initializing  problem:" );
         e.printStackTrace ( );
}

// Ask solver to solve.
```

```
// Ask solver to solve.
try {
        if ( solver.solves ( s, 1 ) ) {
              System.out.println ( “Solver returned solution: “ + s.getSolution ( ) );
        }
        else {
        System.out.println ( “Solver is unable to solve the problem with value = “ +
                                s.getValue ( ) );
        }
}
catch ( RemoteException e ) {
        System.out.println ( “Caught remote exception.” );
        System.exit (1);
}
    }
}
```

- Compile the client code: using javac
- Start the client: It is possible to download server classes from the server on demand

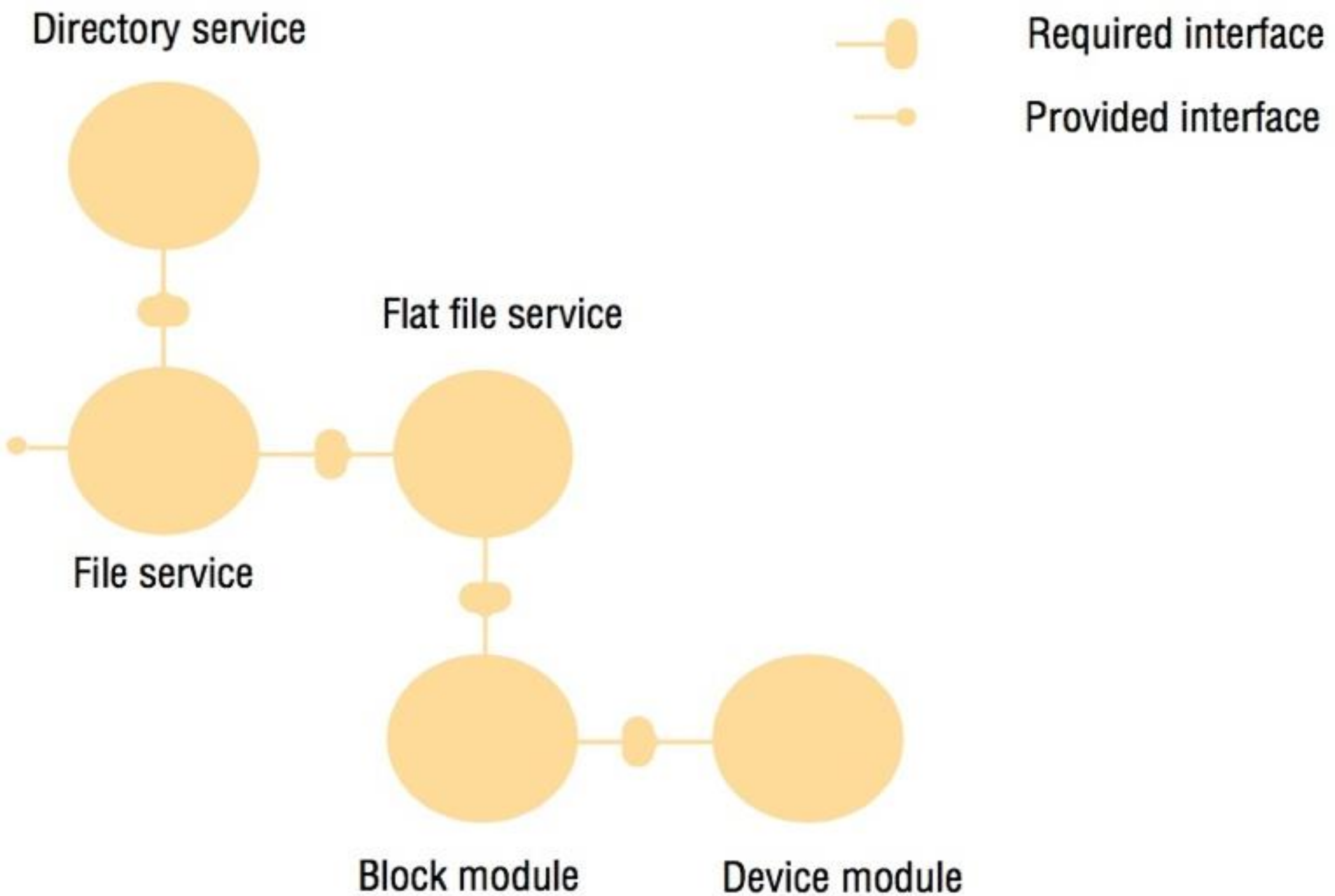
Distributed Objects:

Case Study: Java RMI Garbage Collection

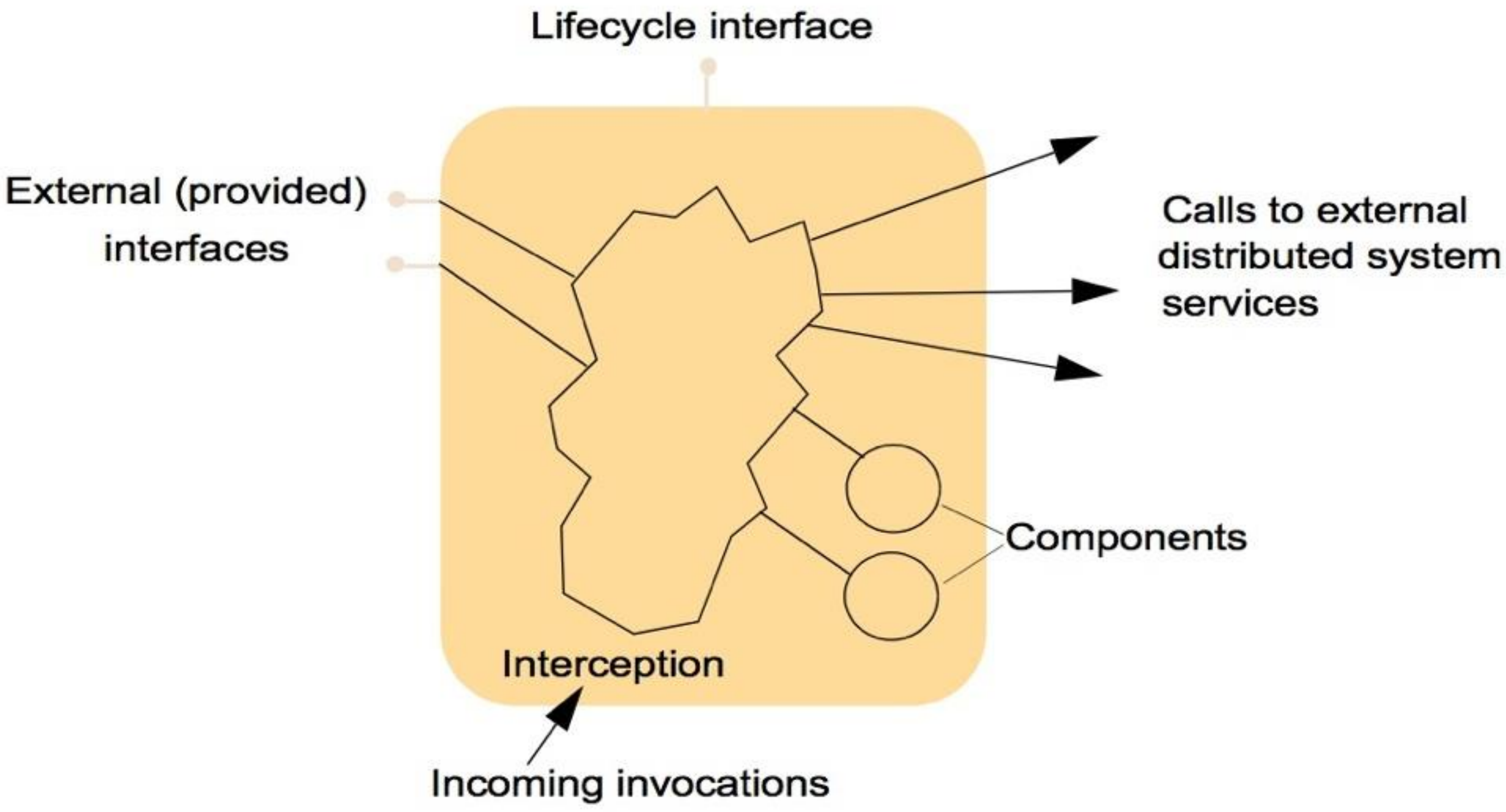
- **In a distributed system,** just as in local system, it is desirable to automatically delete those objects that are no longer referenced! This frees the programmer from needing to keep track of the distributed objects so that it can terminate appropriately
- RMI uses reference-counting garbage collection algorithm similar to Modula-3's Network objects.
- RMI runtime keep track of all live references within each Java VM. A live reference is just a client/server connection over a TCP/IP session.
- As long as a reference to a remote object exists, it cannot be garbage collected.
- You can pass a reference as an argument in remote calls. The object to which a reference is passed is added to a reference list. Every time a client receives a reference, the corresponding reference count is incremented by one. It is decremented by one when the client stops referencing the object. When reference count reaches zero, RMI puts the server object on the weak reference list, The Java garbage collector can now discard this object; calling the object **finalize**() method before deleting from memory

3. From Objects to Components

Distributed Objects:
An example software architecture



Distributed Objects:
The structure of a container



Distributed Objects:
Application servers

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

Case Study: EJB

Distributed Objects:
Case Study: Java Enterprise Edition (JEE)

- **Enterprise Java Beans (EJB)** provides a distributed component model that enables developers to focus on solving business problems while relying on the J2EE platform to handle complex system-level issues.
- **Business logic** decomposes a business function into a set of components or elements called business objects. Like other objects it has state and behavior/methods. Common requirements of business objects include:
  - Maintain state
  - Operate on shared data
  - Participate in transactions
  - Service large number of clients

Distributed Objects:
Case Study: Java Enterprise Edition (JEE)

- Remain available to clients
- Provide remote access to data
- Control access
- Reusable

- **EJB as JEE Business Objects:** EJB architecture defines components - called Enterprise Beans - that allow the developer to write business objects that use the services provided by the JEE platform. There are three kinds of EJB components: *Session beans*, *Entity beans*, and *Message-Driven beans*.
  - **Session Beans**: intended to provide resources used only by the client that created them. They are extension to the client on the server side.

Distributed Objects:

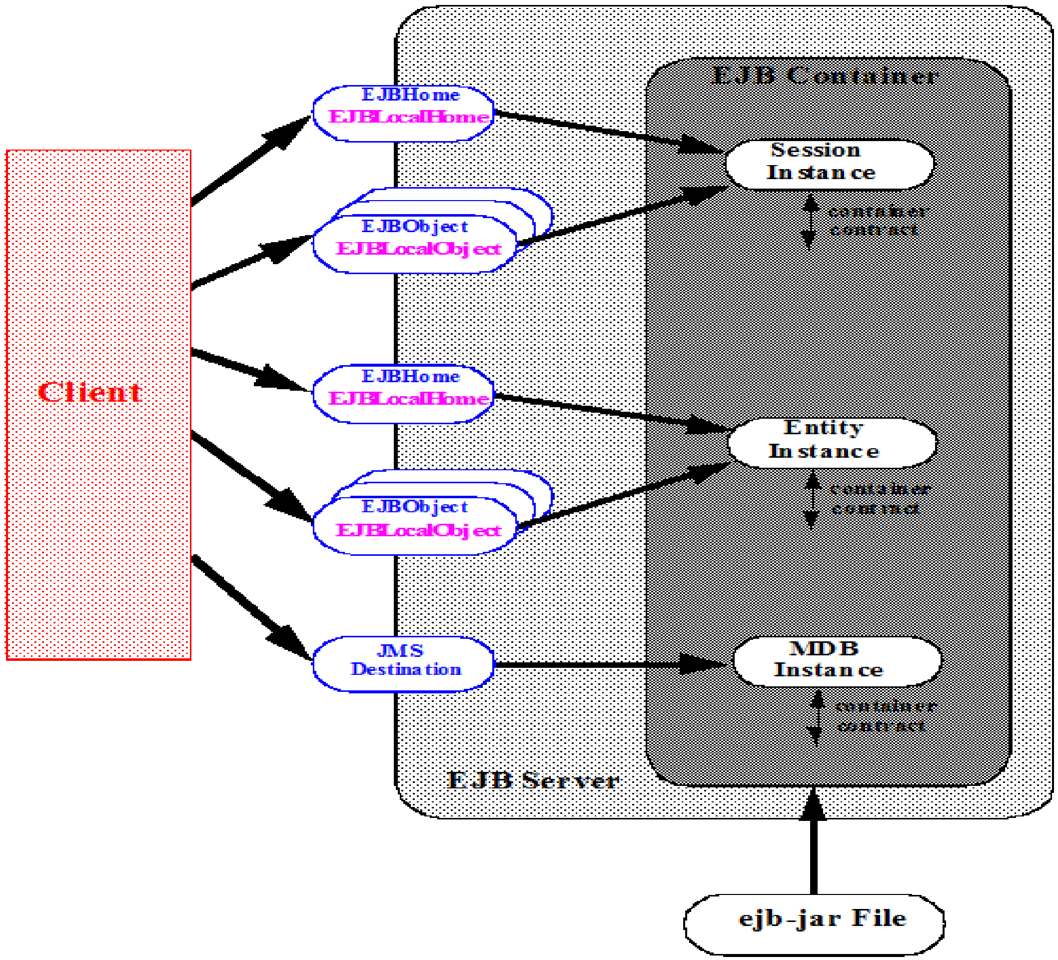
Case Study: Java Enterprise Edition (JEE)

- **Entity Beans**: represent an object view of some entity that is stored in persistent store, such as an RDBMS. In contrast to session beans, every entity bean has a unique identity that is exposed to clients as a primary key.
- **Message-Driven Beans**: new to EJB 2.0 architecture. Supported in JEE 1.3 platform. They are components that process asynchronous messages by implementing a JMS message listener interface. They consume/serve asynchronous messages sent to a JMS queue or topic.

- EJB components live inside EJB containers; which provide life cycle management, transaction, security, persistence, and a variety of other services for them.

Distributed Objects:
Case Study: Java Enterprise Edition (JEE)

- When a client invokes an operation on an EJB, the call is intercepted by its container, as a result container can inject services (such as start a transaction) that propagate across calls and components, and even across containers running on different servers and different machines



Client's View of Enterprise Beans

Distributed Objects:
Case Study: Transaction attributes in EJB

Attribute	*Policy*
REQUIRED	If the client has an associated transaction running, execute within this transaction; otherwise, start a new transaction.
REQUIRES_NEW	Always start a new transaction for this invocation.
SUPPORTS	If the client has an associated transaction, execute the method within the context of this transaction; if not, the call proceeds without any transaction support.
NOT_SUPPORTED	If the client calls the method from within a transaction, then this transaction is suspended before calling the method and resumed afterwards – that is, the invoked method is excluded from the transaction.
MANDATORY	The associated method must be called from within a client transaction; if not, an exception is thrown.
NEVER	The associated methods must not be called from within a client transaction; if this is attempted, an exception is thrown.

Distributed Objects:

Case Study: Invocation contexts in EJB

Signature	*Use*
public Object getTarget()	Returns the bean instance associated with the incoming invocation or event
public Method getMethod()	Returns the method being invoked
public Object[] getParameters()	Returns the set of parameters associated with the intercepted business method
public void setParameters(Object[] params)	Allows the parameter set to be altered by the interceptor, assuming type correctness is maintained
public Object proceed() throws Exception	Execution proceeds to next interceptor in the chain (if any) or the method that has been intercepted

Distributed Objects:
Case Study: Fractal

- **Fractal** is another component model. It is lightweight component model that brings the benefits of component-based programming to the development of distributed systems (http://fractal.ow2.org/)
- **Fractal** provides support for programming with interfaces, benefits of separating interface from implementation
- **Fractal** supports distributed objects
- **Fractal** is deliberately a minimal approach and does not support all component-based functionality such as deployment or full container pattern offered by application servers
- **Fractal** is simple and as a result, it is not only configurable but also reconfigurable at runtime to match the current operational environment and requirements

Distributed Objects:
Case Study: Fractal

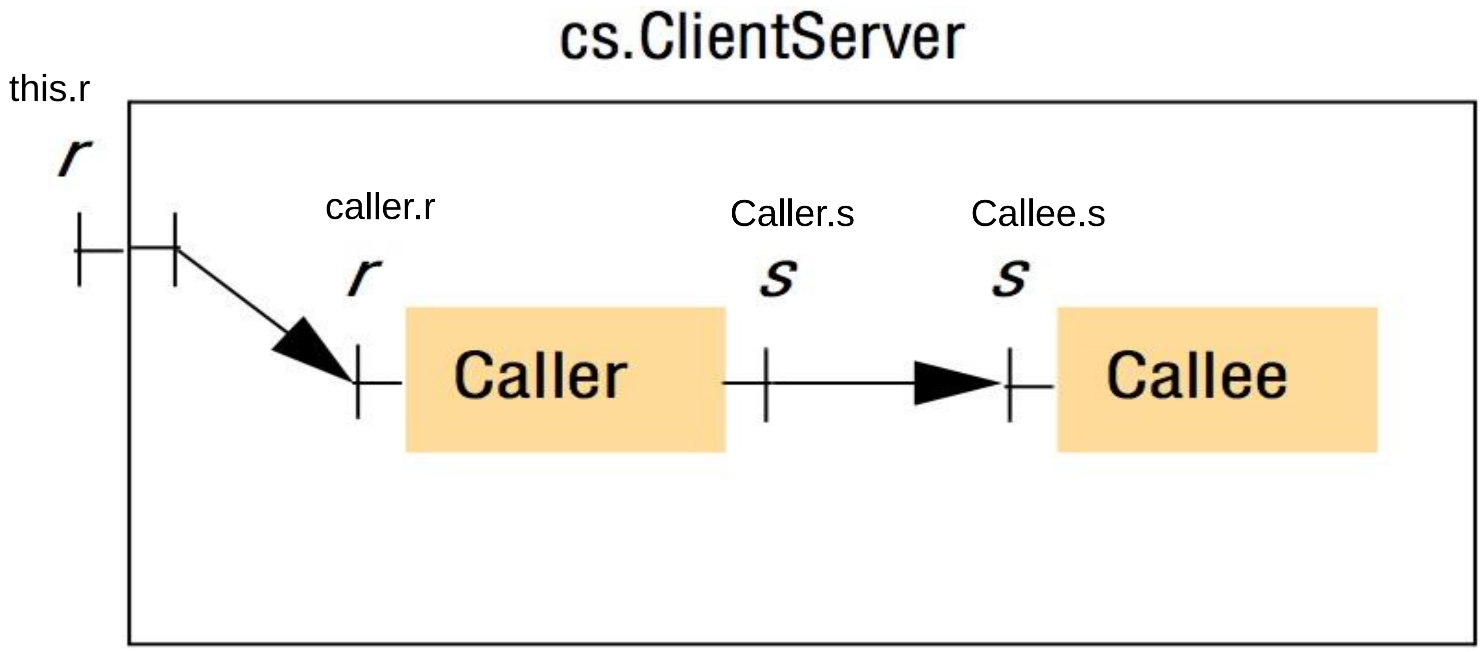
- **Fractal** defines a programming model and is programming language-agnostic.
- Implementations are available in several languages:
  - Julia and AOKell (Java based)
  - Cecilia and Think (C-based)
  - FracNet (.NET-based)
  - FracTalk (Smalltalk-based)
  - Julio (Python-based)
- **Julia** and **Cecilia** are treated as the reference implementation
- **Fractal** is supported by the OW2 consortium (https://www.ow2.org/)

Distributed Objects:

Case Study: Fractal

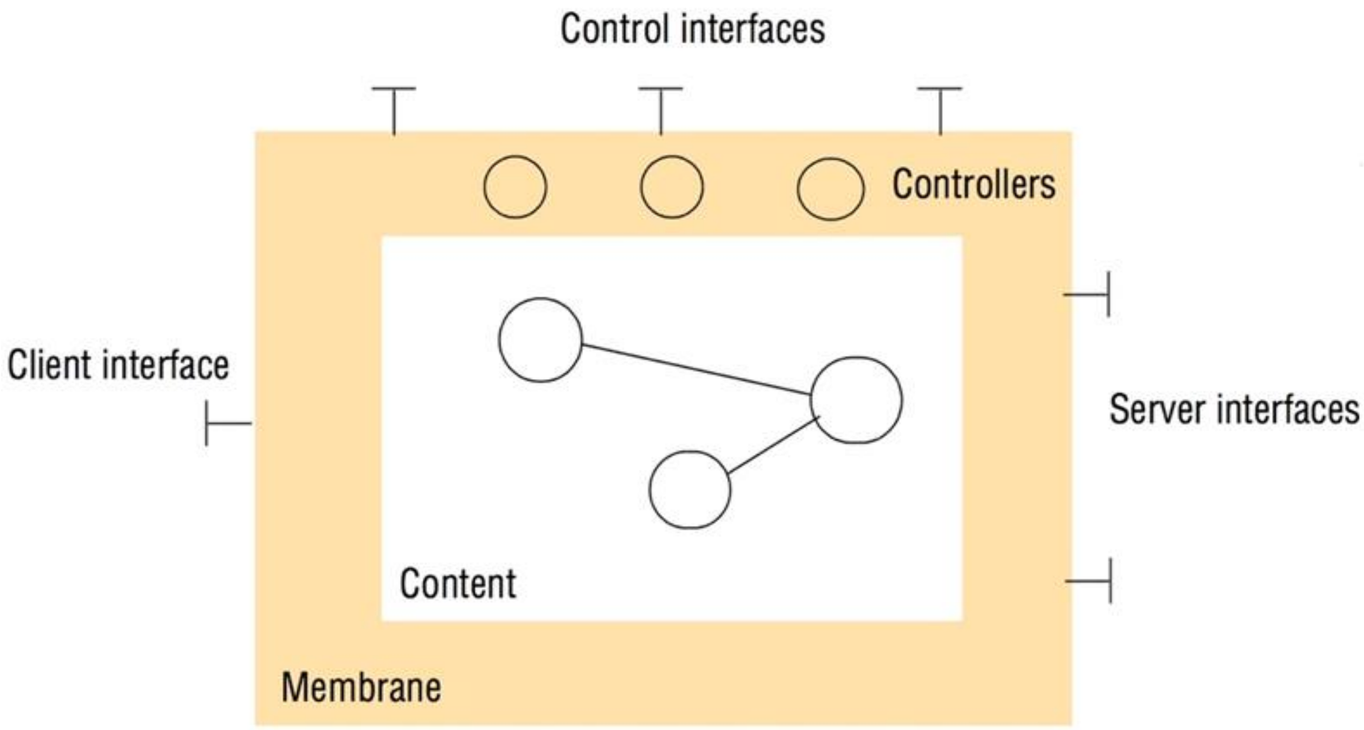
- **Fractal** is sued in building many middleware platforms including: Think, DREAM, Jasmine, GOTM, Grid Component Model (GCM), etc.
- **Fractal Core Component Model:** Server interfaces and Client interfaces. An interface is an implementation of an interface type, which defines the operations that are supported by that interface
- **Binding in Fractal:** supports two styles of binding:
  - **Primitive bindings:** direct mapping between client interface and one server interface
  - **Composite bindings:** an arbitrarily complex software architecture that implements the communication between number of interfaces potentially on different machines

Distributed Objects:
An example component configuration in Fractal



- The above cs.ClientServer is an example of a component with two subcomponents, Caller and Callee and using primitive binding
- Binding is created between the client interface (*this.r*) defined by the containing component cs.ClientServer, and the associated caller.r interface, and between the client interface caller.s and corresponding server interface callee.s

Distributed Objects:
The structure of Fractal component



- In implementation, a component consists of a ***membrane***, which define control capabilities associated with the component through set of controllers, and the associated ***content***.
- Controllers offers: lifecycle management, reflection capabilities, and interception capabilities like EJB

Distributed Objects: Component and ContentController interfaces in Fractal

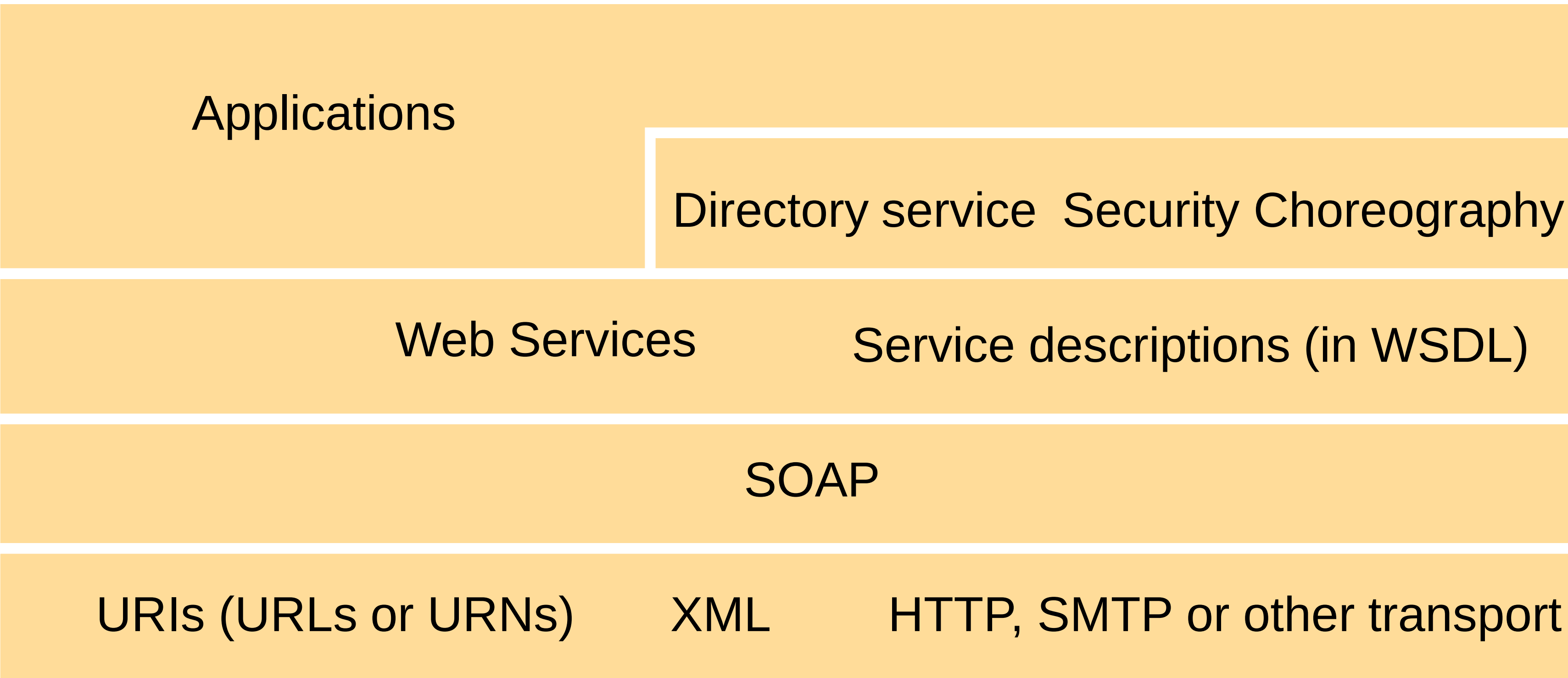
```
public interface Component {
    Object[] getFcInterfaces ();
    Object getFcInterface (String itfName);
    Type getFcType ();
}

public interface ContentController {
    Object[] getFcInternalInterfaces ();
    Object getFcInterfaceInterface(String itfName);
    Component[] getFcSubComponents ();
    void addFcSubComponent (Component c);
    void removeFcSubComponent(Component c);
}
```

Web Services

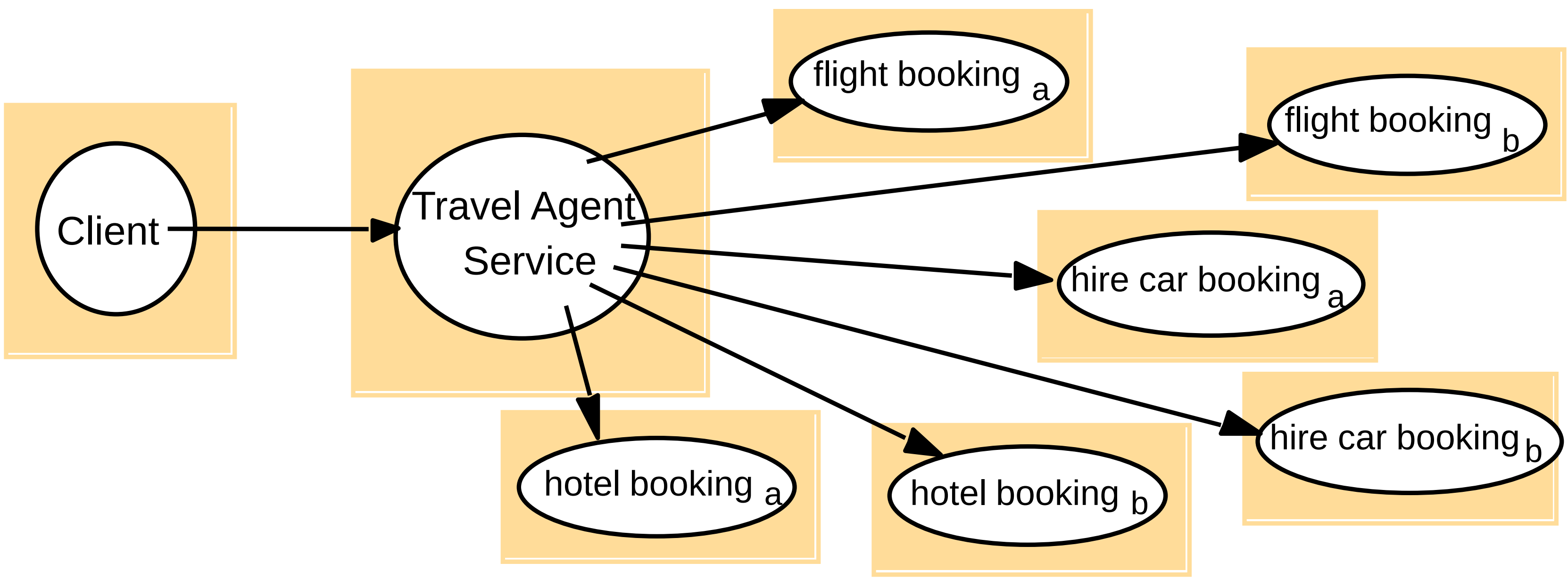
Web Services:

Web Services infrastructure and components



- **SOAP**: Simple Object Access Protocol - https://www.w3.org/TR/soap/
- **WSDL**: Web Services Description Language - https://www.w3.org/TR/2007/REC-wsdl20-20070626/

Web Services: The ‘travel agent service’ combines other web services

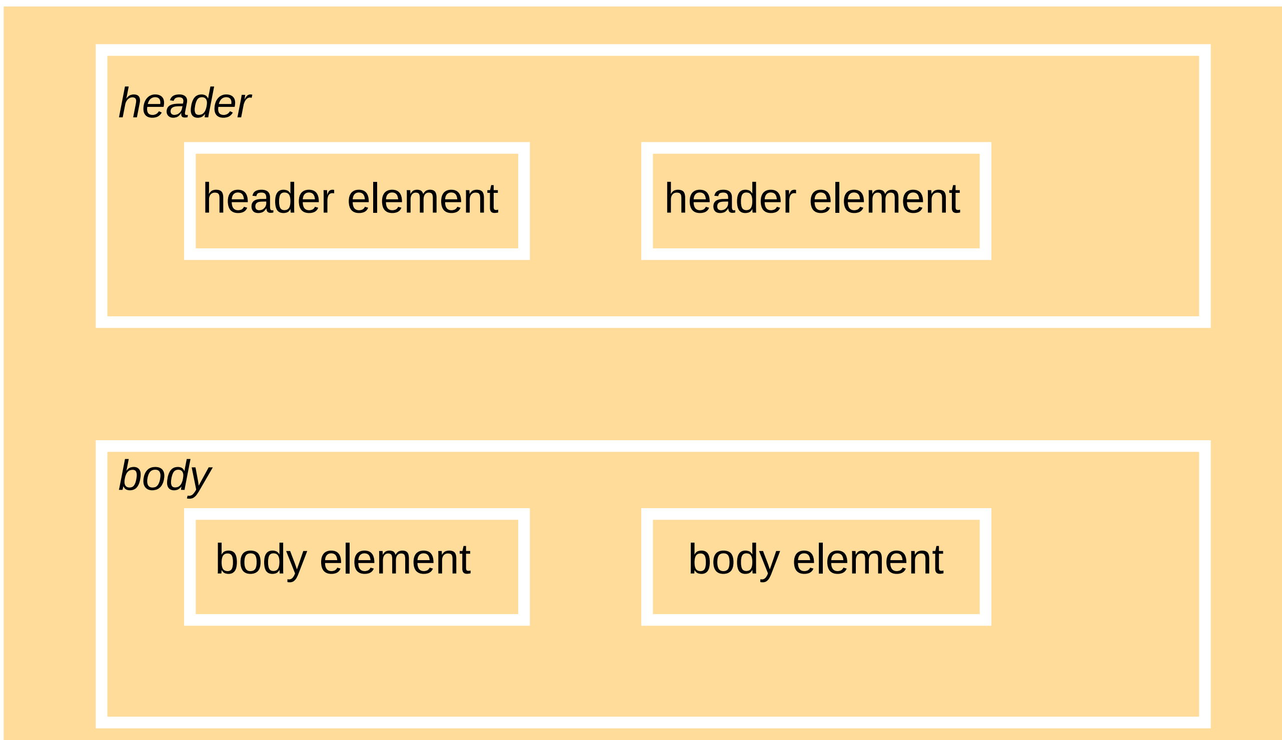


Web Services:
SOAP message in an envelope

SOAP in TCP/IP- protocol staple

Anwendung	**SOAP**				
	HTTP	HTTPS	...		
Transport	TCP				
Internet	IP (IPv4, IPv6)				
Netzzugang	Ethernet	Token Bus	Token Ring	FDDI	...

envelope



- **SOAP message** is carried in an envelope. Inside the envelope there is an optional header and body

Web Services:

Example of a simple request without headers

env:envelope xmlns:env = namespace URI for SOAP envelopes

env:body

m:exchange

xmlns:m = namespace URI of the service description

m:arg1
Hello

m:arg2
World

- **In this figure and the next**, each XML element is represented by a shaded box with its name in italic followed by any attributes and its content
- **XML elements envelope**, header, body together with other attributes are defined as the schema in the SOAP XML namespace

Web Services: Example of a reply corresponding to the previous request

env:envelope xmlns:env = namespace URI for SOAP envelope

env:body

m:exchangeResponse
xmlns:m = namespace URI for the service description

m:res1
World

m:res2
Hello

Web Services: Use of HTTP POST request in SOAP client-server communication

```
POST /examples/stringer
Host: www.cdk4.net
Content-Type: application/soap+xml
Action: http://www.cdk4.net/examples/stringer#exchange
```

endpoint address

action

HTTP header

```
<env:envelope xmlns:env=
<env:header> </env:header>
<env:body> </env:body>
</env:Envelope>
```

namespace URI for SOAP envelope

Soap message

- **SOAP Messages are protocol independent.** Typically HTTP is used for SOAP message transport.
- **Action** is meant to specify the name of the operation requested w/o analyzing the message body

Web Services:

Java web service interface ShapeList

```
import java.rmi.*;

public interface  ShapeList extends  Remote {
        int newShape(GraphicalObject g) throws RemoteException; 1
        int numberOfShapes()   throws RemoteException;
        int getVersion()   throws    RemoteException;
        int getGOVersion(int i)      throws RemoteException;
        GraphicalObject getAllState(int i)   throws   RemoteException;
}
```

Web Services:
Java implementation of the ShapeList server

```
import java.util.Vector;

public  class  ShapeListImpl implements  ShapeList {
        private Vector theList = new Vector();
        private int version = 0;
        private Vector theVersions = new Vector();
        public  int  newShape (GraphicalObject g) throws RemoteException {
                version++;
                theList.addElement(g);
                theVersions.addElement(new Integer(version));
                return theList.size();
        }
        public int numberOfShapes() { }
        public int getVersion() { }
        public int getGOVersion(int i){  }
        public GraphicalObject getAllState(int i) { }
}
```

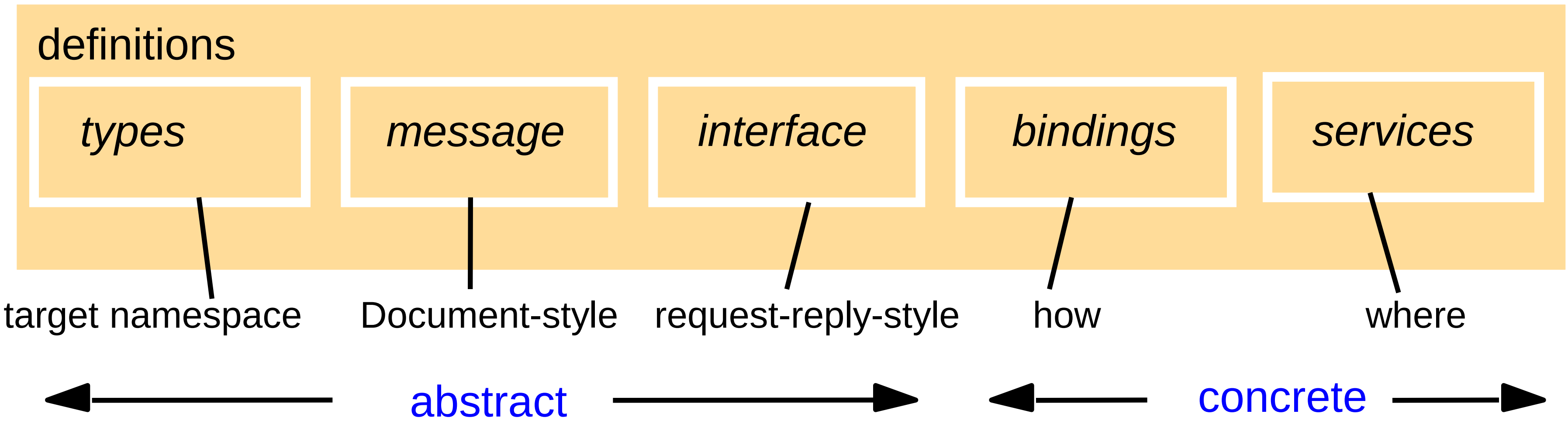
Web Services:

Java implementation of the ShapeList client

```
package staticstub;
import javax.xml.rpc.Stub;
public class ShapeListClient {
        public static void main(String[] args) {        /* pass URL of service */
          try {
                Stub proxy = createProxy();                                        1
                proxy._setProperty                                                 2
                  (javax.xml.rpc.Stub.ENDPOINT_ADDRESS_PROPERTY, args[0]);
                ShapeList aShapeList = (ShapeList)proxy;                           3
                GraphicalObject g = aShapeList.getAllState(0);                     4
          } catch (Exception ex) { ex.printStackTrace(); }
        }
        private static Stub createProxy() {                                        5
                return
                  (Stub) (new MyShapeListService_Impl().getShapeListPort());      6
        }
}
```

Web Services:

The main elements in a WSDL description



- **Service Description** are used to generate client stubs that automatically implement the correct behavior for the client. The ability to specify URI of a service as part of the service description avoids the need for name service. **WSDL** is used for service description
- **WSDL separates** the abstract part of a service description from the concrete part
- **Method of communication** is left to the service provider

Web Services: WSDL request and reply messages for the newShape operation

message name = "ShapeList_newShape" *part* name ="GraphicalObject_1" type = "ns:GraphicalObject "	*message* name = "ShapeList_newShapeResponse" *part* name= "result" type= "xsd:int"
tns – target namespace	xsd – XML schema definitions

- The above Figure shows the request and reply messages for the *newShape* operation, which has single input argument of type *GraphicalObject* and a single output argument of type *int*.

Web Services:

Message Exchange patterns for WSDL operations

Name	*Messages sent by*			
	Client	*Server*	*Delivery*	*Fault message*
In-Out	*Request*	*Reply*		may replace *Reply*
In-Only	*Request*			no fault message
Robust In-Only	*Request*		guaranteed	may be sent
Out-In	*Reply*	*Request*		may replace *Reply*
Out-Only		*Request*		no fault message
Robust Out-Only		*Request*	guaranteed	may send fault

Web Services:
WSDL operation newShape

*operation*name = "newShape"
pattern = In-Out

input message = tns:ShapeList_newShape

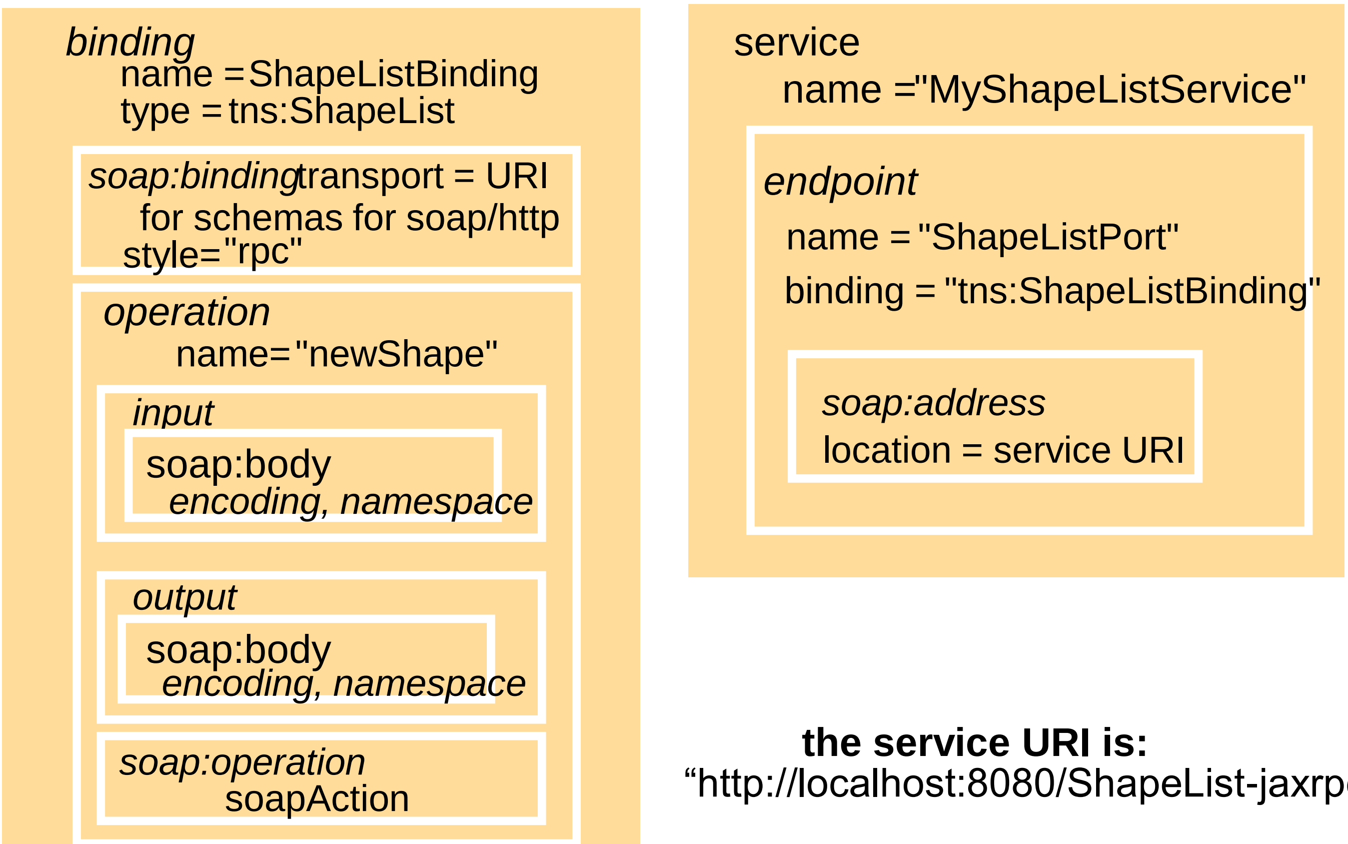
output message = "tns:ShapeList_newShapeResponse"

tns – target namespace xsd – XML schema definitions

The names: *operation*, *pattern*, *input* and *output* are defined in the XML schema for WSDL

Web Services:

SOAP binding and service definitions

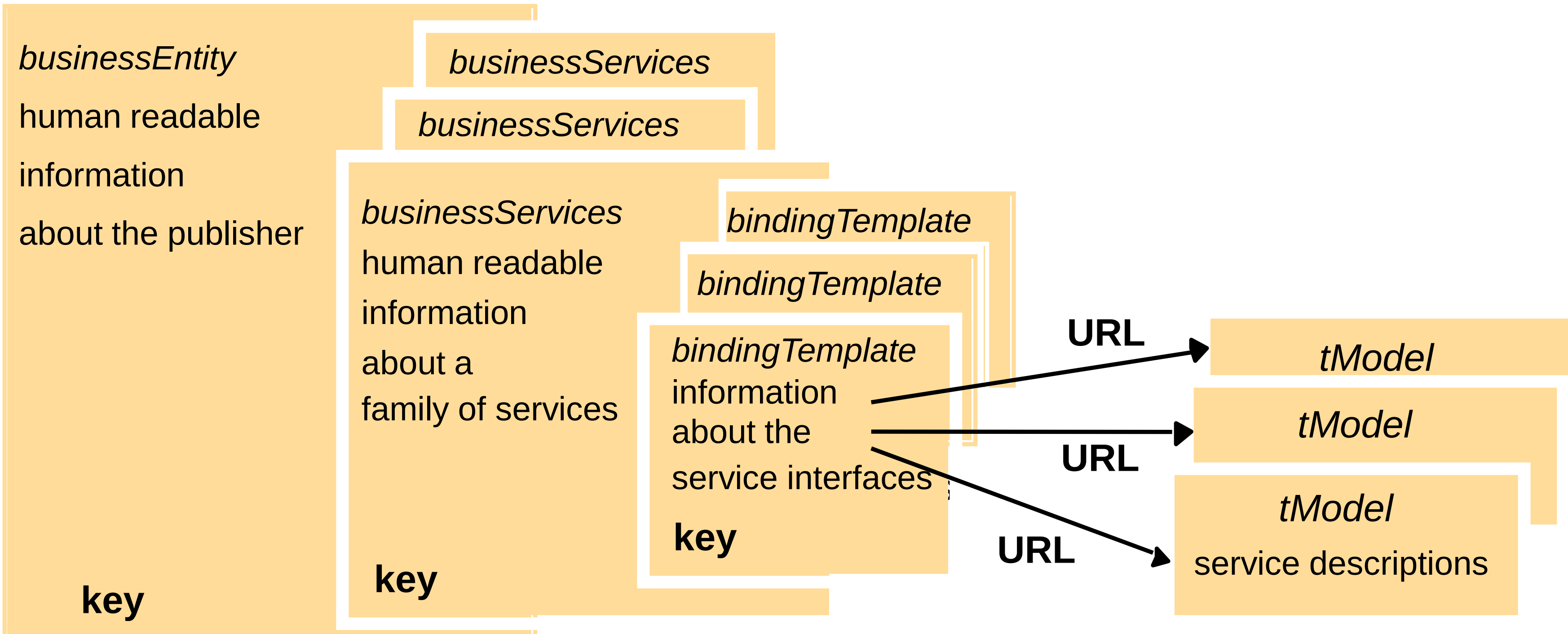


the service URI is:
“http://localhost:8080/ShapeList-jaxrpc/ShapeList”

- **The above Figure** shows the binding for one of the operations (*newShape*) specifying that input & output messages should travel in a soap:body, using a particular encoding style + this operation should be transmitted as a *soapAction*

Web Services:

The main UDDI data structures



- **UDDI:** Universal Description, Discovery, and Integration
  http://uddi.xml.org/
- Universal method to dynamically discover and invoke web services

Web Services:

Algorithms required for XML signature

Type of algorithm	*Name of algorithm*	*Required*	*reference*
Message digest	SHA-1	Required	Section 7.4.3
Encoding	base64	Required	[Freed and Borenstein 1996]
Signature	DSA with SHA-1	Required	[NIST 1994]
(asymmetric)	RSA with SHA-1	Recommended	Section 7.3.2
MAC signature (symmetric)	HMAC-SHA-1	Required	Section 7.4.2 and Krawczyk *et al.* [1997]
Canonicalization	Canonical XML	Required	Page 810

- **Specifications for digital signature** in XML is defined as an XML element types to hold: signature, name of the algorithm, keys and reference to signed information

Web Services:
Algorithms required for encryption

Type of algorithm	*Name of algorithm*	*Required*	*reference*
Block cipher	TRIPLEDES, AES 128 AES-256	required	Section 7.3.1
	AES-192	optional	
Encoding	base64	required	[Freed and Borenstein 1996]
Key transport	RSA-v1.5, RSA-OAEP	required	Section 7.3.2 [Kaliski and Staddon 1998]
Symmetric key wrap (signature by shared key)	TRIPLEDES KeyWrap, AES-128 KeyWrap, AES 256KeyWrap	required	[Housley 2002]
	AES-192 KeyWrap	optional	
Key agreement	Diffie-Hellman	optional	[Rescorla, 1999]

Web Services:

Travel agent scenario – pg-62

1. The client asks the travel agent service for information about a set of services; for example, flights, car hire and hotel bookings.
2. The travel agent service collects prices and availability information and sends it to the client, which chooses one of the following on behalf of the user:
   (a) refine the query, possibly involving more providers to get more information, then repeat step 2;
   (b) make reservations;
   (c) quit.
3. The client requests a reservation and the travel agent service checks availability.
4. Either all are available;
   or for services that are not available;
   either alternatives are offered to the client who goes back to step 3;
   or the client goes back to step 1.
5. Take deposit.
6. Give the client a reservation number as a confirmation.
7. During the period until the final payment, the client may modify or cancel reservations

Web Services:

A selection of Amazon Web Services (AWS)

Web service	*Description*
Amazon Elastic Compute Cloud (EC2)	Web-based service offering access to virtual machines of a given performance and storage capacity
Amazon Simple Storage Service (S3)	Web-based storage service for unstructured data
Amazon Simple DB	Web-based storage service for querying structured data
Amazon Simple Queue Service (SQS)	Hosted service supporting message queuing (as discussed in Chapter 6)
Amazon Elastic MapReduce	Web-based service for distributed computation using the MapReduce model (introduced in Chapter 21)
Amazon Flexible Payments Service (FPS)	Web-based service supporting electronic payments

Web Services:
RESTfull APIs overview

- **REST** is **RE**presentational **S**tate **T**ransfer
- **It is an application program interface (API)** that uses HTTP requests to GET, PUT, POST and DELETE data.
- **REST technology** is preferred to the more robust SOAP technology because REST uses les bandwidth, making it more suitable for internet usage.
- **API** is code that allows two software programs to communicate with each other, e.g., browser communicate with cloud services.
- **With REST**, networked components are a resource that you request access to.
- **All calls are stateless**, i.e., nothing is retained by the RESTful service between calls.

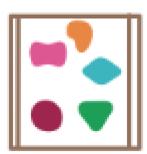
Web Services:
RESTfull APIs overview

- **RESTful** are relevant to cloud, away from HTTP, as stateless components can be freely redeployed if something fails, and can scale better to accommodate load change.
- **REST is Resource-based (vs. Action-based like SOAP)**
  - **Nouns (person, user, address) vs. verbs (RPC, methods, actions)**
  - **Identified by URIs –** multiple URIs may refer to same resource
- **Representations passed between client and server**
- **Six Constraints of the architecture:**
  - **Uniform interface:** use HTTP and its verbs (get, put, post, delete)
  - **Stateless:** server has no client state
  - **Client-server:** assume a disconnected system
  - **Cacheable:** server response at the client side may be cacheable
  - **Layered system:** client can’t assume direct connection to server
  - **Code on demand:** server can temporarily extent client (transfer logic to client, e.g. Java applet, JavaScript

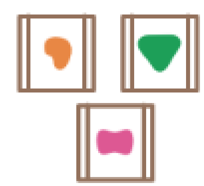
Web Services:
Microservices overview

- **Microservices architectural** style is an approach to developing a single application as a suite of small services, each running in its own process and communicating with lightweight mechanisms, often an HTTP resource API.
- **These services** are built around business capabilities and independently deployable by fully automated deployment machinery. There is a bare minimum of centralized management of these services, which may be written in different programming languages and use different data storage technologies.

A monolithic application puts all its functionality into a single process...



A microservices architecture puts each element of functionality into a separate service...



Web Services:

What is Microservices

- **Microservices is a Service Architecture:**
    - Microservices is an architecture style. It expects you to structure an applications loosely coupled services – refining one service should not impact other services
- **Focus on Single Business Function:**
    - **Micro** does not mean small services, rather read Micro in terms of the business capabilities they support. Each Service:
        - Perform one business function
        - Developed by small team
        - Develops, tests, deploys, evolve, and manage software that embody a business capability and size is secondary
- **Microservices are not always the Answer:**
    - Don’t underestimate the complexity of that change, automation tools helps partially. Examples: Monitoring and testing are more complex as you have larger number mini-apps to deploy and monitor, etc.

Peer-to-Peer Systems

Peer-to-Peer Systems: Overview

- The client-server model is appropriate for service-oriented situations. However, there are other computational goals for which a more equal division of labor is a better choice. The term *peer-to-peer* is used to describe such distributed systems. All the computers send and receive data, and they all contribute some processing power and memory. As a distributed system increases in size, its capacity of computational resources increases.
- This means that peers need to be able to communicate with each other reliably. In order to make sure that messages reach their intended destinations, peer-to-peer systems need to have an organized network structure. The components in these systems cooperate to maintain enough information about the locations of other components to send messages to intended destinations.

Peer-to-Peer Systems: Overview

- In some peer-to-peer systems, the job of maintaining the health of the network is taken on by a set of specialized components. Such systems are not pure peer-to-peer systems, because they have different types of components that serve different functions. The components that support a peer-to-peer network act like scaffolding: they help the network stay connected.
- The most common applications of peer-to-peer systems are data transfer and data storage. For data transfer, each computer in the system contributes to send data over the network. For data storage, the data set may be too large to fit on any single computer, or too valuable to store on just a single computer. Each computer stores a small portion of the data, and there may be multiple copies of the same data spread over different computers. When a computer fails, the data can be restored from other copies and put back when a replacement arrives.

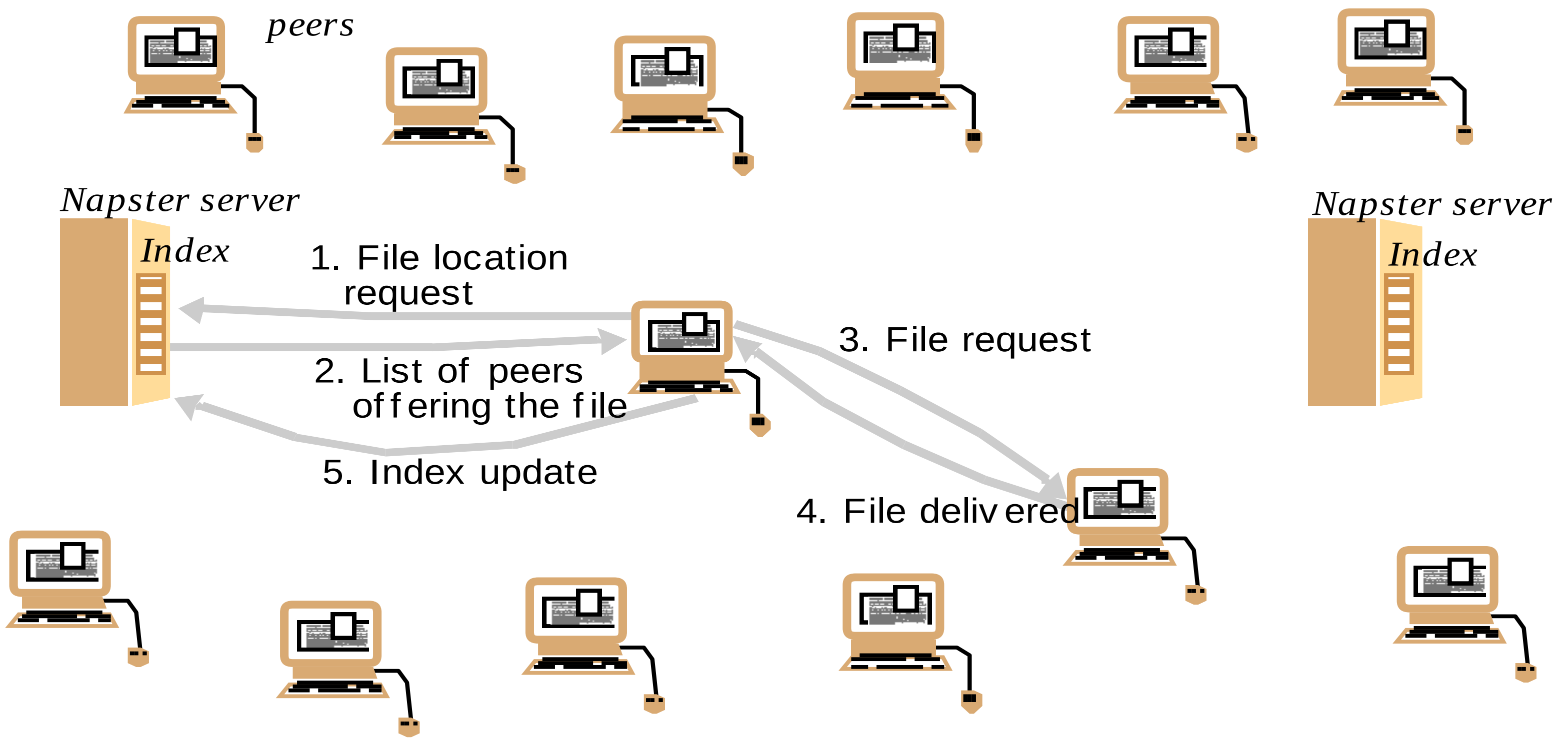
Peer-to-Peer Systems: Overview

- Skype, the voice- and video-chat service, is an example of a data transfer application with a peer-to-peer architecture. When two people on different computers are having a Skype conversation, their communications are broken up into packets of 1s and 0s and transmitted through a peer-to-peer network. This network is composed of other people whose computers are signed into Skype. Each computer knows the location of a few other computers in its neighborhood. A computer helps send a packet to its destination by passing it on a neighbor, which passes it on to some other neighbor, and so on, until the packet reaches its intended destination. Skype is not a pure peer-to-peer system. A scaffolding network of *supernodes* is responsible for logging-in and logging-out users, maintaining information about the locations of their computers, and modifying the network structure to deal with users entering and leaving.

Peer-to-Peer Systems: Distinctions between IP and overlay routing for P2P applications

	IP	*Application-level routing overlay*
Scale	IPv4 is limited to 232 addressable nodes. The IPv6 name space is much more generous (2128), but addresses in both versions are hierarchically structured and much of the space is pre-allocated according to administrative requirements.	Peer-to-peer systems can address more objects. The GUID name space is very large and flat (>2128), allowing it to be much more fully occupied.
Load balancing	Loads on routers are determined by network topology and associated traffic patterns.	Object locations can be randomized and hence traffic patterns are divorced from the network topology.
Network dynamics (addition/deletion of objects/nodes)	IP routing tables are updated asynchronously on a best-efforts basis with time constants on the order of 1 hour.	Routing tables can be updated synchronously or asynchronously with fractions of a second delays.
Fault tolerance	Redundancy is designed into the IP network by its managers, ensuring tolerance of a single router or network connectivity failure. *n*-fold replication is costly.	Routes and object references can be replicated *n*-fold, ensuring tolerance of *n* failures of nodes or connections.
Target identification	Each IP address maps to exactly one target node.	Messages can be routed to the nearest replica of a target object.
Security and anonymity	Addressing is only secure when all nodes are trusted. Anonymity for the owners of addresses is not achievable.	Security can be achieved even in environments with limited trust. A limited degree of anonymity can be provided.

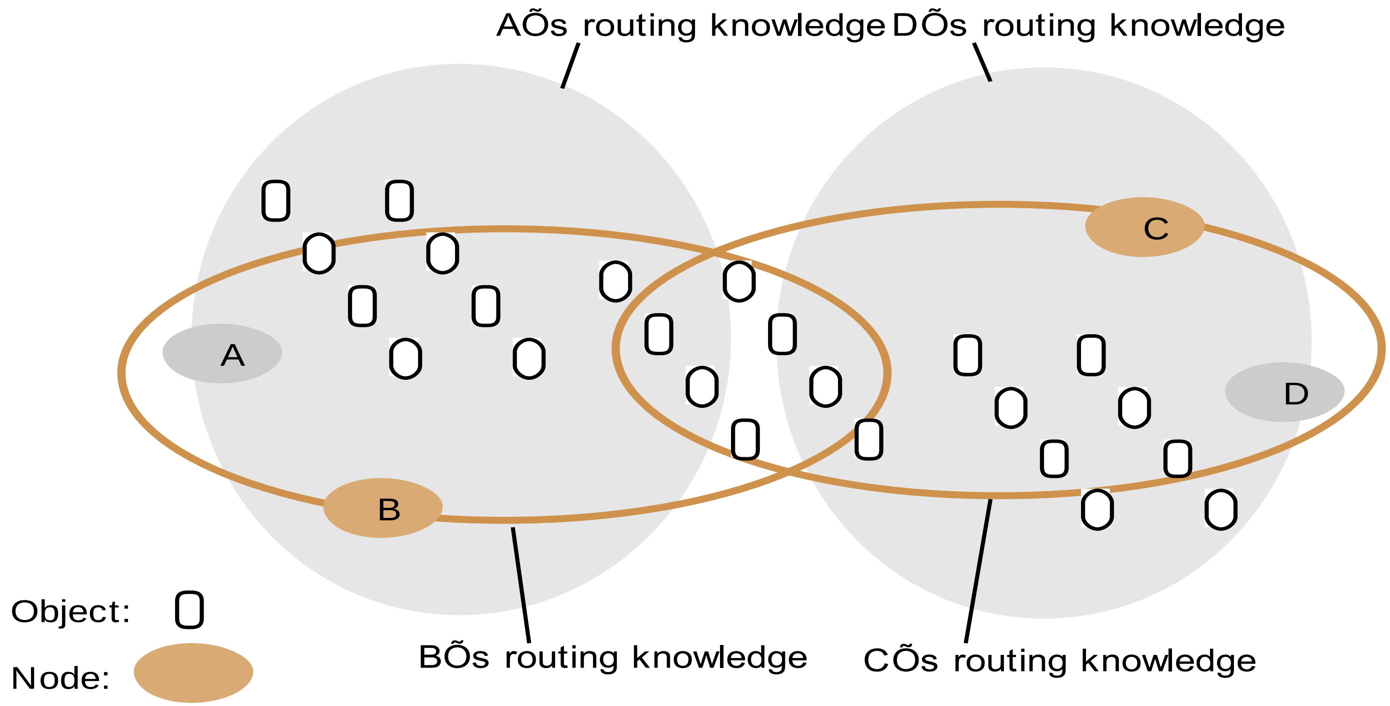
Peer-to-Peer Systems: Napster P2P file sharing with a centralized replicated index



- **Napster** uses centralize Index server.
- In the above diagram, user goes to the index server to find target location(s), gets the music file they want. Ultimately the user would update the index to add themselves as having that movie file as well.

Peer-to-Peer Systems:

Distribution of information in a routing overlay



- A main problem in P2P is the “location problem” – in Napster it is centralized index server which is not acceptable. Location information must be partitioned and distributed throughout the network. Each node is made responsible for maintaining detailed location information of nodes in a portion of the namespace. A high degree of replication is necessary (system used replication factors as high as **16**).

Peer-to-Peer Systems:
Routing overlays

Basic programming interface for a distributed hash table (DHT) as implemented by the PAST API over Pastry

put*(GUID, data)*
The *data* is stored in replicas at all nodes responsible for the object identified by *GUID*.

remov*e(GUID)*
Deletes all references to *GUID* and the associated data.

value = ***get****(GUID)*
The data associated with *GUID* is retrieved from one of the nodes responsible it.

Peer-to-Peer Systems:
Routing overlays

Basic programming interface for distributed object location and routing (DOLR) as implemented by Tapestry

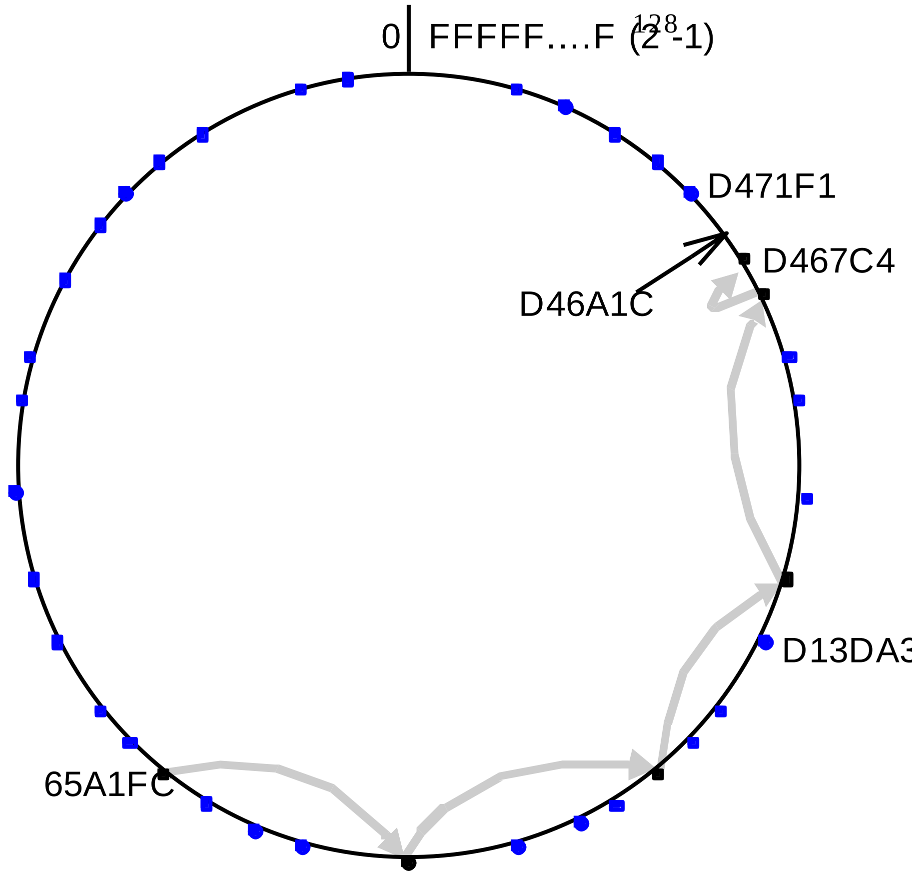
publish*(GUID)*
GUID can be computed from the object (or some part of it, e.g. its name). This function makes the node performing a *publish* operation by the host for the object corresponding to *GUID*.

unpublish*(GUID)*
Makes the object corresponding to *GUID* inaccessible.

sendToObj*(msg, GUID, [n])*
Following the object-oriented paradigm, an invocation message is sent to an object in order to access it. It might be a request to open a TCP connection for data transfer or to return a message containing all or part of the object's state. The final optional parameter *[n]*, if present, requests the delivery of the same message to n replicas of the object.

Peer-to-Peer Systems:
Routing overlays

Circular routing alone is correct but inefficient - Based on Rowstron and Druschel [2001]

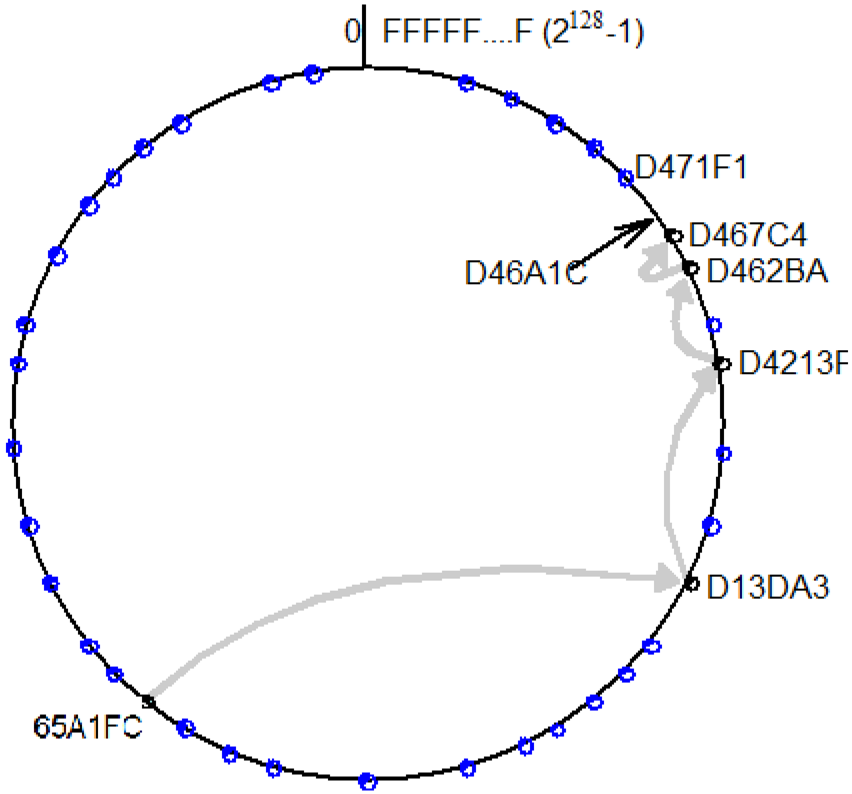


- The dots depict live nodes. The space is considered as circular: node 0 is adjacent to node (2^{128}-1). The diagram illustrates the routing of a message from node 65A1FC to D46A1C using leaf set information alone, assuming leaf sets of size 8 (l= 4).
- Each active node stores a leaf set of (size 2l) containing the <GUID, IP> of the nodes whose GUID are numerically closest on either side of its own (GUID ± 4)
- This is a degenerate type of routing that would scale very poorly; it is not used in practice.

Peer-to-Peer Systems:

Pastry routing example

Based on Rowstron and Druschel [2001]



Routing a message from node 65A1FC to D46A1C. With the aid of a well-populated routing table the message can be delivered in ~$\log_{16}$(N) hops.

- Routing at any node A uses the information in its routing table R and the leaf set L to handle any request from another node using the algorithm presented next page.

Peer-to-Peer Systems:
Pastry's routing algorithm

To handle a message M addressed to a node D (where $R[p,i]$ is the element at column i, row p of the routing table):

1. If ($L_{-l} < D < L_l$) { // the destination is within the leaf set or is the current node.
2. Forward M to the element L_i of the leaf set with GUID closest to D or the current node A.
3. } else { // use the routing table to despatch M to a node with a closer GUID
4. find p, the length of the longest common prefix of D and A. and i, the $(p+1)^{th}$ hexadecimal digit of D.
5. If ($R[p,i]$ ° $null$) forward M to $R[p,i]$ // route M to a node with a longer common prefix.
6. else { // there is no entry in the routing table
7. Forward M to any node in L or R with a common prefix of length i, but a GUID that is numerically closer.

   }

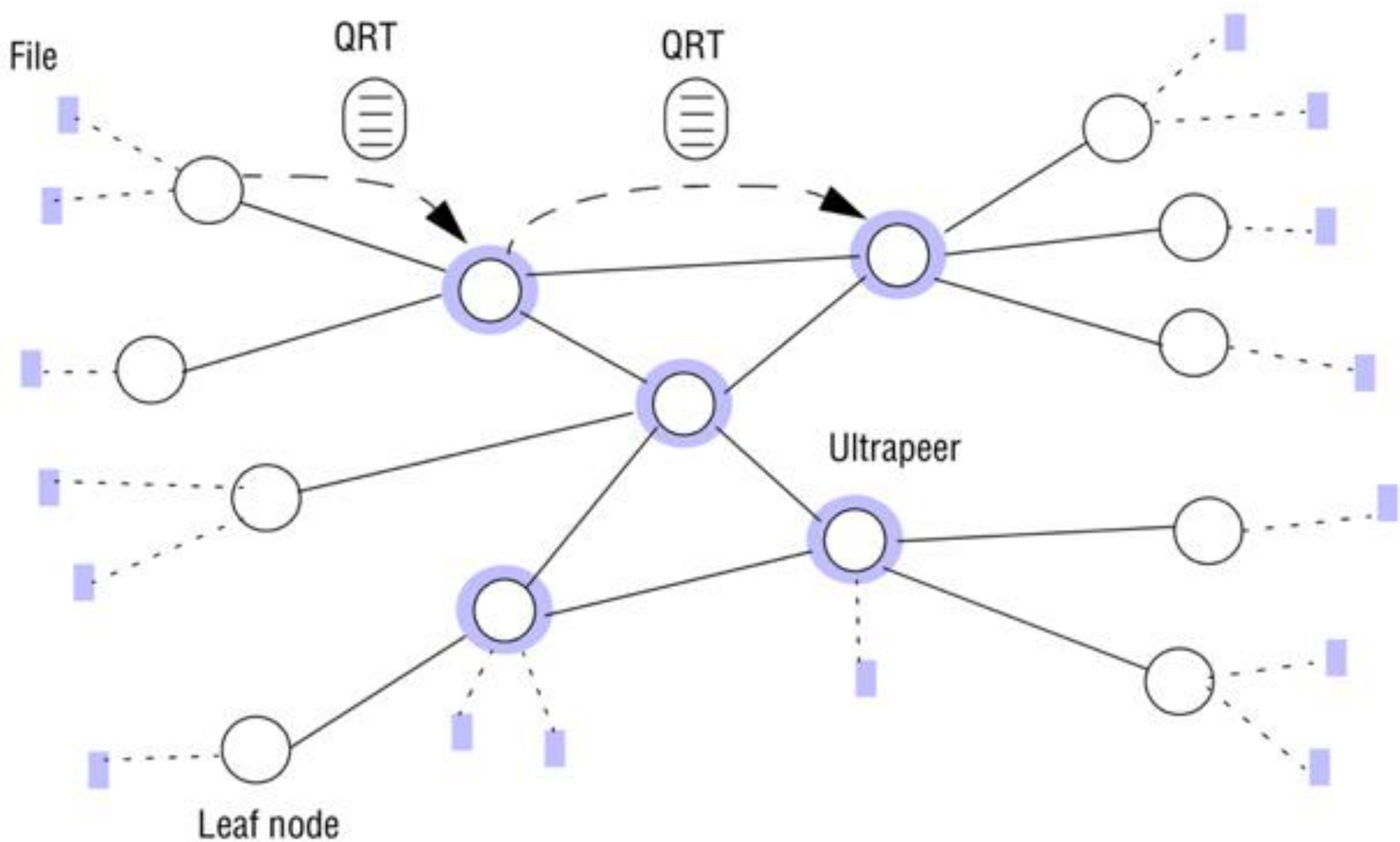
}

Peer-to-Peer Systems:

Structured vs. unstructured P2P systems

	Structured peer-to-peer	*Unstructured peer-to-peer*
Advantages	Guaranteed to locate objects (assuming they exist) and can offer time and complexity bounds on this operation; relatively low message overhead.	Self-organizing and naturally resilient to node failure.
Disadvantages	Need to maintain often complex overlay structures, which can be difficult and costly to achieve, especially in highly dynamic environments.	Probabilistic and hence cannot offer absolute guarantees on locating objects; prone to excessive messaging overhead which can affect scalability.

Peer-to-Peer Systems:
Key elements in the Gnutella protocol



Peer-to-Peer Systems:

Key elements in the Gnutella protocol

- Originally, Gnutella adopted simple flooding protocol that did not scale well.
- As a follow up, move from pure P2P architecture, where all nodes are equal, but with some nodes designated to have additional resources, *ultrapeers*, and form the heart of the network.
- Other nodes takes on the role of leaf nodes (leaves). Leaf nodes connect themselves to small number of *ultrapeers* which are heavily connected to other *ultrapeers* (with over 32 connections each).
- This approach dramatically reduces the maximum number of hops required for exhaustive search.
- A leaf node hash local file into tokens into a Query Routing Table (QRT) and send it to all its associated *ultrapeers*.
- An *ultrapeer* node produces its own QRT containing union of all the entries from all connected leaves together in addition to files on that node.

Peer-to-Peer Systems:
Key elements in the Gnutella protocol

- Exchange the constructed QRT with other *ultrapeer* nodes. As a result *ultrapeers* have global view of the location information.
- A leaf node can query closest *ultrapper* node to get the location of any file in the network.
- This style of Peer-to-Peer is referred to as hybrid architecture; this approach is also adopted by Skype.

END