Infrastructure Intergicielle pour Applications Distribuées
Infraestruturas "Middleware" para Aplicaciones Distribuídas
LAFMI, Xalapa, 6-10.05.2002

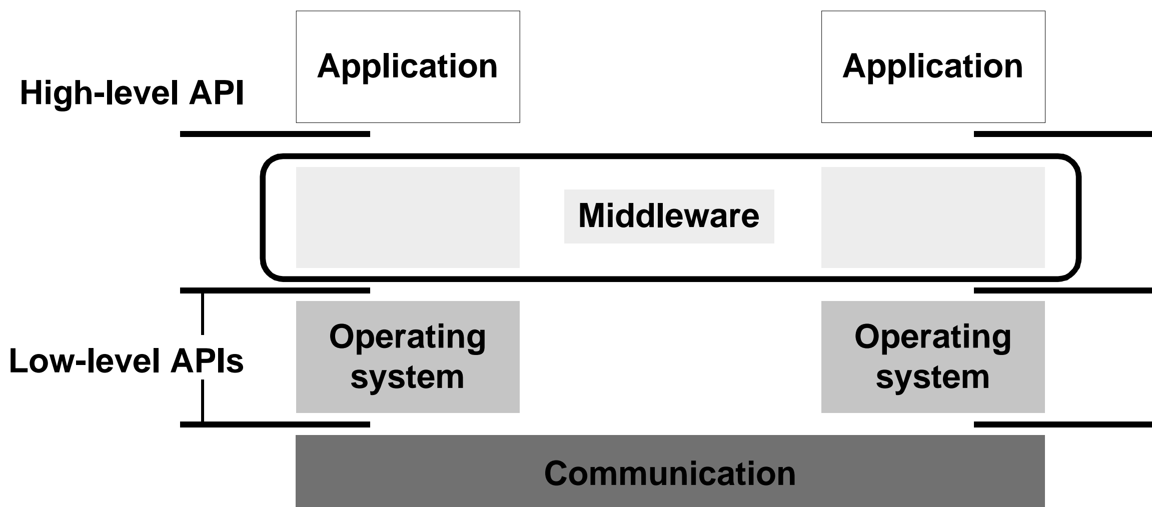
Patterns and Frameworks for Middleware Construction

Sacha Krakowiak
Université Joseph Fourier, Grenoble
INRIA, projet Sardes

http://sardes.inrialpes.fr/people/krakowia

An introduction to Middleware

- Middleware sits "in the middle"



- Middleware is a relatively recent notion (early 90's)

© 2002, S. Krakowiak

Functions of Middleware

- Middleware aims to improve distributed programming
  - Make application development easier by providing a common programming abstraction and by hiding low-level details
  - Facilitate evolution
  - Enhance reusability
  - Improve portability on new platforms
  - Allow interoperability of heterogeneous applications
- Middleware has four main functions
  - To provide an uniform, high level API (Applications Programming Interface) to applications
  - To hide the heterogeneity of the underlying hardware and operating systems
  - To make distribution invisible (“transparent”)
  - To provide general purpose distributed services

Categories of Middleware

Middleware takes a wide variety of forms

- Remote Procedure Call
- Distributed Objects
  - CORBA, DCOM, .NET, Java RMI
- Distributed Components
  - Java Beans, Enterprise Java Beans, CCM
- Message-Oriented Middleware (MOM)
  - Message Queues, Publish-Subscribe
- Distributed tuples
  - Linda, Jini
- Data access and persistence

Goal of this course

- To present the principles of middleware architecture In a systematic way
  - By identifying the main design and implementation problems
  - By exhibiting the main design patterns relevant to middleware construction
  - By illustrating these patterns with useful frameworks
- To present detailed examples of how middleware actually works
  - By going down to the source code level

The examples are taken from ObjectWeb, a consortium dedicated to the development of innovative open source middleware (www.objectweb.org)

Plan of the course

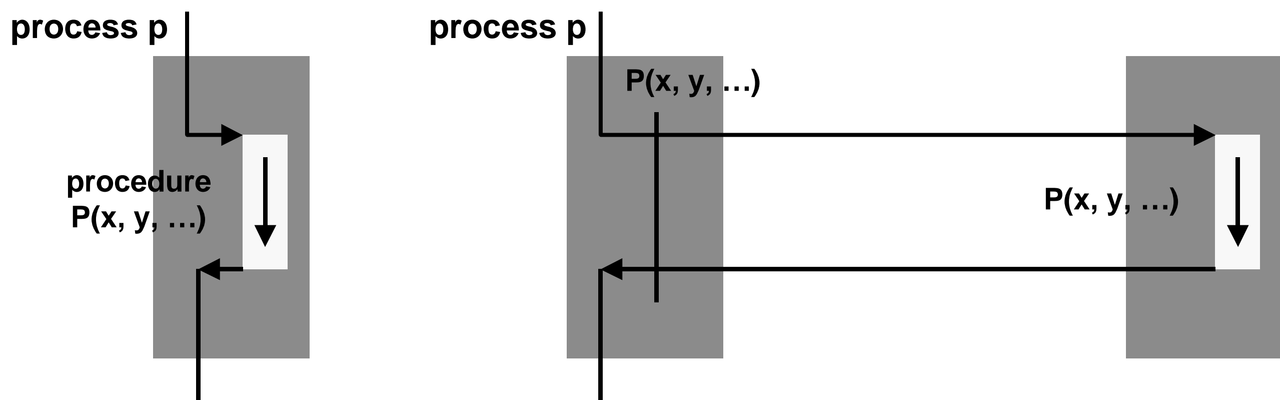
- A refresher on RPC and Java RMI
  - introduce the main issues of middleware
  - Introduce basic architectural concepts
- From objects to components
  - Examples with Enterprise Java Beans (EJB)
- A few basic patterns for middleware
  - Proxy, Stub, Factory, Wrapper, Interceptor, etc.
  - How these patterns are actually used and combined
- Inside middleware: a case study
  - An introduction to Jonathan, an open source middleware framework
  - A code walkthrough of Jonathan, with examples
  - Three frameworks in use: naming and binding, communication, configuration

A Refresher on Client-Server Middleware

RemoteProcedure Call (RPC)
Java Remote Method Invocation (RMI)

A Refresher on RPC

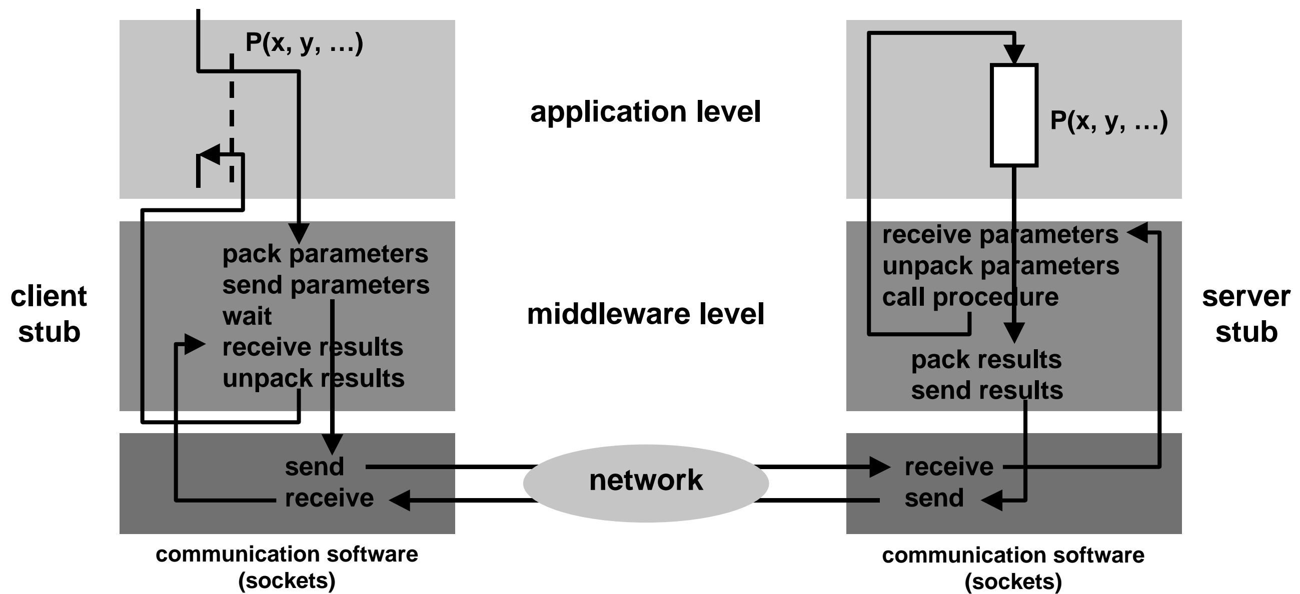
- **Remote procedure Call (RPC) is a tool for the construction of client-server applications**



The effect of the call must be identical in both situations. This cannot be achieved in the presence of failures

© 2002, S. Krakowiak

Implementation of RPC



Functions of the stubs

- **Client stub**
  - Represents the server on the client site
  - Receives local call
  - Marshalls parameters
  - Creates unique identifier for the call
  - Executes the call
  - Blocks calling process
  - Receives and unmarshalls results
  - Returns to calling process
- **Server stub**
  - Represents the client on the server site
  - Receives the call as a message
  - Unmarshalls parameters
  - Creates or selects thread to execute procedure
  - Gets and marshalls results
  - Send results to caller

In addition, both stubs must detect errors (timeouts) and react to detected errors according to the « semantics of the RPC

RPC Implementation Issues

- How does the client know the server's address
- How are parameters and results transmitted?
  - value or reference?
  - what about complex structures?
  - encoding, marshalling, unmarshalling
  - heterogeneity issues
- How are the errors dealt with?
  - error detection
  - semantics
- How are the stubs constructed?
  - automatic stub generation
- How is the execution managed?
  - server starting and stopping, etc.

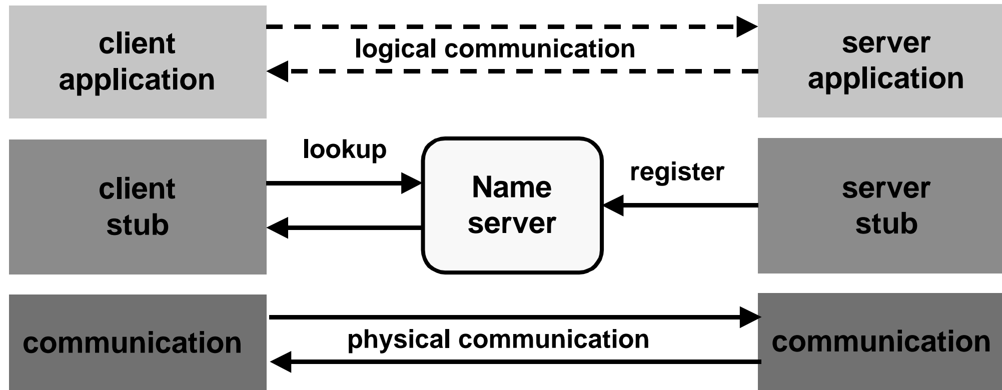
Naming and Binding in RPC

Naming: associating names with objects
Binding: using names as an access path to objects

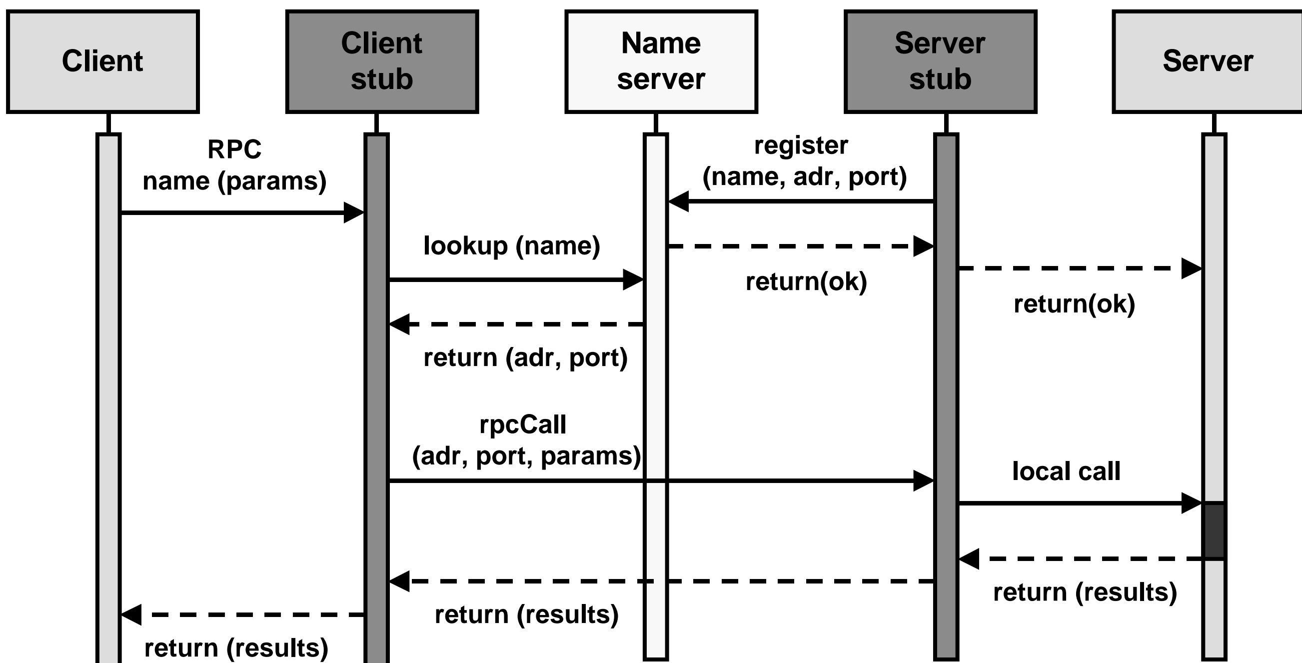
- Naming
  - Objects to name: called procedure, server site
  - Desirable properties: location-independent naming, flexibility (location of objects may change)
- Binding time
  - Early binding (static) or late binding (dynamic)
  - Static: server location known at compile time
  - Dynamic: server location unknown at compile time
    - Use symbolic names for services, and name service

Naming and Binding in RPC (2)

- Static binding: all addresses "hardwired" in code, no name server
- Binding at first call: look up name server at first call, then reuse address
- Binding at each call: look up name server at each call



Naming and binding in RPC: a dynamic view



Stub Generation

- **Principles**
  - The structure of the stubs is well defined, and they can be generated automatically
  - What information is needed?
    - environment dependent: conversion procedures, communication protocols
    - application dependent: formats of parameters and results, for marshalling and unmarshalling
- **Implementation**
  - Application dependent information is described in an Interface Definition Language (IDL)
  - The IDL defines a contract between the client (caller) and the server (callee)
  - The IDL serves as input to a stub compiler, which generates stubs

Stub Generation

- **What is in a stub?**
  - An interface (the same as that ot the remote object that it represents
  - A reference to the remote object
  - The reference must contain a means to call the object
    - An identification of the object
    - A communication object (session)
- **Generating stubs**
  - Two steps: generating the program and generating an instance
    - Program: done by a stub compiler that should implement each method of the remote object in terms of local calls
    - Instance: done by a "stub factory"
      - In the case of RPC, the loader
    - Examples later …

Object Oriented Middleware

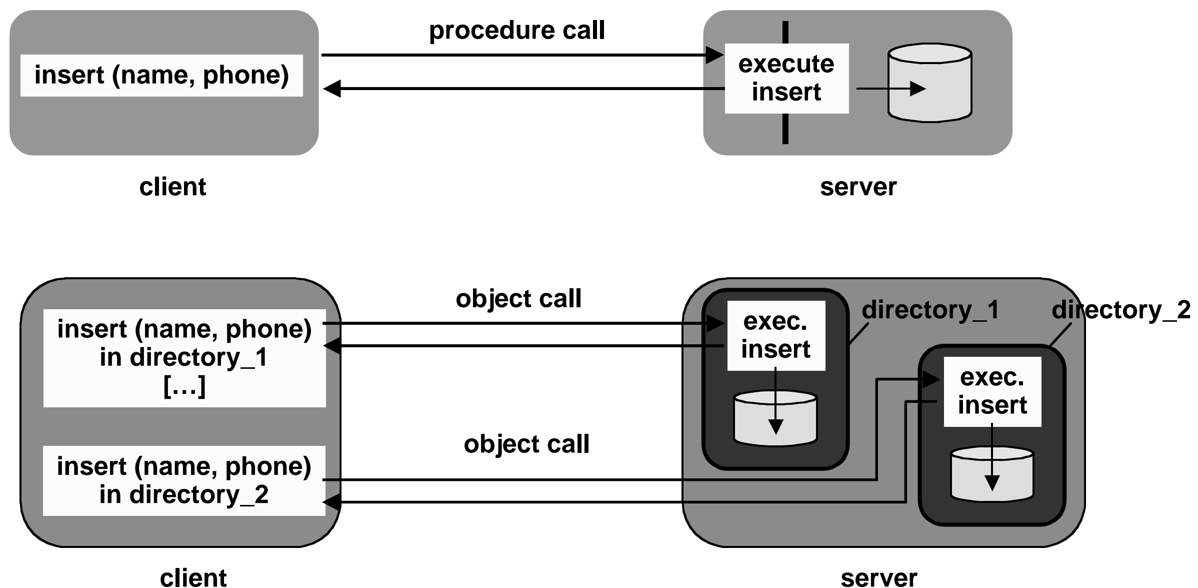
Why are objects useful for distributed applications ?

- **Encapsulation**
  - The interface (methods + attributes) is the only way to access the internal state of the object
- **Classes and instances**
  - Mechanism to generate instances according to a predefined pattern
- **Inheritance**
  - Mechanism for specialization: facilitates reuse
- **Polymorphism**
  - Different implementations of the methods of an interface
  - Replacing an object by an other one if interfaces conform
  - Facilitates application evolution and adaptation

Extending RPC to objects

Procedure call vs method call (on an object)

- Example: insert a new entry in a directory



Java RMI (*Remote Method Invocation)*

http://java.sun.com/docs/books/tutorial/rmi/

- Motivation: building distributed applications with Java
  - Method call instead of procedure call
- Principle: similar to RPC
  - The program provides
    - An interface description
      - no separate IDL: Java is used as IDL
    - The server program
      - objects that implement the interface (“servants”)
      - a server
    - The client program
  - The Java environment provides
    - A stub generator (rmic)
    - A name service (Object Registry)

Java RMI : Usage Rules (1/2)

- Interface
  - The interface of a remote object distant (Remote) is that of a Java object, with some constraints:
  - The remote interface must be public
  - The remote interface must extend the interface java.rmi.Remote
  - Each method must declare (at least) the exception java.rmi.RemoteException
- Passing objects as parameters
  - Local objects are passed by value (copy) and must be serializable (extend the interface java.io.Serializable)
  - Remote objects are passed by reference and are designated by their interface

Java RMI : Usage Rules (2/2)

- **Implementation of remote classes (Remote)**
  - A remote class must implement an interface that is itself remote (Remote)
  - A remote class must extend the class java.rmi.server.UnicastRemoteObject (there are other possibilities)
  - A remote class may also have some methods that may only be called locally (not part of its Remote interface)

Java RMI : Server Implementation Rules

- **A server is a class that implements the remote object's interface**
  - Specify the remote references that should be implemented (objects passed as parameters)
  - Define the constructor of the remote object
  - Provide the implementation of remotely callable methods
  - Create and install the security manager
  - Create at least one instance of the server class
  - Register at least one instance in the name server

Java RMI : Example (Hello world) - 1

Interface Definition

```
import java.rmi.*
public  interface HelloInterface
  extends Remote {

  /* method that prints a message
     (predefined in the called object) */

  public String sayHello ()
            throws java.rmi.RemoteException;
}
```

A class implementing the interface

```
import java.rmi.*
import java.rmi.server.*;
 public  class Hello
    extends java.rmi.server.UnicastRemoteObject
    implements HelloInterface {
              private String message;

/* the constructor */
           public Hello (String s)
                      throws RemoteException
                      {
                                message = s ;
                      }

/* implementation of the method */
           public String sayHello ()
                       throws RemoteException
                      {
                                return message ;
                      }


}
```


Java RMI : Example (Hello world) - 2

Client Program

```
import HelloInterface.*
import java.rmi.*;
public class HelloClient {
public  static void main (String [ ] argv) {

    try {

   /*find a reference to the remote object */
      HelloInterface hello =
           (HelloInterface) Naming.lookup
               ("rmi://boole.imag.fr/Hello1") ;

    /* remote method call */
       System.out.println (hello.sayHello()) ;

   } catch (Exception e)  {
       System.out.println
           ("Hello client exception : " + e) ;
   }
}}
```

Server Program

```
import HelloInterface.*
import Hello.*;
import java.rmi.*;
public class HelloServer {
public  static void  main (String [ ] argv) {

   /*start SecurityManager */

   System.setSecurityManager (
          new RMISecurityManager ()) ;

   try {

   /* create an instance of class Hello and
      register it in the nameserver */
   Naming.rebind ("Hello1",
          new Hello ("Hello world !")) ;
   System.out.println ("server ready.") ;

   } catch ((Exception e)  {
       System.out.println
          ("Server  error: " + e) ;
   }
}}
```

Java RMI : Execution Steps

- **Compilation**
  - Compile the Remote classes (javac), and the client and server programs
  - Compile the interface (rmic) to create the client and server stubs
- **Execution**
  - Start the name server (on the server machine)

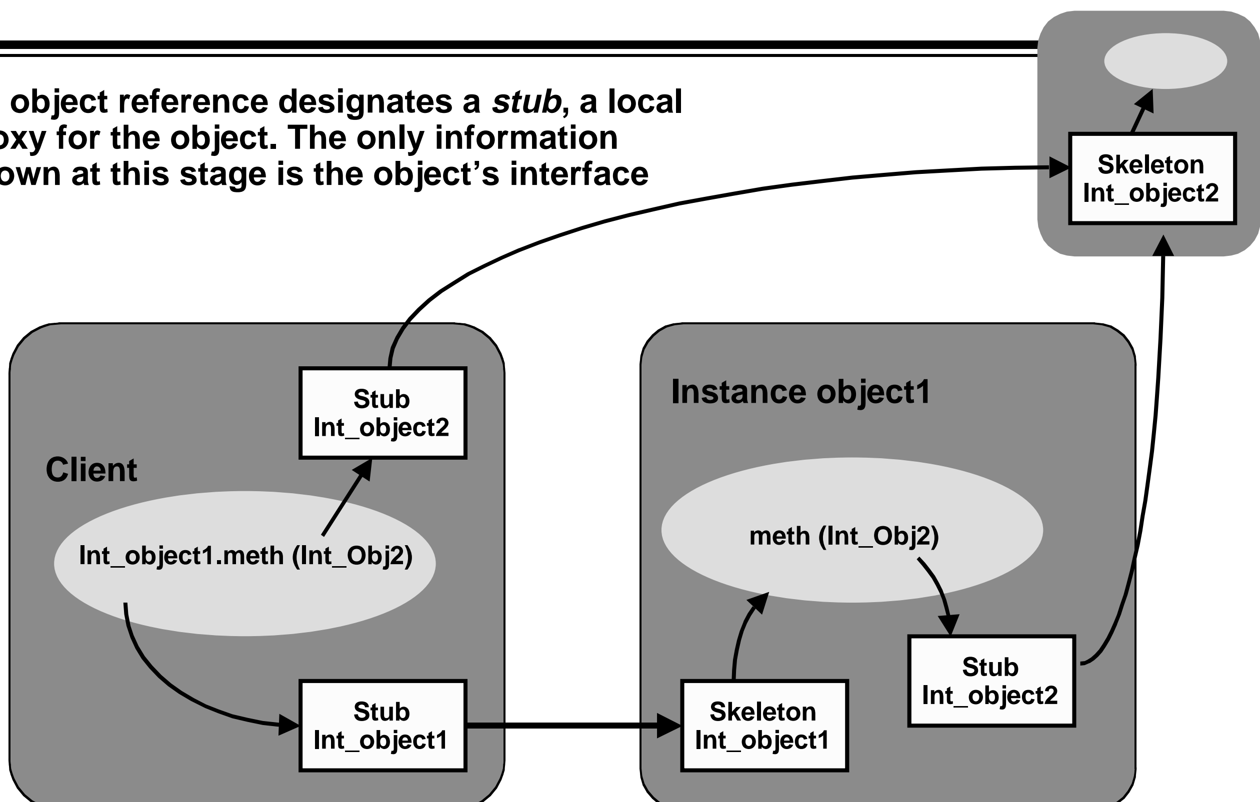
    rmiregistry &
  - Start the server

    java HelloServer &
  - Start the client

    java HelloClient

Reference Passing in Java RMI

An object reference designates a *stub*, a local proxy for the object. The only information known at this stage is the object's interface



Conclusion on Java RMI

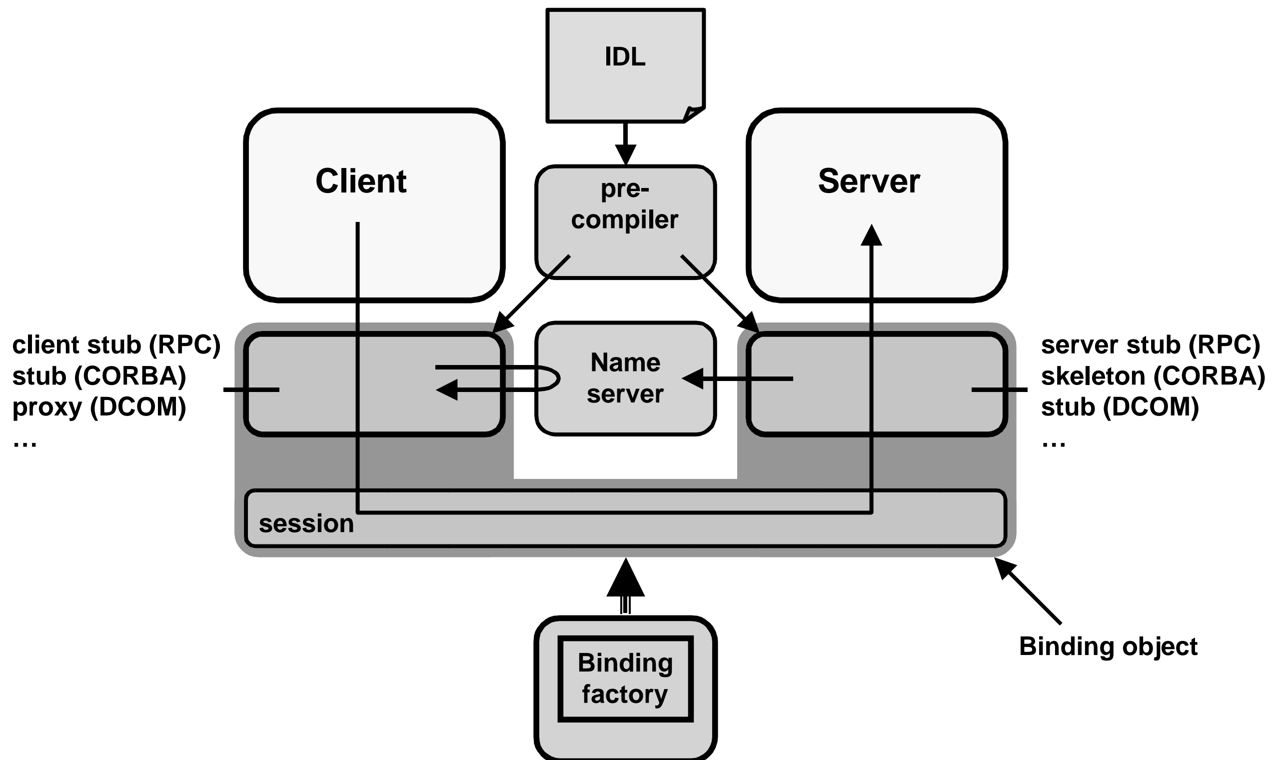
- Extending RPC to objects
  - Allows access to remote objects
  - Allows extension of the local environment by dynamic dynamic code loading
  - No separate Interface Description Language
- Limitations
  - Single language environment (Java)
    - But possible interconnection
  - Minimal services
    - Simple name service (no trading)
    - No additional services
      - Objects replication
      - Transactions
      - …

Patterns for Middleware

Introduction to Design Patterns

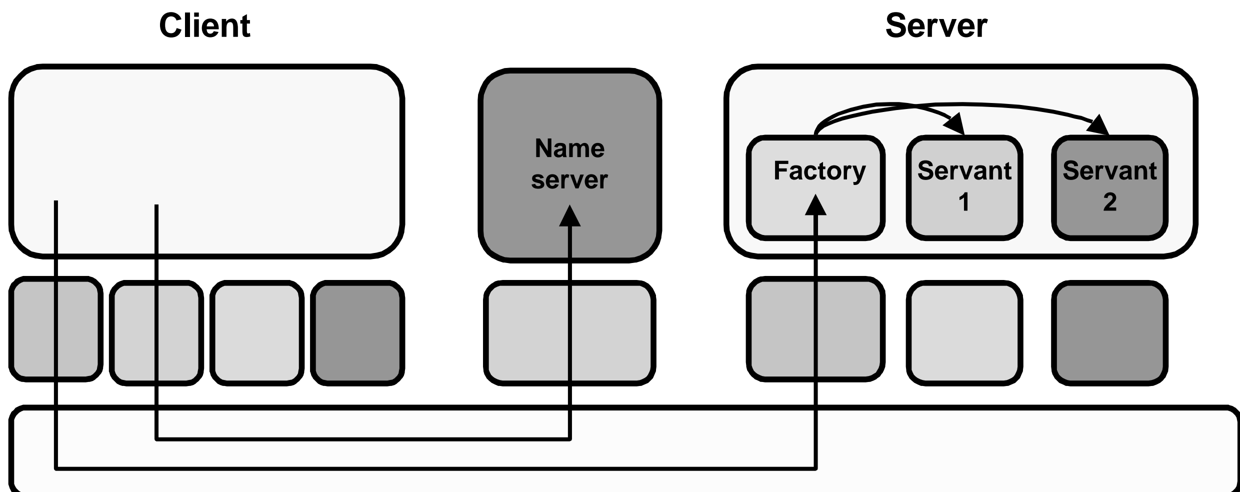
A Few Useful Patterns and their Applications

A View of Client-Server Middleware



© 2002, S. Krakowiak

Common Constructions



Two constructions:

A "representative" (*Proxy*)

A "constructor" (*Factory*)

Generic (reusable)

May be combined:

Factory Proxy
Factory Factory
Proxy Factory
Proxy Proxy

© 2002, S. Krakowiak

Design patterns

- **Definition** [not limited to program design]
  - A set of design rules (element definitions, element composition principles, rules of usage) that allow the designer to answer a class of specific needs in a specific environment.
- **Properties**
  - A design pattern is elaborated from the experience acquired during the resolution of a class of related problems ; it captures solution elements common to those problems.
  - A design pattern defines design principles, not specific implementations of those principles.
  - A design pattern provides an aid to documentation, e.g. by setting up a common terminology, or even a formal description ("pattern language")

E. Gamma, R. Helm, R. Johnson, J. Vlissides. *Design Patterns - Elements of Reusable Object-Oriented Software*, Addison-Wesley, 1995
F. Buschmann, R. Meunier, H. Rohnert, P. Sommerlad, M. Stal. *Pattern-Oriented Software Architecture* - vol. 1, Wiley 1996
D. Schmidt, M. Stal, H. Rohnert, F. Buschmann. *Pattern-Oriented Software Architecture* - vol. 2, Wiley, 2000

More about Patterns

- **Defining a Pattern**
  - Context: the design solution giving rise to a design problem; should be as generic as possible (but not overly general)
  - Problem: requirements, desirable properties of the solution; constraints of the environments
  - Solution:
    - Static aspects: components, relationships between components; may be described by class/collaboration diagrams
    - Dynamic aspects: run-time behavior, lifecycle (creation, termination, etc.); may be described by sequence/state diagrams
- **Categories of Patterns**
  - Design: small scale, commonly recurring structure within a particular context
  - Architectural: large scale, structural organization, defines subsystems and relationships between them
  - Idioms: language specific - how to implement a particular aspect in a given language

Source: F. Buschmann, R. Meunier, H. Rohnert, P. Sommerlad, M. Stal. *Pattern-Oriented Software Architecture* - vol. 1, Wiley 1996

A Few Examples of Patterns

- **Proxy**
  - Design pattern: representative for remote access
- **Factory**
  - Design pattern: object creation
- **Wrapper**
  - Design pattern: interface transformation
- **Interceptor**
  - Architectural pattern: adaptable service provision

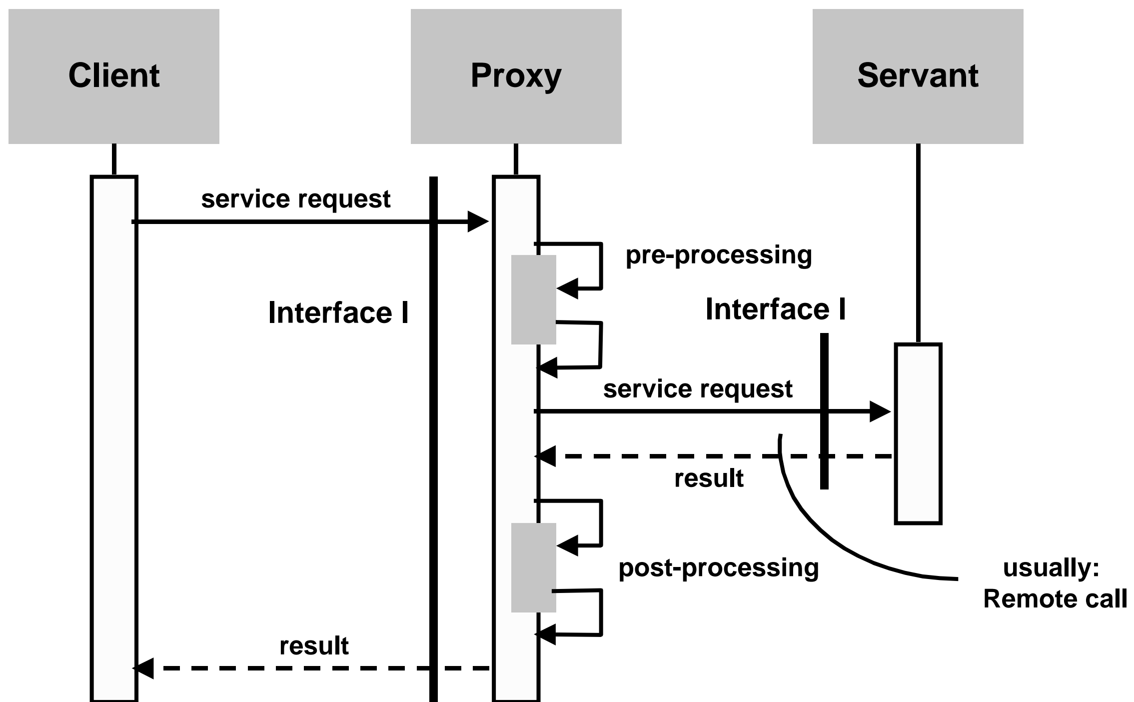
These patterns appear very frequently in middleware construction

Their use will be illustrated later in the Jonathan framework

Proxy

- **Context**
  - Applications organized as a set of objects in a distributed environment; a client needs access to the services provided by some possibly remote object (the “servant”)
- **Problem**
  - Define an access mechanism that does not involve
    - hard-coding the location of the servant into the client code
    - deep knowledge of the communication protocols by the client
  - Desirable properties
    - Access should be efficient at run time and secure
    - Programming should be simple for the client; ideally there should be no difference between local and remote access
  - Constraints
    - Distributed environment (no single address space)
- **Solutions**
  - Use a local representative of the server on the client site (isolates client from servant and communication system)
  - Keep the same interface for the representative as for the servant
  - Define a uniform proxy structure to facilitate automatic generation

Proxy in Use



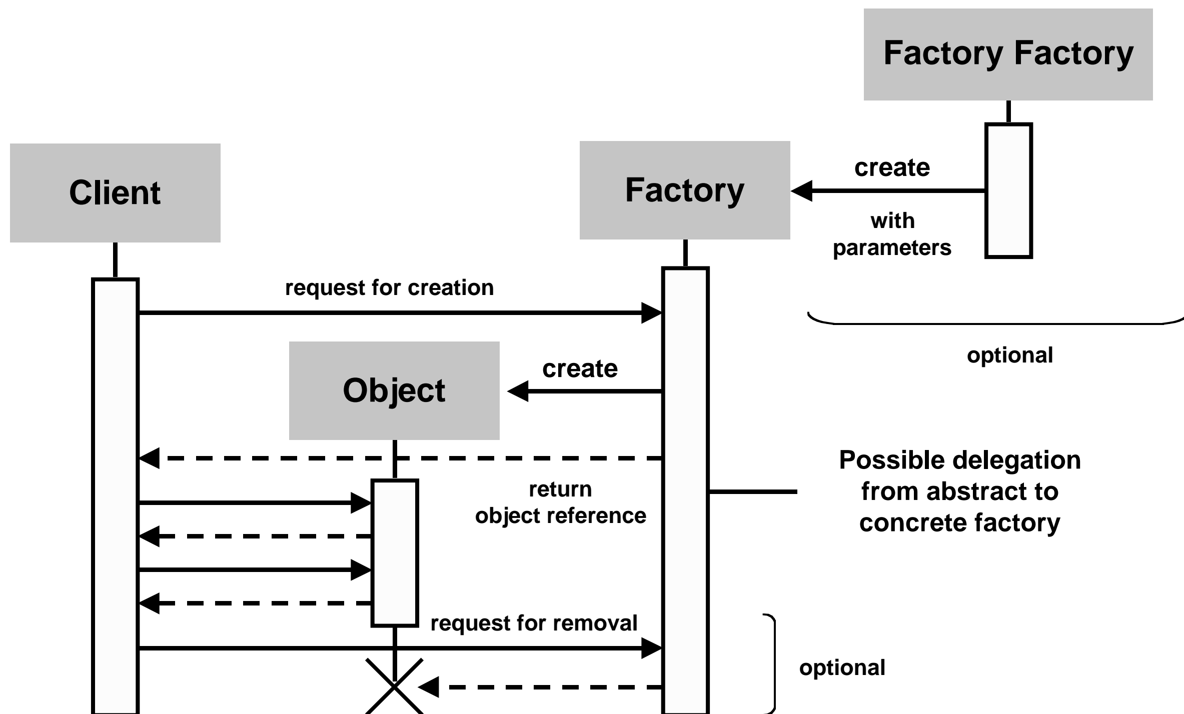
© 2002, S. Krakowiak

Factory

- **Context**
  - Applications organized as a set of objects in a distributed environment
- **Problem**
  - Dynamically create multiple instances of a class
  - Desirable properties
    - Instances should be parameterized
    - Evolution should be easy (no hard-coded decisions)
  - Constraints
    - Distributed environment (no single address space)
- **Solutions**
  - Abstract Factory: defines a generic interface and organization for creating objects; the actual creation is deferred to concrete factories that actually implement the creation methods
  - Abstract Factory may be implemented using Factory Methods (a creation method that is redefined in a subclass)
  - A further degree of flexibility is achieved by using Factory Factory (the creation mechanism itself is parameterized)

© 2002, S. Krakowiak

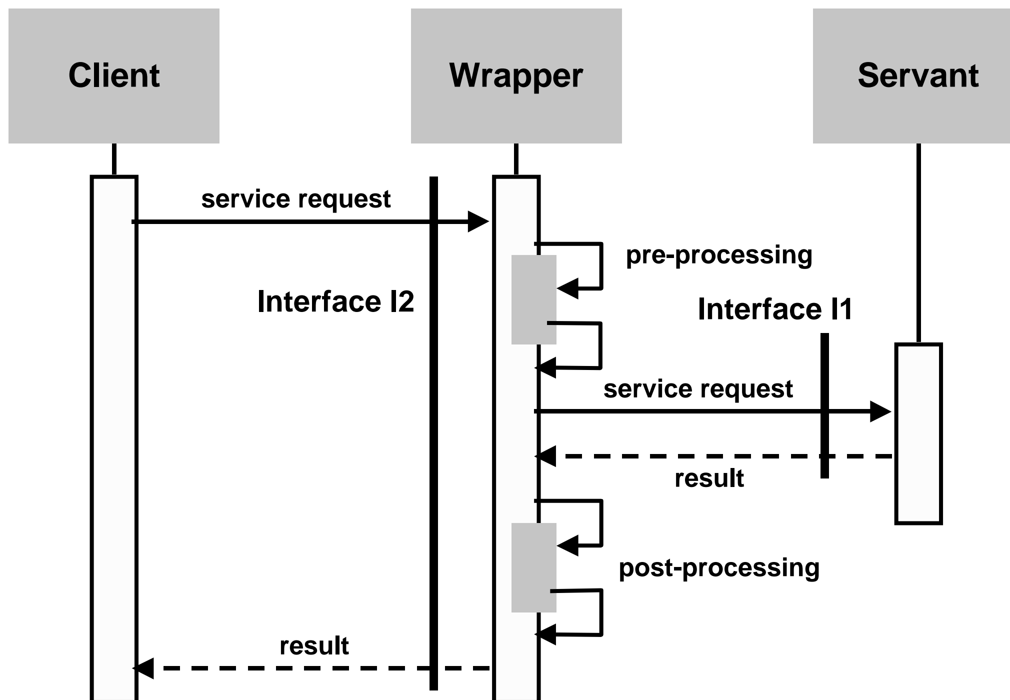
Factory in Use



Wrapper (a.k.a. Adapter)

- **Context**
  - Clients requesting services; servers providing services; services defined by interfaces
- **Problem**
  - Reuse an existing server by modifying either its interface or some of its functions in order to satisfy the needs of a client (or class of clients)
  - Desirable properties: should be run-time efficient; should be adaptable because the needs may change and may not be anticipated; should be itself reusable (generic)
  - Constraints:
- **Solutions**
  - The wrapper screens the server by intercepting method calls to its interface. Each call is prefixed by a prologue and followed by an epilogue in the wrapper.
  - The parameters and results may need to be converted.

Wrapper in Use



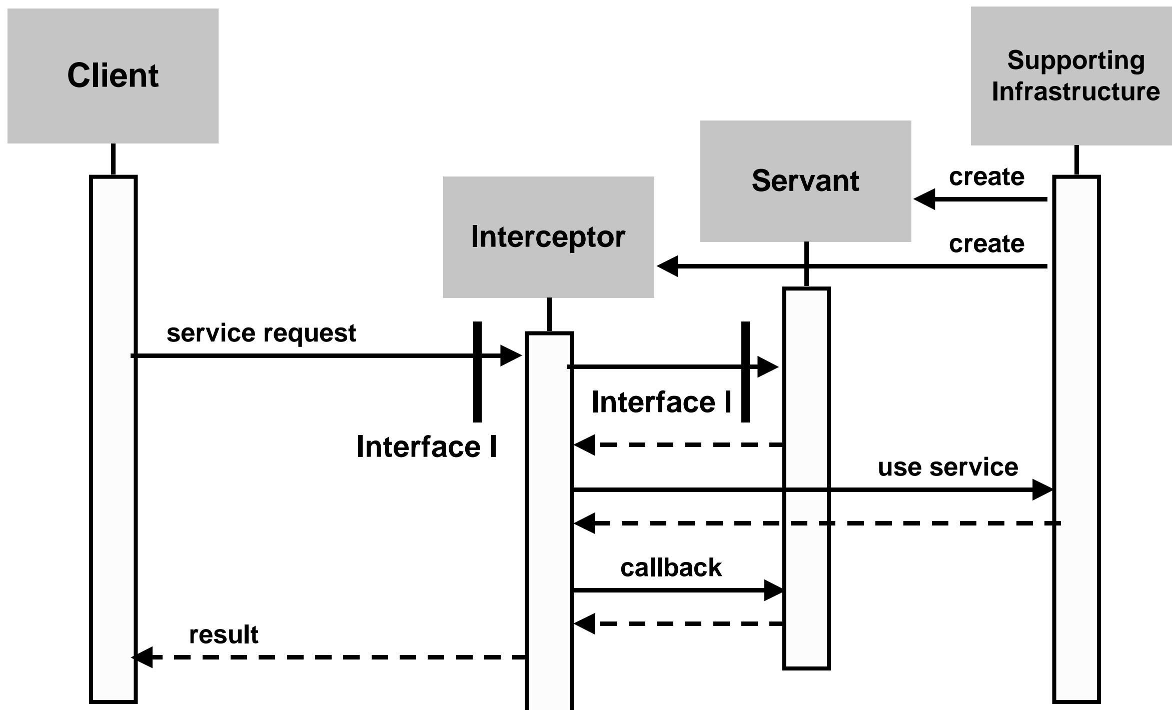
© 2002, S. Krakowiak

Interceptor

- **Context**
  - Service provision (in a general setting)
    - Client-server, Peer to peer, High-level to low-level
    - May be uni- or bi-directional, synchronous or asynchronous
- **Problem**
  - Transform the service (adding new functions), by different means
    - Interposing a new layer of processing (like wrapper)
    - Changing the destination (may be conditional)
  - Constraints
    - Services may be added/removed dynamically
- **Solutions**
  - Create interposition objects (statically or dynamically). These objects
    - Intercept calls (and/or returns) and insert specific processing, that may be based on contents analysis
    - May redirect call to a different target
    - May use callbacks

© 2002, S. Krakowiak

Interceptor in Use



Similarities and Differences between Patterns

- **Wrapper vs Proxy**
  - Wrapper and Proxy have a similar structure
    - Proxy preserves the interface; Wrapper transforms the interface
    - Proxy often (not always) involves remote access; Wrapper is ususally on-site
- **Wrapper vs Interceptor**
  - Wrapper and Interceptor have a similar function
    - Wrapper transforms the interface
    - Interceptor transforms the functionality (may even completely screen the target)

Frameworks for Middleware: a Case Study

From Patterns to Frameworks

Jonathan: A Framework for Middleware Construction

Examples and Code Walkthrough

- Naming & binding
- Communication
- Configuration

Software Frameworks

- **Definition**
  - A framework is a program "skeleton" that may reused (and adapted) for a family of applications
  - In object oriented languages: a framework consists of
    - A set of classes (often abstract) to be adapted (e.g. by overloading) to specific environments and constraints
    - A set of rules of usage for these classes
- **Patterns and frameworks**
  - Both are techniques for reuse
  - Patterns reuse design; frameworks reuse code
  - A framework implements a design pattern (usually several)

© 2002, S. Krakowiak

Jonathan, an Open Distributed Processing Environment

- Motivations
  - Need for a flexible environment, to be adapted to
    - Specific runtime constraints
    - Customized resource management policies
  - Lack of flexibility of existing distributed processing environments
    - Monolithic (lack of modularity)
    - Hard coded policies
- History
  - In 1996-98, developed by a group at France Télécom R&D (Bruno Dumant et al.)* as contribution to the ReTina project (for telecom applications)
  - In 1999-2000, became part of the ObjectWeb consortium (open source middleware), now hosted by INRIA
  - In 2001, creation of the Kelua company, which develops products derived from Jonathan

(*) B. Dumant, F. Dang Tran, F. Horn, and J.-B. Stefani. Jonathan: an open distributed processing environment in Java. In *Middleware'98: IFIP International Conference on Distributed Systems Platforms and Open Distributed Processing*, The Lake District, U.K., September 1998.

What Jonathan Provides

- A set of components from which an Object Request Broker (ORB) can be assembled
  - Buffer management
  - Activity management
  - Binding factories
  - Communication protocols
  - Marshallers and unmarshallers
- Configuration tools
  - Used to build a system from a set of selected components
- Instances of specific ORBs ("personalities")
  - Jeremie, a Java RMI personality
  - David, a CORBA personality

The Jonathan Frameworks

- **Binding**
  - Tools for managing names (identifiers) and developing binding factories, with different binding models (e.g. for QoS)
- **Communication**
  - Tools for developing and composing communication components (sessions, etc.) to implement various communication protocols
- **Resources**
  - Abstractions for the management of various basic resources
    - Processor (Threads)
    - Memory (Buffers)
    - Network (Connections)
- **Configuration**
  - Tools to create new instances of components and to assemble them to form specified configurations

Jonathan Design Principles

Achieve flexibility through separation of concerns and specialization

- **Separate abstractions from implementations**
  - Separate apis.* from libs.*
- **Independent modules implement specific services**
  - Non-functional issues (e.g. resource management) in separate service modules
  - A configurable kernel gives access to these services
  - Most users do not access service implementations (left to "system integrators")
- **A few generic programming patterns are used throughout**
  - Factory, FactoryFactory
  - Delegation, abstract classes
  - Helpers, holders

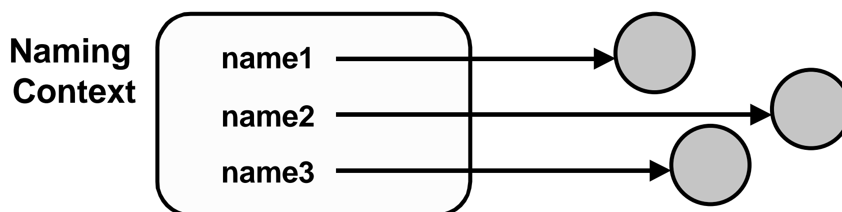
Naming (1)

- **Names**
  - A name is an information that designates an entity. It has two functions
    - Identification: to distinguish this entity from other entities
    - Reference: to provide a means of access to the entity
  - Example
    - A Java identifier identifies an object (an instance of a class)
    - The run time representation of the identifier is a reference (a pointer, invisible to the programmer) that gives access to the representation of the instance in the JVM
  - In Jonathan, the main named entities are interfaces (need for abstraction and separation of concerns)
  - Binding provides the link between identification and reference; these two notions are independent
    - To allow an interface to participate to bindings of different types
    - To allow an interface to be designated even if it cannot be accessed (because of failure, mobility)
    - To allow an interface to be transmitted in a binding, even if it cannot be part of a binding of this type

© 2002, S. Krakowiak

Naming (2)

- **A name is only valid in a naming context (a set of associations between names and entities)**

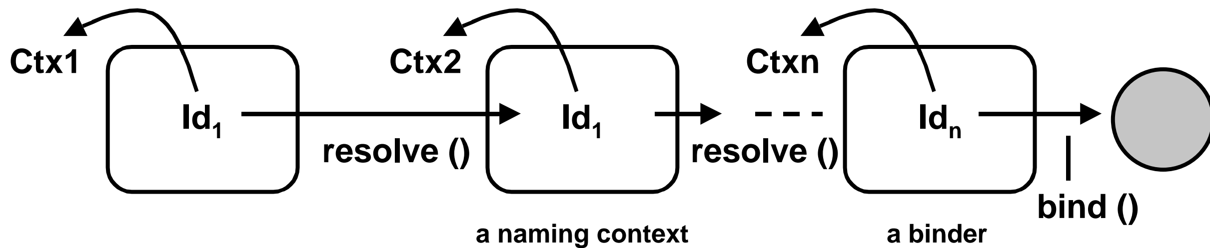


- **In Jonathan, basic forms of names are identifiers**
  - Plain identifiers
  - Session identifiers (in protocols)
  - A name may be composite (a chain of identifiers)
  - An identifier has a link to the naming context in which it was created

© 2002, S. Krakowiak

Naming (3)

- Resolving a name: finding the entity associated with the name (in a context)
- Names may be composite: name resolution may be an iterative process - analogy with names in file systems



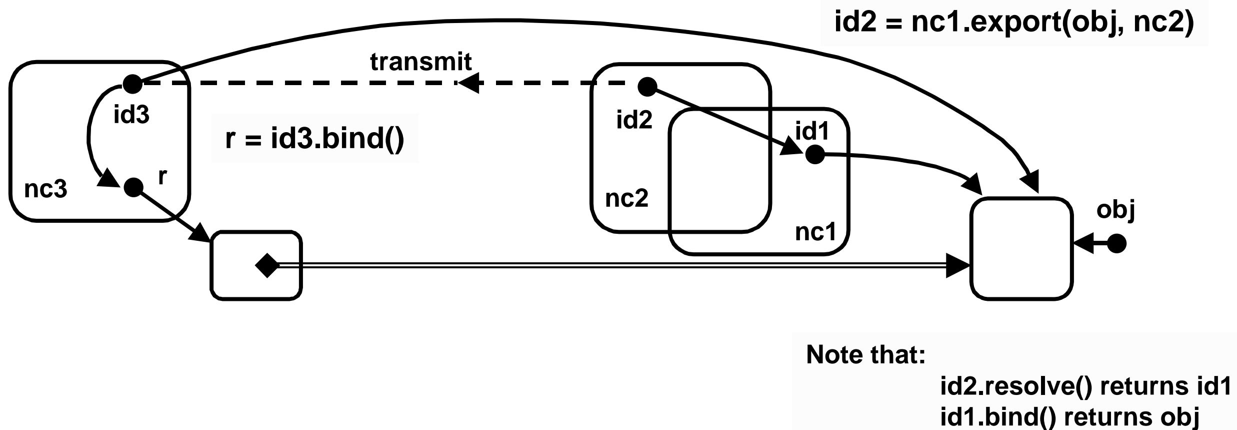
- In Jonathan:
  - resolve() goes down a chain of identifiers
  - bind() gets to the actual target at the end of the chain

Binding

- Definition
  - Binding is concerned with actual access (as opposed to identification)
    - Note the difference between naming and binding
      - Identifying an object is different from getting access to it
  - Binding is the process by which an object, the origin, gets access to another object, the target (it is also the result of this process)
    - Specific case: binding an identifier to an object that it designates
- A brief reminder about binding
  - There are many instances of bindings
    - Language level (by compiler + linker), system level, network level
  - Binding may occur at various times in the lifecycle of a system
    - Late (dynamic) binding adds flexibility (but usually at a price)
  - Just like naming, binding usually is a multi-stage process (binding chain)
  - In object systems
    - Binding an identifier to an interface
    - Binding an interface (used) to an implementation (provided)

Binding in Jonathan (1)

- **Binding is a 2-phase process**
  - ***export***: **the target object is "exported" to one (or more) naming context(s);** ***export*** **makes the object potentially accessible and usually prepares data structures to speed up the binding process**
  - ***bind***: **actual access is set up, possibly by building a local representative of a remote object**

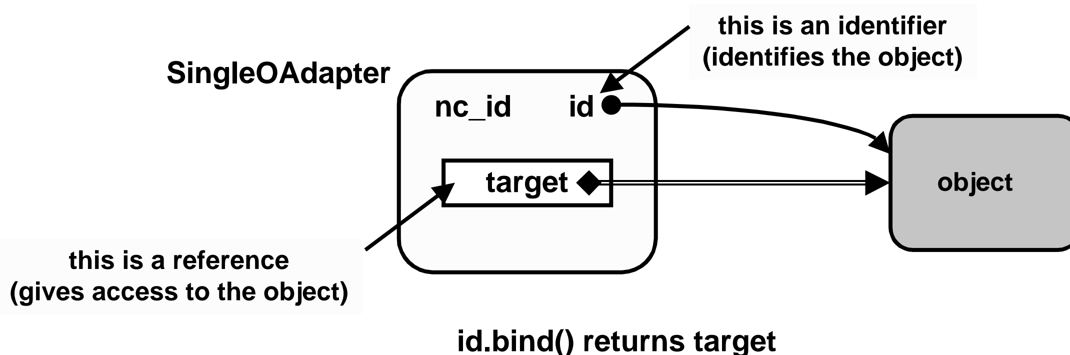


Note that:
id2.resolve() returns id1
id1.bind() returns obj

© 2002, S. Krakowiak

Binding in Jonathan (2)

- **Example**
  - **Single Object Adapter: a manager for a single object (the simplest possible example !)**
  - **Class SingleOAdapter**
    - **The target object is designated by an identifier *id***
    - **Only one object may be designated in the context**
    - **It may be exported to another naming context (with *nc_id*)**

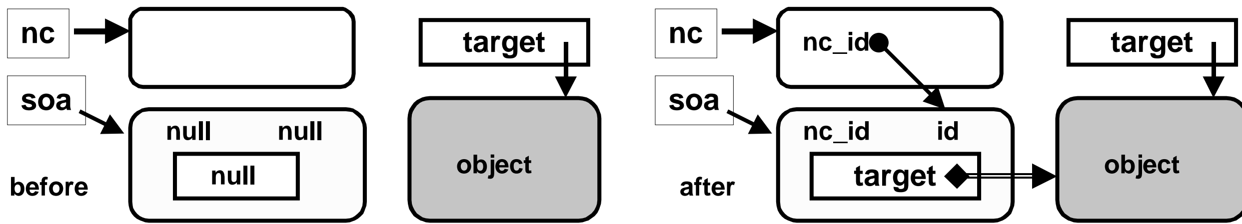


id.bind() returns target

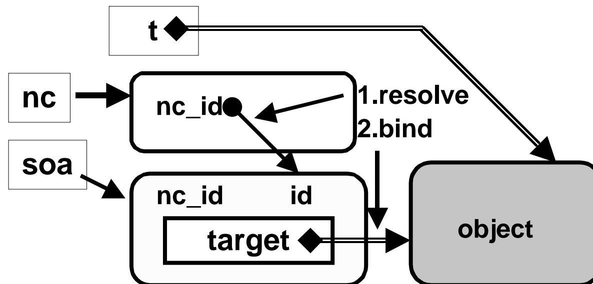
© 2002, S. Krakowiak

Binding Example: Single Object Adapter

nc_id ← soa.export(target, nc)



t ← nc_id.bind()

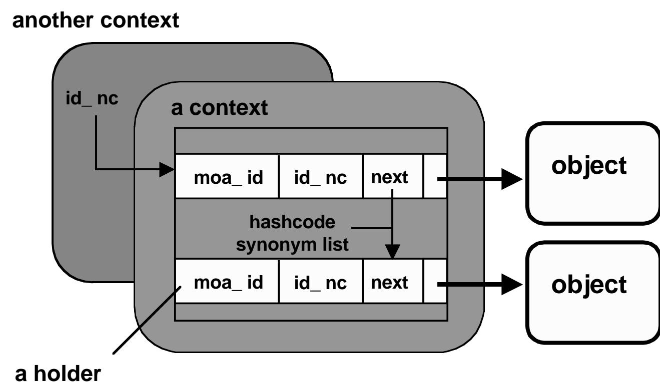


A fine point: an soa instance must be preserved from garbage collection as long as the object is being exported. To ensure this, a thread (waiter) is kept waiting on a lock.

Minimal Object Adapter

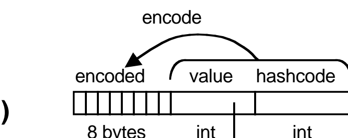
A table of Java objects
Implemented as a collection of holders, accessed by hashcode. Each holder contains a couple moa_id, ref to object) + last exported identifier. export creates a holder, bind tries to find a holder with a matching moa_id.

Optimizations.
There is a static free holder list, to be reused when a new object is exported (reduces creation overhead).
The table is rehashed and its size doubled when half full.
Again a waiter thread is kept alive as long as one holder is present.



MinimalAdapter

moa_ id (instance of `MOAIdentifier`)



More on identifiers

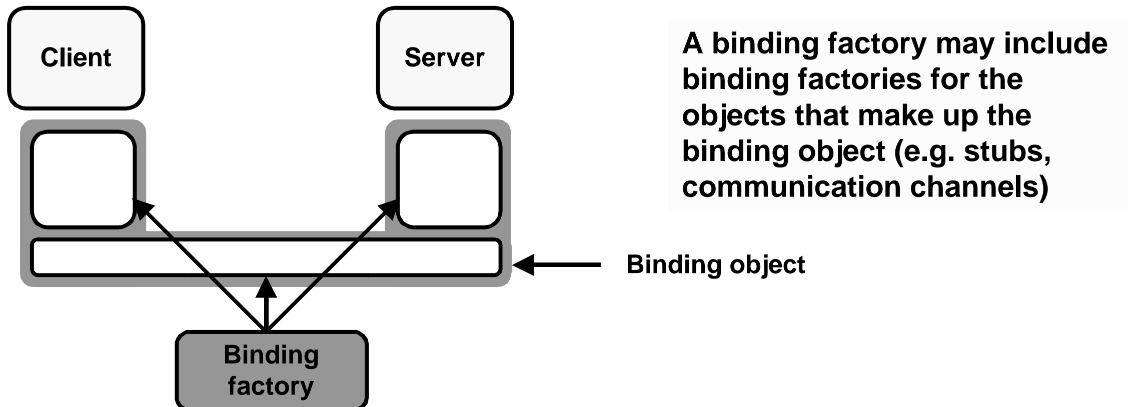
- Identifiers may need to be transmitted on a network
  - Therefore they need to be put into a suitable sequential form (a byte array)
  - id.encode() returns an encoded form for id; the algorithm depends on id's naming context
  - nc.decode(encoded) returns a previouly encoded id
    - Note that you need to know the naming context. This is usually provided for by attaching a label to the encoding, and associating this label with a context. More on this in the configuration framework

The Identifier and NamingContext Interfaces

- **Summary** (with some approximations)
  - id = nc.export(obj)
    - obj may be an identifier or a direct reference (address)
    - if obj is an identifier, id.resolve() returns obj
  - id.bind() returns a handle to obj, which may be
    - obj itself (a direct reference)
    - A representative for obj (a proxy)
  - id.unexport() cancels the effect of export (id.bind () will fail)
  - id.encode() returns encoded form of id suitable for transmission
  - if nc=id.getContext(), then nc.decode(id.encode()) returns id
  - id.isvalid() returns false if id can be neither bound not resolved

Binding Objects and Binding Factories

- **Bindings may take various forms**
  - In a single address space: a reference (a pointer, a memory address)
  - Between different address spaces: stubs, communication channel
  - It is convenient to think of a binding as an object ("binding object")
  - Then this object has to be constructed by a "binding factory"



A binding factory may include binding factories for the objects that make up the binding object (e.g. stubs, communication channels)

A very schematic outline of *export* & *bind*

```
[in implementation of binding factory]

Includes
   a context table
   the definition of a class of identifiers
              (implements Identifier)

Identifier export (Object obj, Context hints){
  create new identifier id
       (using binder's identifier class)
  create new element in context table with (id, obj)
  possibly export obj to other context
  possibly create new session
  return id (or last identifier created)
  }
```

```
[in implementation of Identifier]

Object bind (parameters) {
  case of
    local object:
      lookup target identifier in context table;
      if (found)
          {return associated object}
    remote object:
      determine session from target identifier
          (or create it if needed)
      create stub with session and parameters
          (using stub factory)
      return stub
  }
```

Jeremie, the Java RMI personality of Jonathan

- **Main components**
  - Binding
    - the binding factory: JIOP
    - adapters: single object, multiple objects
  - The IIOP protocol (GIOP over TCP-IP)
    - GIOP protocol
    - IIOPBinder
  - Stub factories
    - specialized for method call
  - Presentation
    - marshallers and unmarshallers
  - The naming server (Registry)
    - may reside on a different node from the server

© 2002, S. Krakowiak

A Use Case of Jeremie: *Hello World*

Example (centralized)

Interface

```
public interface Hello {

   String sayHello();
};
```

Implementation

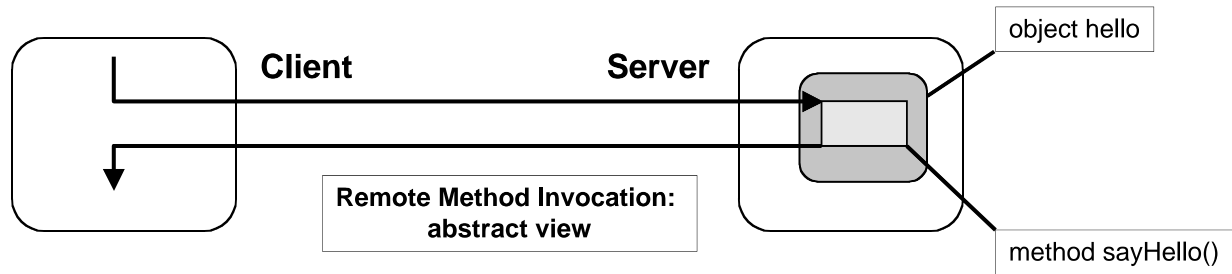
```
class HelloImpl implements Hello {

   HelloImpl() {
   }
   public String sayHello() {
      return "Hello World!";
   }
}
```

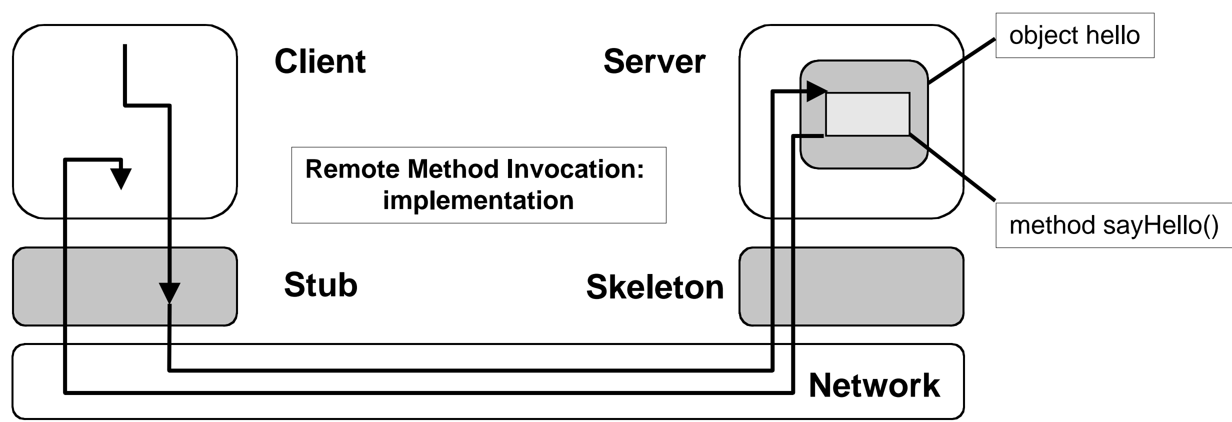
Use

```
...
Hello hello = new HelloImpl ();
...
hello.sayHello();
...
```

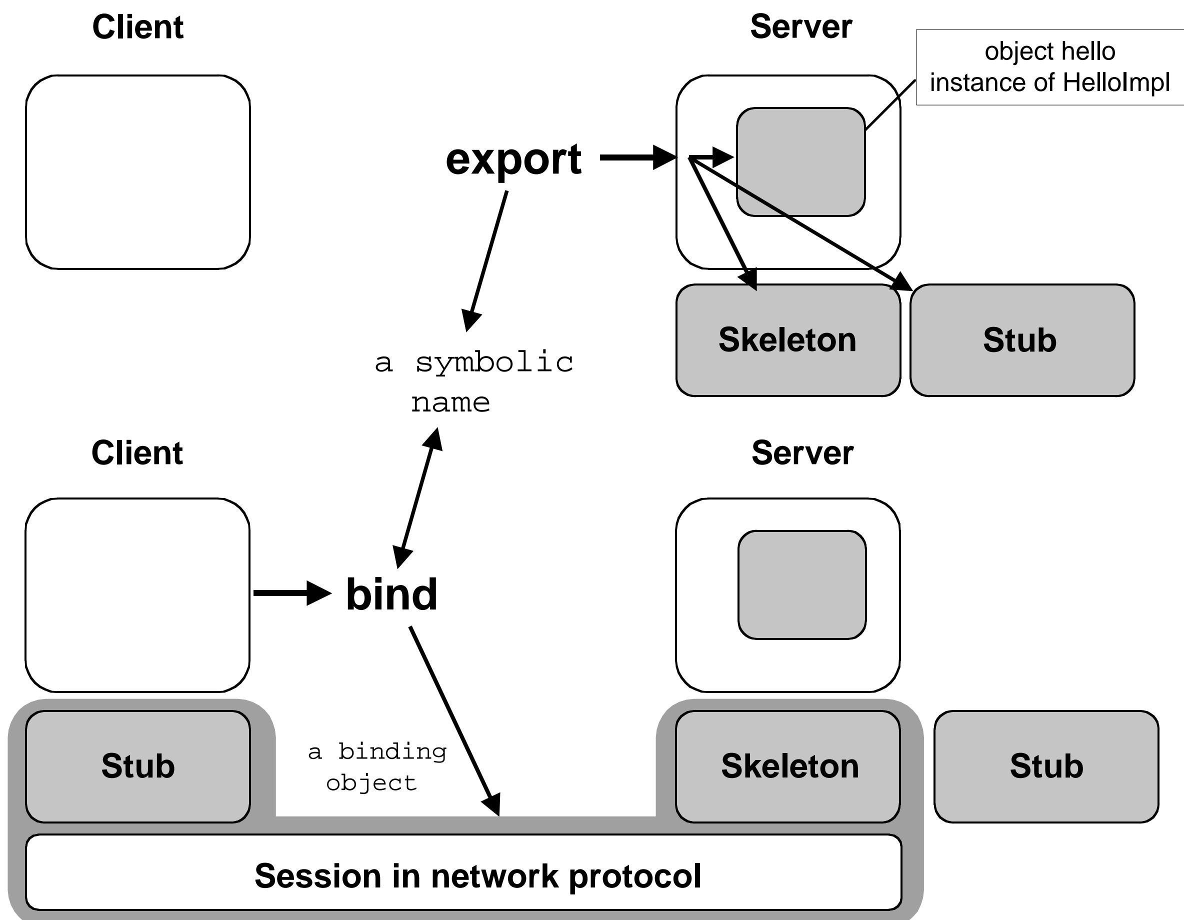
© 2002, S. Krakowiak

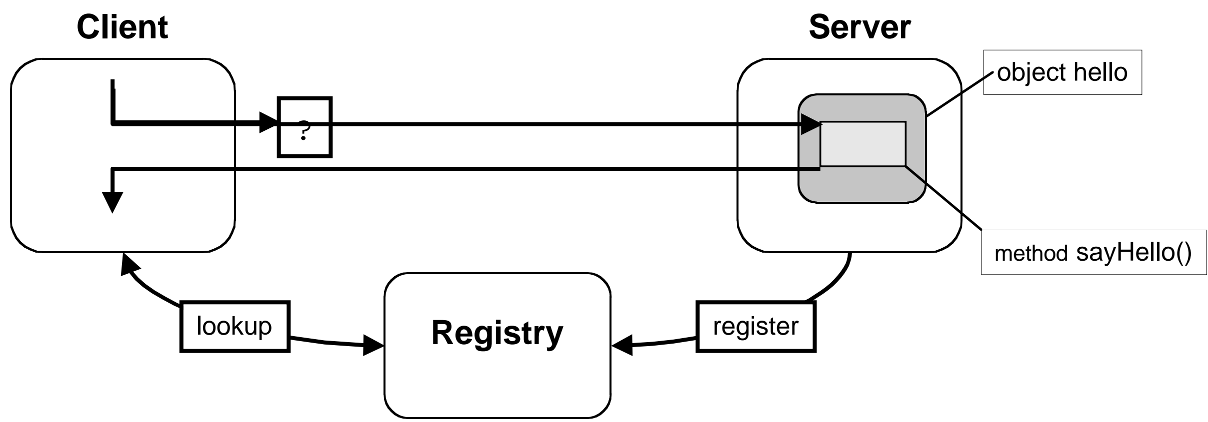
Remote Invocation

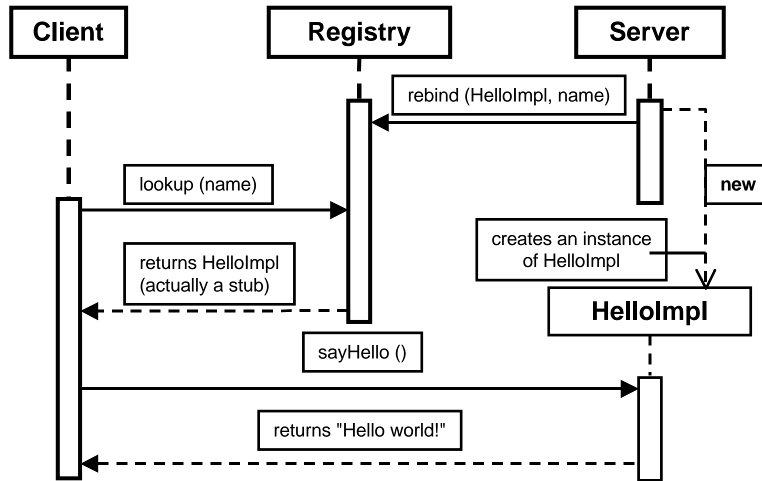




© 2002, S. Krakowiak







JRMIRegistry
main
creates an instance of RegistryImpl, RegistryImp _Stub and RegistryImpl_Skel
new
Server
Naming
Naming
LocateRegistry
main
Client
local interaction
RegistryImpl
rebind (HelloImpl, name)
getRegistry
new
local interaction
return stub for RegistryImpl
main
LocateRegistry
lookup (name)
getRegistry
RegistryImpl_Stub
return stub for RegistryImpl
rebind (HelloImpl, name)
RegistryImpl_Stub
remote interaction
lookup (name)
remote interaction
return stub for HelloImpl
return stub for HelloImpl
creates an instance of HelloImpl, HelloImpl_Stub and HelloImpl_Skel
HelloImpl
HelloImpl_Stub
sayHello()
remote interaction
"Hello World!"

Server-side initialization (export)

HelloImpl
extends
UnicastRemoteObject
HelloImplt()
new
UnicastRemoteObject()
super()
creates an instance impl of HelloImpl
MOAContext
export (impl, port)
export0bject(impl, binder, port)
JIOP
IIOPBinder
getStubFactory
getStubFactory
returns an instance of StdStubFactory
an instance of StdStubFactory
StdStubFactory
newRequestSession (impl)
MinimalAdapter
creates an instance of HelloImpl_Skel
RequestSession session
export (session, port, binder)
creates identifier moa_id
export (moa_id)
export (moa_id)
SrvIdentifier id
SrvIdentifier id
SrvIdentifier id
newStub(impl, id)
creates an instance of HelloImpl_Stub
RemoteStub stub
stub
stub

JRMIRegistry
main
creates an instance of RegistryImpl, RegistryImp _Stub and RegistryImpl_Skel
new
Server
Naming
Naming
main
LocateRegistry
rebind (HelloImpl, name)
new
Client
local interaction
RegistryImpl
getRegistry
return stub for RegistryImpl
local interaction
LocateRegistry
main
lookup (name)
getRegistry
RegistryImpl_Stub
return stub for RegistryImpl
rebind (HelloImpl, name)
RegistryImpl_Stub
remote interaction
lookup (name)
remote interaction
creates an instance of HelloImpl, HelloImpl_Stub and HelloImpl_Skel
return stub for HelloImpl
return stub for HelloImpl
HelloImpl
HelloImpl_Stub
sayHello()
remote interaction
"Hello World!"

Binding to Registry

Server

new HelloImpl ( )

creates an instance of HelloImpl, HelloImpl_Stub and HelloImpl_Skel (details on previous Figure)

main

LocateRegistry

Naming

JIOP

IIOPBinder

rebind (HelloImpl, name)

getRegistry (host, port)

bind (" RegistryImpl", host, port)

newId (host, port, key)

return new CltIdentifier

IIOPBinder.CltIdentifier

bind (…)

bind (…)

GIOPProtocol

newSessionIdentifier(…)

cltsession_id

StdStubFactory

newStub (cltsession_id, …)

return stub for RegistryImpl

stub

stub

stub

return stub for RegistryImpl

RegistryImpl_Stub

RegistryImpl

rebind (HelloImpl, name)

rebind (HelloImpl, name)

(remote interaction)

RegistryImpl_Stub

ObjectOutputStream

ObjectOutputStream

StdMarshaller

supersedes

RefImpl

JDomain

rebind (HelloImpl, name)

writeValue

writeObject

writeExternal

encode

return encoded

Stream

write

activated by message receipt

RegistryImpl _ Skel

ObjectInputStream

ObjectInputStream

StdUnmarshaller

supersedes

RefImpl

send

readValue

readObject

readExternal

read

JDomain

decode

return decoded

readObject calls readResolve on returned stub

StdStub

IIOPBinder

StdStubFactory

bind

readResolve

newStub (cltsession_id, …)

callsRegistry.rebind with returned stub

returns stub

returns stub

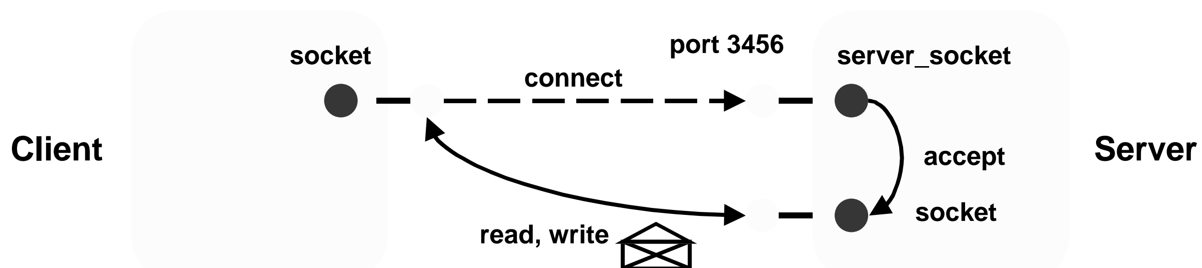
returns stub

The Jonathan Communication Framework

Introduction: Another View of Java Sockets

```
Socket socket=new Socket(server, 3456)
```

```
server_socket=new ServerSocket(3456)
Socket socket=server_socket.accept()
```

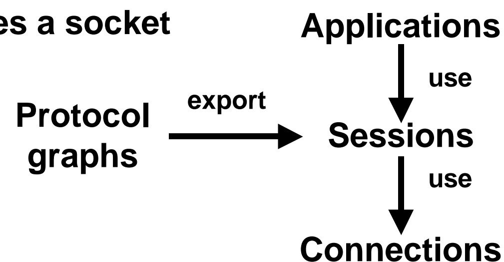


- ServerSocket is a socket factory (it creates communication sockets)
- *accept* is equivalent to *export* (it sets up structures for connections and waits for *connect*)
- *connect* is equivalent to *bind* (it sets up the actual connection, to be used for message exchange)

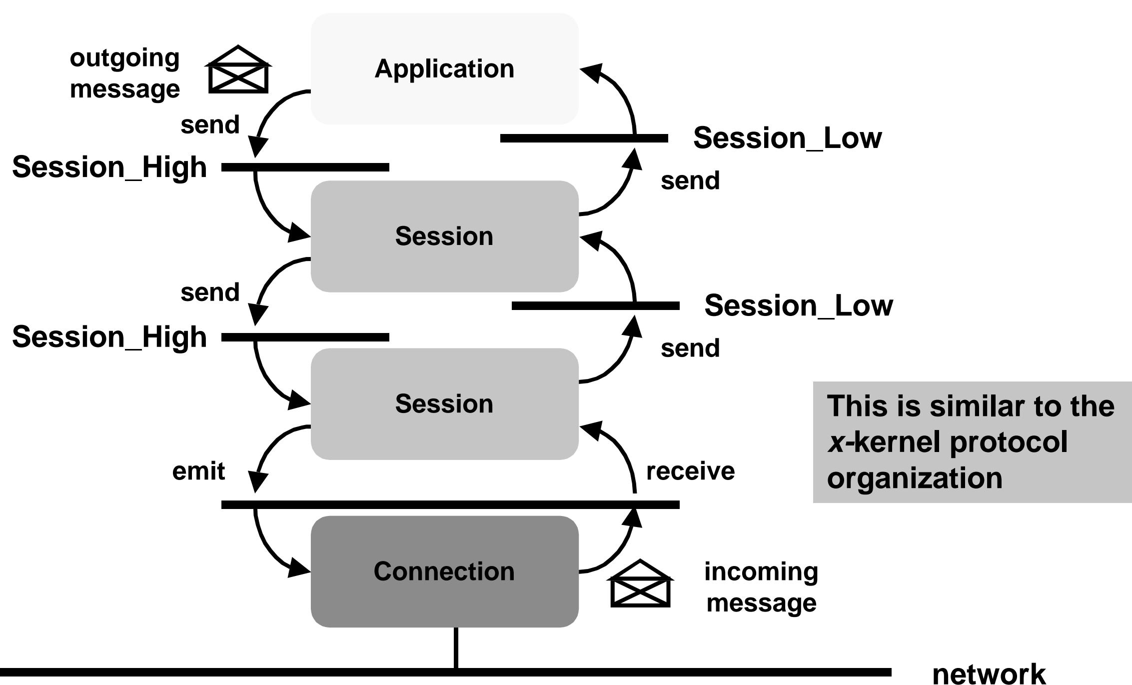
© 2002, S. Krakowiak

Communication in Jonathan

- **Communication relies on 3 main entities**
  - Protocols: a protocol is a naming context and factory for sessions
    - A protocol graph combines several protocols (in a stack or acyclic graph) and exports sessions
  - Sessions: a session is a communication channel fo sending and receiving messages (the main abstraction used by applications)
    - Two interfaces, *Session_Low* and *Session_High* to send messages up and down the protocol layer
  - Connections: a connection is the basic communication mechanism provided by the OS and network
    - A connection encapsulates a socket



Sending and receiving Messages



This is similar to the *x*-kernel protocol organization

Managing Connections

class JConnectionMgr
  manages connections for an application
  newSrvConnectionFactory(int port)
   returns new server connection factory encapsulating a server socket on *port*

  newCltConnection(String host, int port, IPSession session)
   returns a client connection encapsulating a socket, assigns it to session

← **Used by a server to receive service requests** (newSrvConnectionFactory)

← **Used by a client to connect to a server** (newCltConnection)

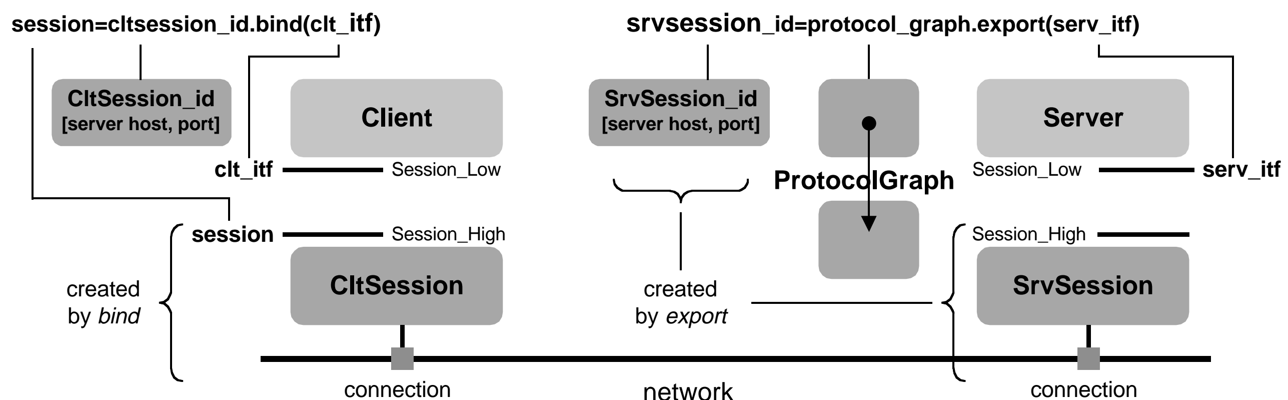
delegation ↓

class IPv4ConnectionFactory
  implements connections using sockets
  class Connection
   public void emit(Chunk c)
    sends message
   public void receive(Chunk c, int sz)
    receives message

Implementation is normally invisible to the applications; may be reimplemented without modifying JconnectionMgr

© 2002, S. Krakowiak

Session Setup

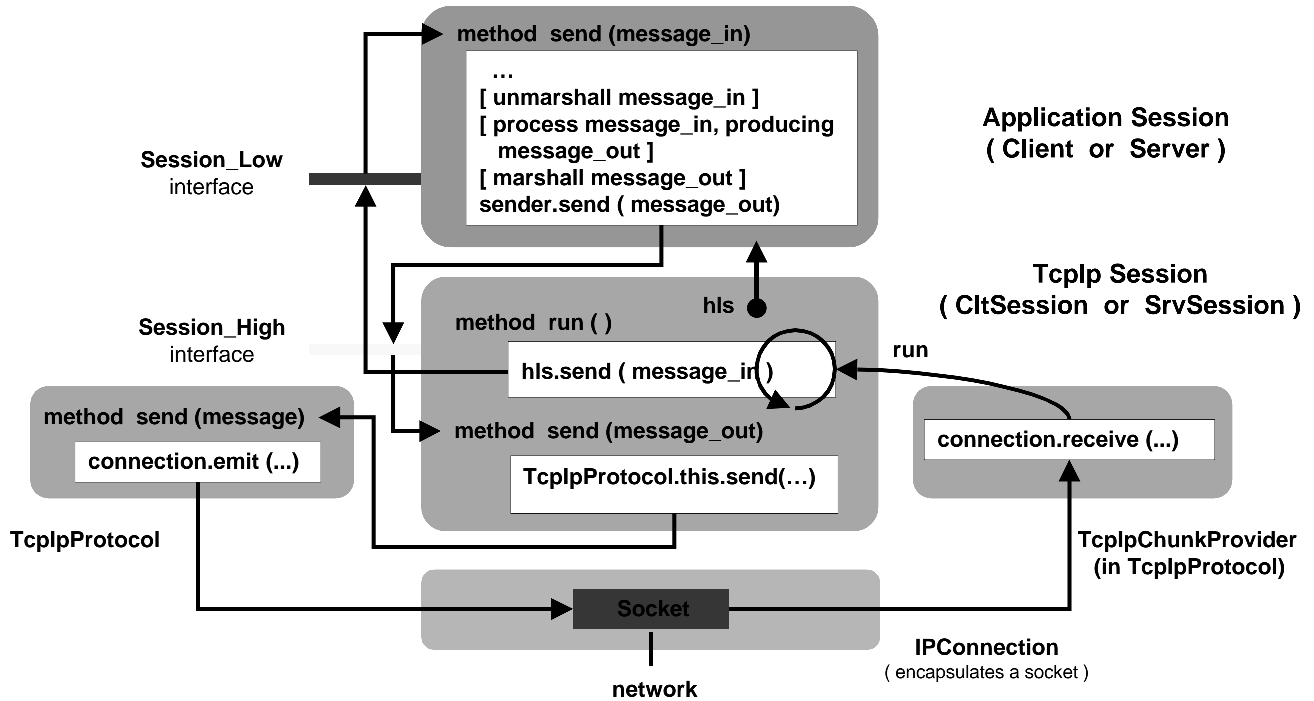


- **After binding:**
  - **the Client application can use *CltSession* to send messages to the server; the Server application can use *SrvSession* to send messages to the client**
  - **both Client and Server can receive messages on their own *clt_itf* or *srv_itf* interface**

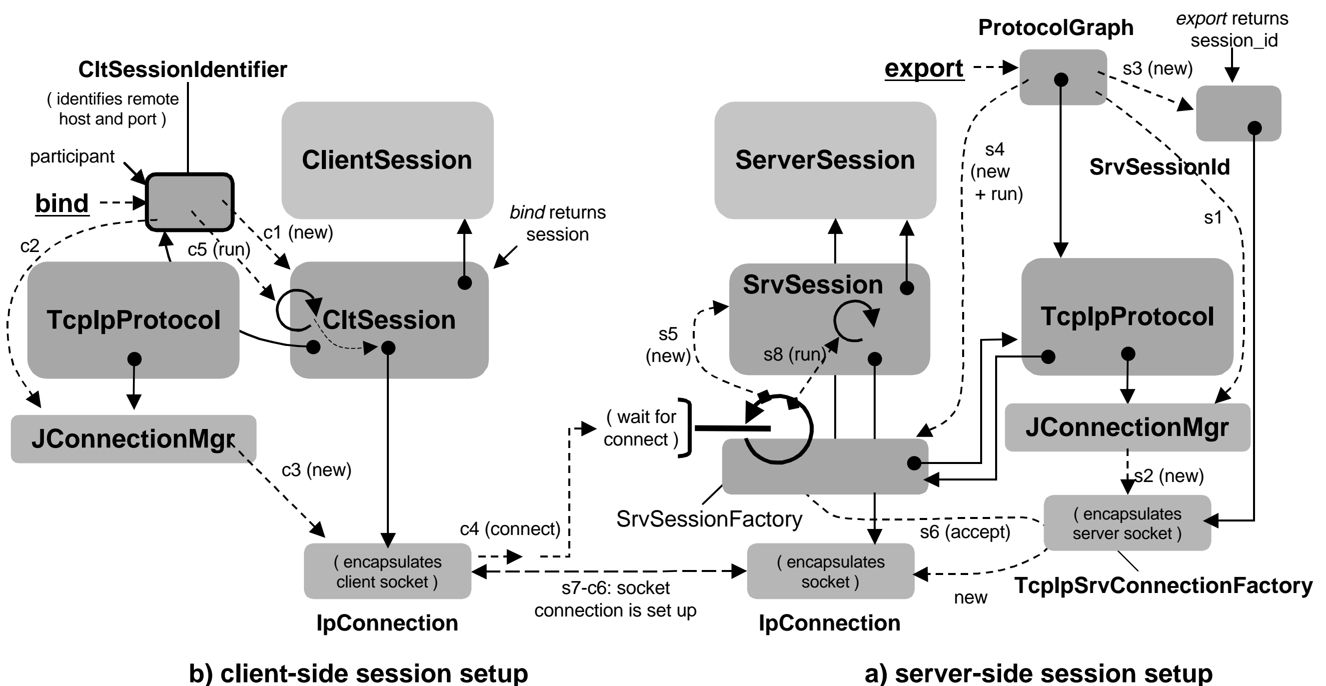
© 2002, S. Krakowiak

The TCP-IP protocol in Jonathan

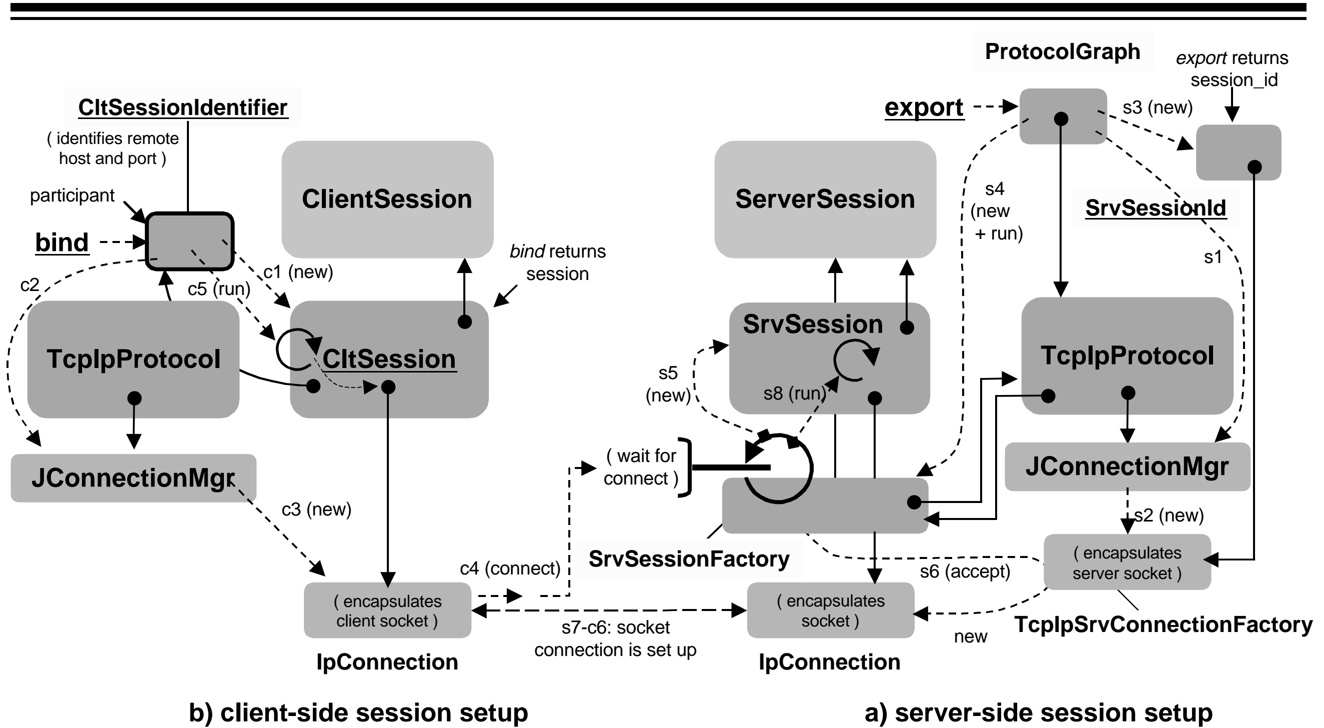
◆ A common pattern for sessions (both *CltSession* and *SrvSession*)



TCP-IP Protocol: a Global View

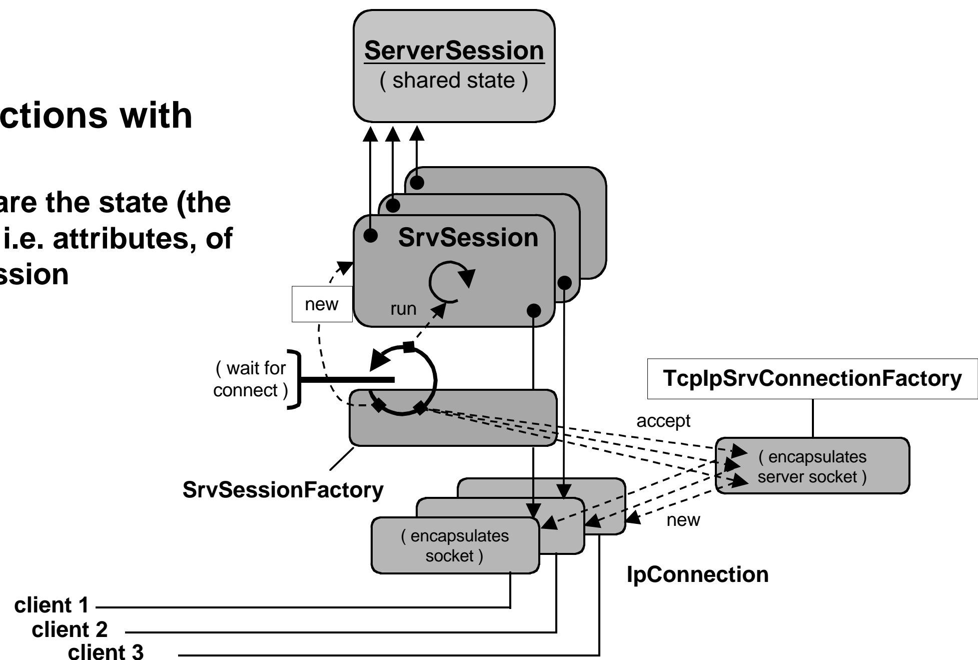


TCP-IP Protocol: a Global View

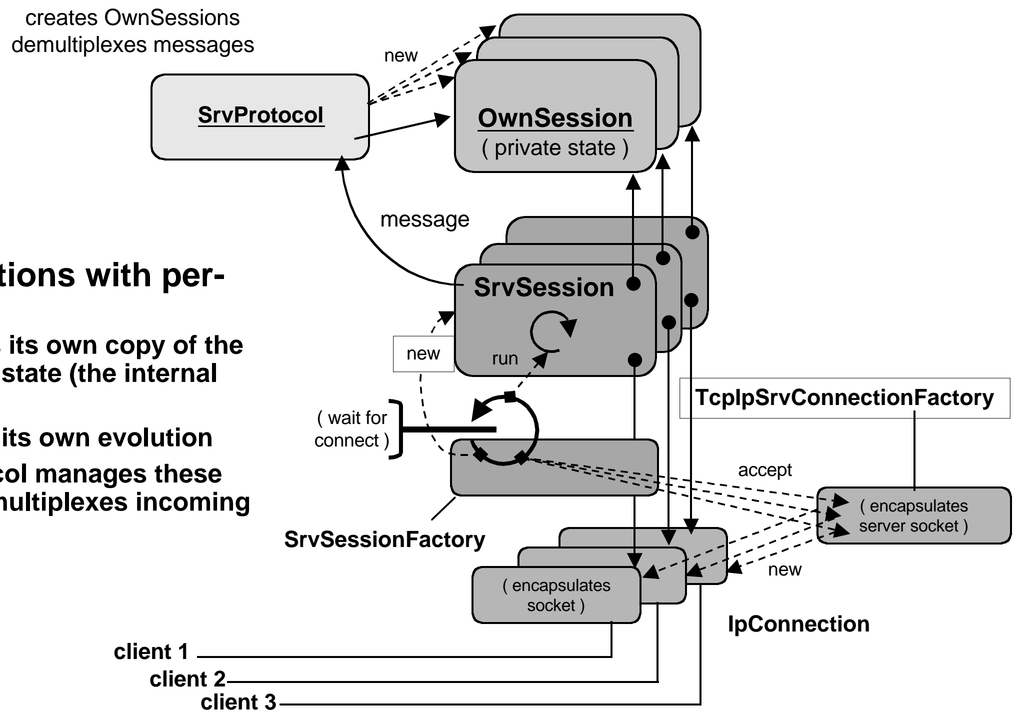


Serving Multiple Clients

- **Multiple connections with shared state**
  - **All clients share the state (the internal data, i.e. attributes, of the server session**



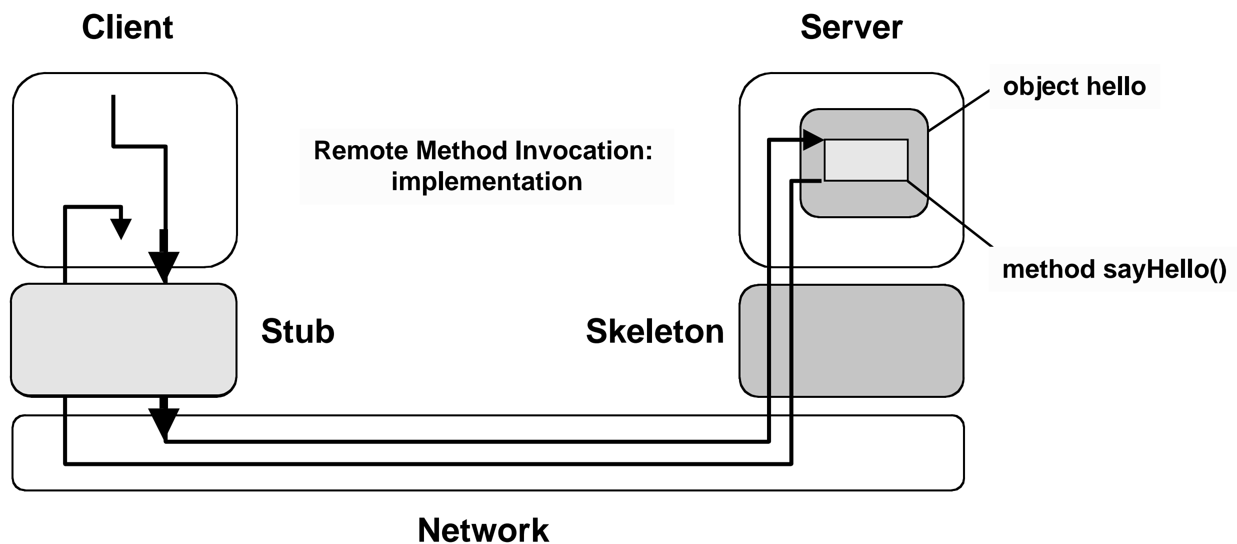
Serving Multiple Clients



- Multiple connections with per-client state
  - Each client has its own copy of the server session state (the internal data)
  - Each copy has its own evolution
  - A server protocol manages these copies and demultiplexes incoming messages

Back to the Hello World Example

Let's take a look at communication



java.io.Serializable

java.io.Externalizable

java.rmi.Remote

org.objectweb.jonathan.apis.bindings.Reference

java.rmi.server.RemoteRef

java.rmi.server.RemoteObject

org.objectweb.jeremie.apis.bindings.JRMIRef

java.rmi.server.RemoteStub

org.objectweb.jeremie.apis.bindings.JRMIOptRef

org.objectweb.jeremie.libs.stub_factories.std.OptStub

org.objectweb.jeremie.libs.stub_factories.std.RefImpl

HelloInterface

```
...
public String sayHello() throws RemoteException {
      …
      marshaller.writeInt(0); // # of the method
      optref .invoke(marshaller);
      …
}
```

Hello_Stub

Stub

Key

also valid for following figures

(grey box)	instance	(arrow)	instance of
(box)	class	(arrow)	implements
(rounded box)	interface	(double arrow)	extends

...jonathan.apis.protocols.Session_High

org.objectweb.jeremie.libs.stub_factories.std.RefImpl

Hello_Stub

```
optref .invoke(marshaller)
```

client stub

...david.libs.protocols.giop.GIOPSession_High

...giop.ClientSession_High

class RefImpl

```
transient protected Session_High session;

public void invoke(Marshaller marshaller) {
  try {synchronized(this){
    if (session == null) throw new MarshallException
          ("null session");
    }
    session .send(marshaller);
  } catch (Exception e) {
    throw new MarshallException ("invocation exception", e);
  }
}
```

..jonathan.apis.protocols.ip.IpSession

...protocols.tcpip.TcpIpProtocol.Session

class GIOPSession_High

```
Session_High lower ...
void send(Marshaller marshaller) throws JonathanException {
      sendMessage(marshaller, lower); }
...
void sendMessage(Marshaller marshaller, Session_High lower) {
      ...
      lower .send(marshaller);
}
```

...TcpIpProtocol.CltSession

continues on next flip

...jonathan.apis.protocols. ip.IpConnection

...jonathan.libs.resources.tcpip.IPv4ConnectionFactory.Connection

lower .send(marshaller);

...tcpip.TcpIpProtocol.CltSession

class TcpIpProtocol.CltSession

```
IpConnection connection ...
void send(Marshaller message, IpConnection connection) {
    Chunk portion;
    // fill up portion with contents of message
    connection .emit(portion)

      final class CltSession extends Session
      public final void send(Marshaller message);

            TcpIpProtocol.this.send (message, connection);
```

java.io.OutputStream

implementation of *sockets* provided by operating system

socket

class IPv4ConnectionFactory.Connection

```
class Connection implements IpConnection {

     OutputStream os;

     protected Connection(Socket socket,  ) {
          os = socket.getOutputStream();
     }
     public void emit(Chunk c) throws IOException {
          synchronized (os) {
           os .write(c.data, c.offset, c.top);
          }
     }
```

Client

Marshaller

HelloImpl_Stub

RefImpl

Session_High

IPConnection

extends

TcpIpProtocol. CltSession

extends

GIOPSession_High

IPv4ConnectionFactory. Connection

sayHello ()

writeInt(0)

invoke

send

send

emit

ObjectOutputStream

write

Socket

write

Network

Session_Low

HelloImpl_Skel

TcpIpProtocol. SrvSession

HelloImpl

activated by message receipt

Socket

send

UnMarshaller

TcpIpChunkProvider

ObjectInputStream

readInt

prepare

readInt

read

run

sayHello ()

The Jonathan Configuration Framework

Configuration Management

"Configuration Management is the process of identifying and defining the items in the system, controlling the change of these items throughout their lifecycle, recording and reporting the status of items and change requests, and verifying the completeness and correctness of items" [IEEE Std-729-1983

- Configuration management is increasingly important
  - Because systems are getting larger and more complex
  - Because systems become more and more adaptable
    - to changing user needs
    - to changing environments and operating conditions
- New approaches
  - Related to
    - component models
    - composition and configuration languages
  - A case study: the Jonathan configuration framework

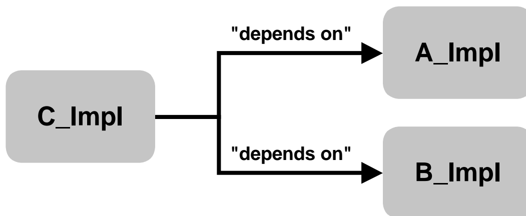
The Static Configuration Problem

```
public class C_Impl implements C {
        A field_A ;
        B field_B ;
        …

        field_A = new A_Impl() ;
        field_B = new B_Impl() ;
}
```

```
public interface  A {
        …
}
public interface B {
        …
}
public interface C {
        …
}
```

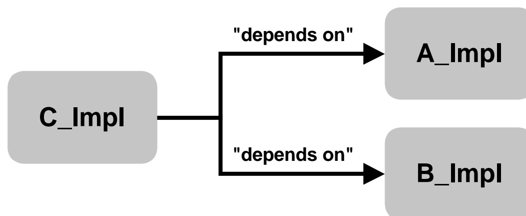
```
public class A_Impl implements A {
        …
}
public class B_Impl implements B {
        …
}
```

What if we change the implementation of A to A_Impl1 ?
We have to modify the code of class C_Impl

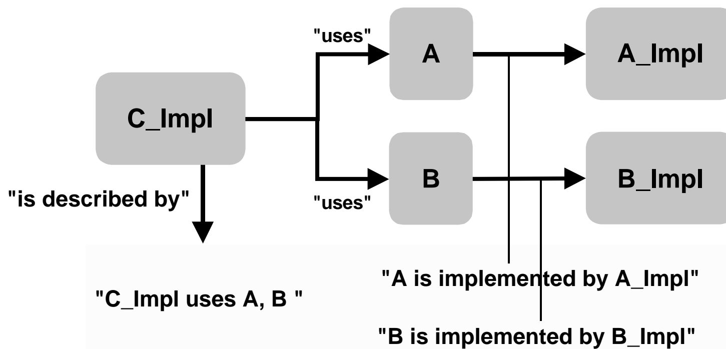


We need a description of the dependency between C_Impl, A_Impl and B_Impl, separate from the code

Describing a Configuration



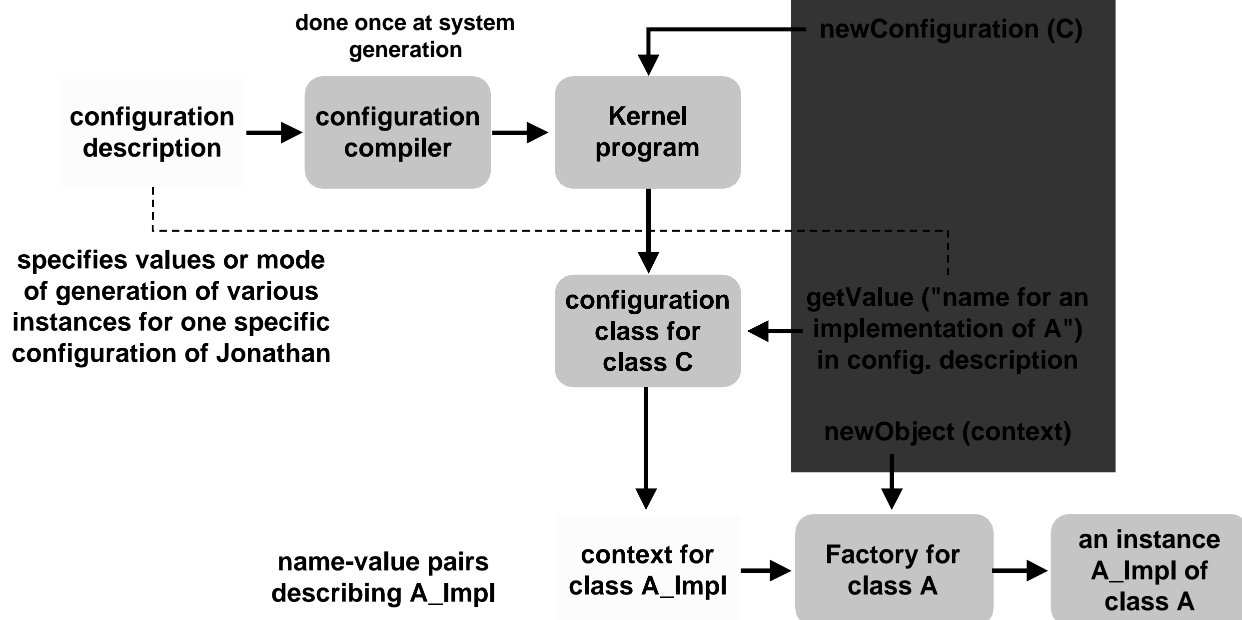

```
public class C_Impl implements C {
        A field_A ;
        B field_B ;
        …

        field_A = new A_Impl() ;
        field_B = new B_Impl() ;
}
```



configuration description

Changing the implementation of A or B means changing the configuration description, not the code of C_Impl

Configuration in Jonathan



Configuration Description in Jonathan

```
[/]
 [jonathan]
   ...
  [IPv4ConnectionFactory]        -- a configuration
       factory =>
          org.objectweb.jonathan.libs.resources.tcpip.
             IPv4ConnectionFactoryFactory    --  an atom
             instance => {factory, .}        -- an assemblage
             verbose -> /jonathan/tcpip/verbose -- an alias
        ...
  [JDomain]
     [binders]
         0 -> /david/orbs/iiop/instance
         org.objectweb.david.libs.binding.orbs.iiop.IIOPORB
           =>(int.class, 0)                 -- a property
     ...
  [tcpip]
     verbose => (Boolean.class, false)   -- a property
     ...
 [jeremie]
     ...
   [jiop]
     factory =>
        org.objectweb.jeremie.libs.binding.jiop.JIOPFactory
     instance => {factory, .}
     Chunkfactory -> /jonathan/JChunkFactory/instance
     ...
```

The actual description is in XML

A configuration [conf] is described by four types of elements

Atom: description of a class
name => class

Property: associates a type and a value
name => (type, value)

Assemblage: associate a factory and a configuration
name => {fact, conf}

Alias: an alternative name for an existing element
name -> target

getValue() uses these descriptions to generate actual values for attributes

An Example

Class LocateRegistry uses a specific instance of class JIOP

In the static initializer of LocateRegistry, we find the following code sequence:

```
$initial_context =
    (Context) Kernel.newConfiguration(LocateRegistry.class).getValue("/jeremie/jiop", '/');
binder = (JIOP) (new JIOPFactory()).newObject($initial_context);
```

Creating a specific instance is a two-step process:

- Create a context describing the instance
  - create a configuration using the configuration description
  - get the parameters corresponding to the created class
- Use a factory to create the instance
  - create an instance of the class-specific factory
  - create an instance of the class using the new context

Configuration in Jonathan (1) : Contexts

A context is a set of elements (typed objects), each identified by a name. Example: a tree context (structured as a tree)

The getValue() method builds a context, starting from a configuration description (another context). Its effect depends on the type of the element in the description:

- A property element (e.g. (Integer.class, 123))
  - return its value
- An atom element (e.g. class name)
  - load the class, then create an instance of it
- An assemblage element (e.g. (factory, context))
  - find factory and context, then create an instance using factory.newObject(context)
- An alias element (e.g. name -> target)
  - apply getValue() to the target element of the alias

Configuration in Jonathan (2) : Factories

```
public abstract class GenericFactory implements Factory {
        public Object newObject (Context _c) {…
            $comps = getUsedComponents (_c) …
            return newInstance (_c, $comps) }
        abstract protected Object[] getUsedComponents (Context _c);
        abstract protected Object newInstance (Context _c, Object[]used_components);
}
public class JIOPFactory extends GenericFactory {      // an example of specific factory
        final protected Object[] getUsedComponents(Context _c) {
            used_components[0] = _c.getValue ("ChunkFactory", (char) 0);
            used_components[1] = …}
        final protected Object newInstance (Context _c, Object[] used_components) {
            return new JIOP(_c, used_components); }
}
public class JIOP … {
        JIOP(Context c, Object[] used_components) … {
            super();
            initialize(c, used_components); }
        protected void initialize (Context c, Object[] used_components) {
            ChunkFactory chunk_factory= (ChunkFactory) used_components[0];
            … }
}
```

© 2002, S. Krakowiak

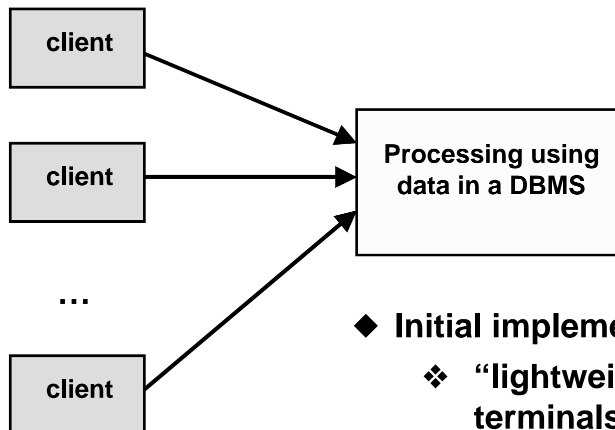
From Objects to Components

Motivations for Components

A Component-Based Middleware: Enterprise Java Bean (EJB)

Motivation for components: Evolution of client-server architectures

- **Basic scheme for an application**
  - A client application applies a specific processing to permanent data (DBMS) and displays the results

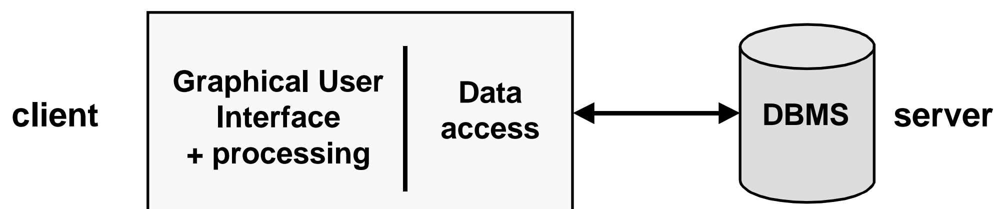


- Initial implementation
  - "lightweight" clients (alphanumeric terminals)
  - processing an data access are intertwined

© 2002, S. Krakowiak

Evolution of client-server architectures (2)

- **2-tier Architecture**
  - Used in simple situation in which all processing may be done on the client site.

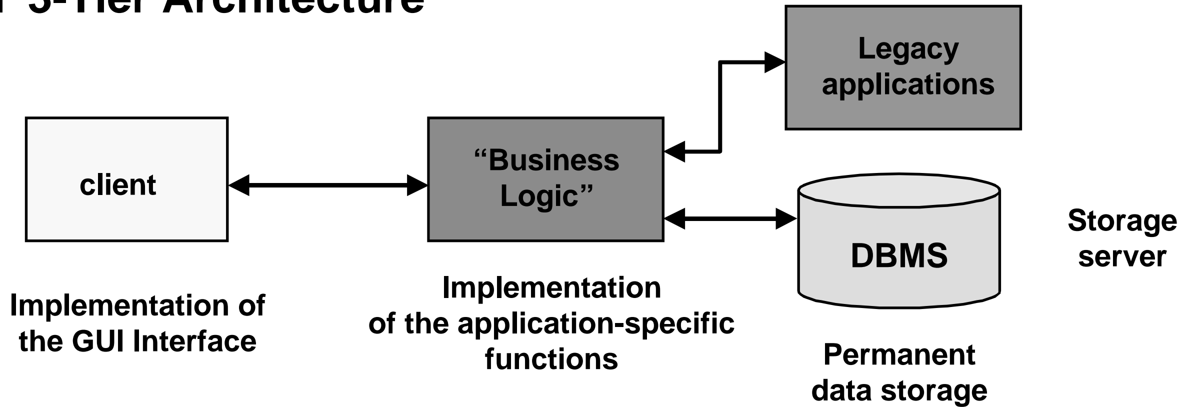


- Limitations
  - Processing power on the client site
  - Data sharing
  - Scalability
  - Evolution

© 2002, S. Krakowiak

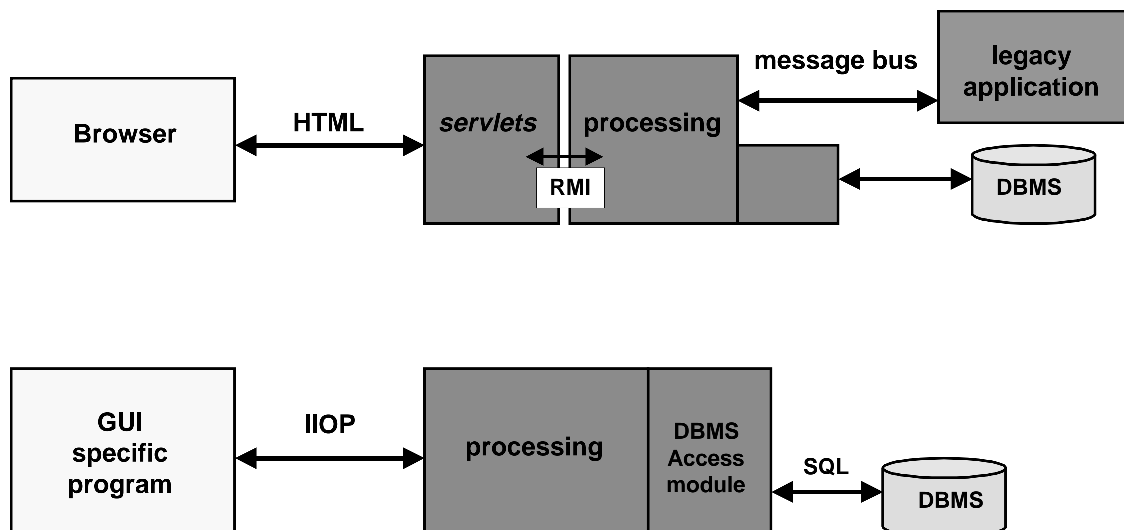
Evolution of client-server architectures (3)

- **3-Tier Architecture**



- Potential benefits
  - Separation of functions
  - Well-defined interfaces, standardization, openness
  - Evolution and scalability
  - Reuse of legacy code

Some Architectural Variations



Developing a distributed application

- Objectives
  - Ease of development (low cost)
  - Ease of evolution, scalability
  - Openness (integration, heterogeneous systems, etc)
- Possible solutions
  - Object-oriented middleware may be used …
    - Java RMI, CORBA, DCOM, Message Bus
  - … but must be extended
    - modular construction for evolution and openness
    - common services (avoid “reinventing the wheel” and concentrate on the specific application)
    - development tools (program development, composition)
    - deployment tools (generating and placing the elements)
    - administration tools (observation, reconfiguration)
  - Component-based middleware aims at providing these extensions

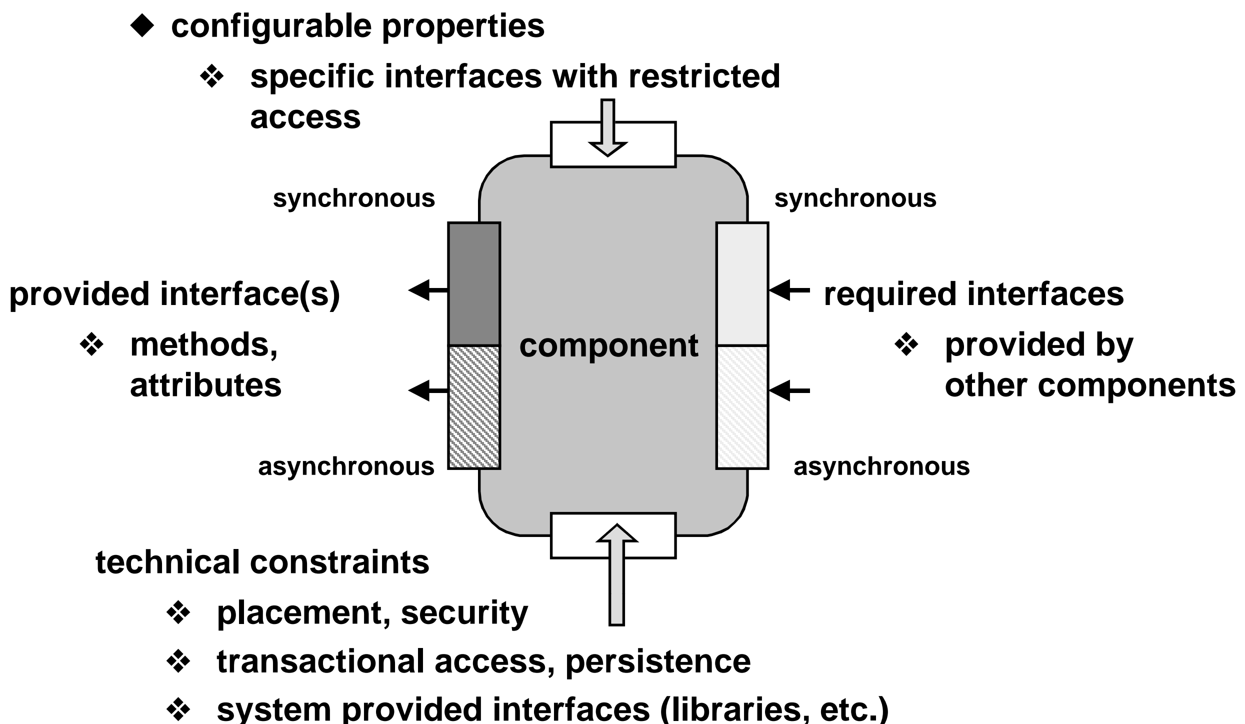
Limits of object-oriented programming

- No global vision of the application
  - Main concepts are defined for a single object
  - No global description of the architecture
- Evolution is difficult
  - Consequence of the lack of a global view
- Services are missing
  - Needed services must be implemented “by hand” (persistence, security, fault tolerance, etc.)
- Tools are missing (composition, deployment)
- Conclusion
  - Important load on the programmer
  - Incidence on the quality of the application
  - Part of the lifecycle is not covered

Components: a definition

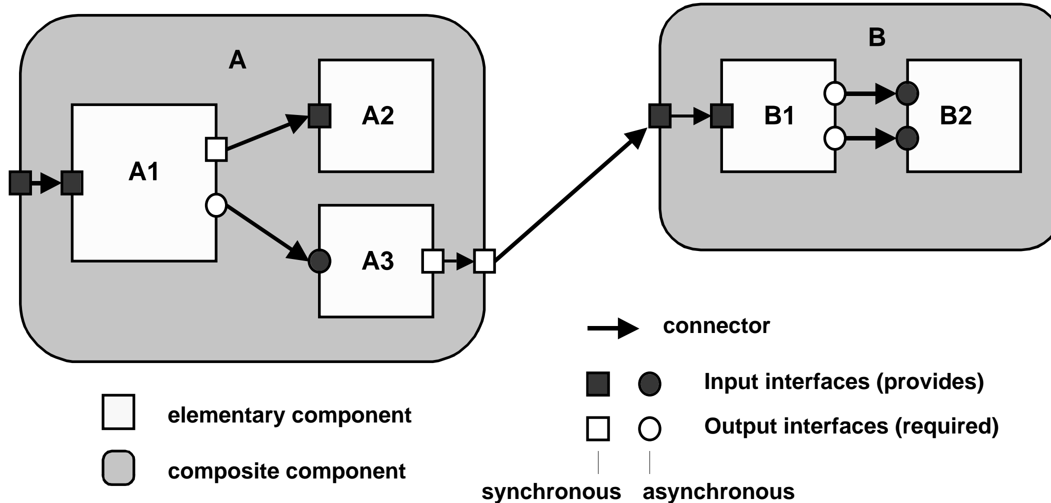
- **Definition**
  - Autonomous software module
    - unit of deployment (installation on various platforms)
    - unit of composition (combination with other components)
- **Properties**
  - explicitly specifies the provided interface(s) (attributes, methods)
  - explicitly specifies the required interface(s)
  - may be configured
  - self-describing
- **Benefits**
  - allow the construction of applications by composition of configurable elementary pieces
  - separate the functions of component provider and assembler (conditions for the development of a component industry

Components : a Generic Model



Using Components

Hierarchical composition and encapsulation
(composite components, sub-components)
Interconnection of components
(connectors, or binding objects)



© 2002, S. Krakowiak

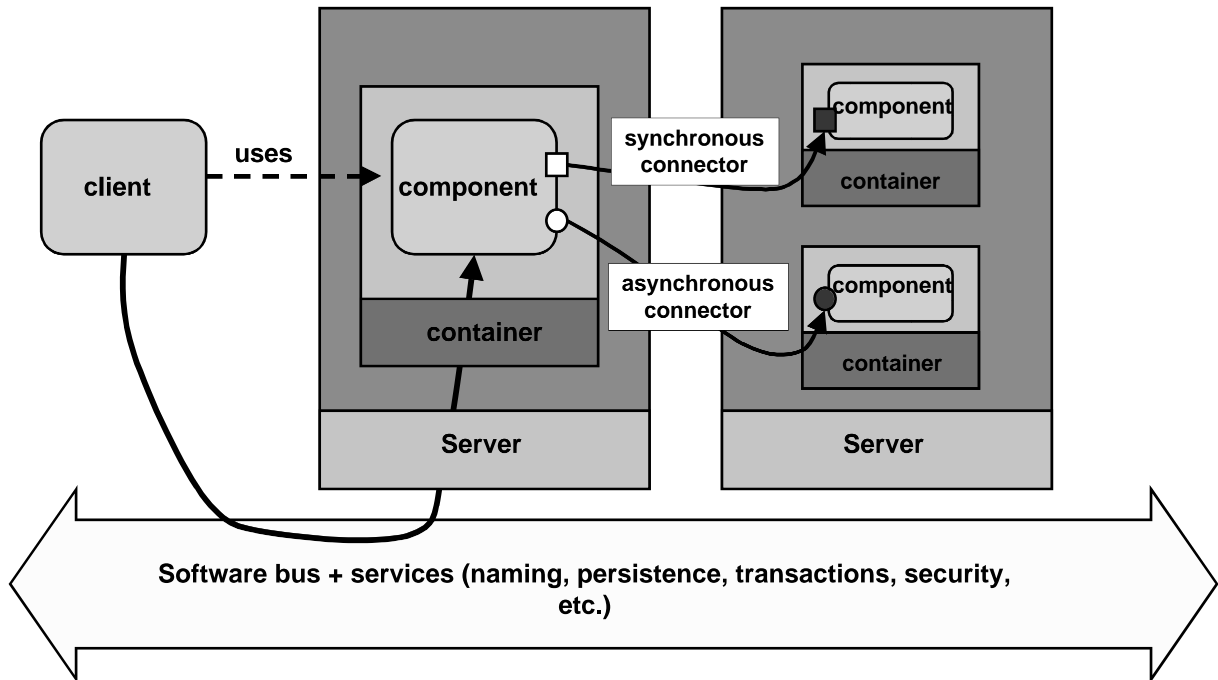
Support Software for Components

In order to fulfill their task, components need software support. This is provided by containers and servers

- **Container**
  - encapsulates one or several component(s)
  - takes care of system services
    - naming, security, transactions, persistence, etc.
  - (partly) takes care of relationships between components
    - invocations, events
  - techniques involved: interposition, delegation
- **Server**
  - execution space for containers and components
  - mediator between containers and system services

© 2002, S. Krakowiak

Components in Action

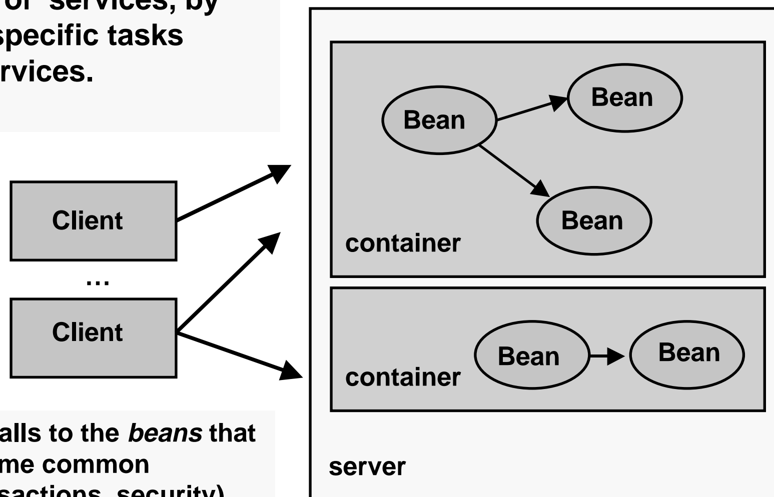


Enterprise Java Beans (EJB)

- **Objectives**
  - **Facilitate the construction of programs for enterprise servers** (the middle tier of a 3-tier architecture) **by assembling reusable components**
  - **Provide an environment for common services (persistence, transactions, etc.), allowing the developer to concentrate on the specific problems of the application**
  - **Favor the development of a component industry by separating the functions of component production, applications assembly, and de service provision**
- **Overview**
  - **2 types of *Enterprise Beans***
    - ***Entity Bean* : represents an object, usually persistent**
    - ***Session Bean* : represents a sequence of actions for a client**
  - **Environment = server + container**

EJB Execution Scheme

The *beans* provide a set of services, by performing application specific tasks
The clients use these services.

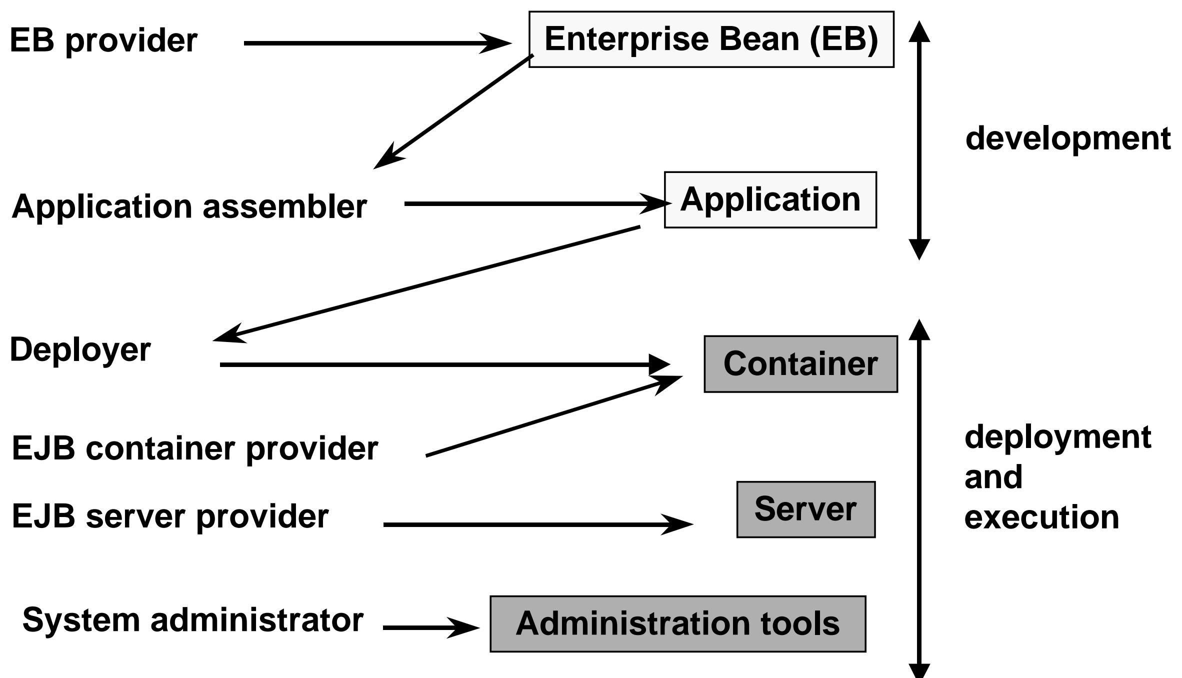


A container intercepts the calls to the *beans* that it contains and performs some common functions (persistence, transactions, security).

The containers isolate the *beans* - a) from the clients, - b) from a specific server implementation

The server is a mediator between the system and the container (it allows the container to perform its functions by calling the system primitives)

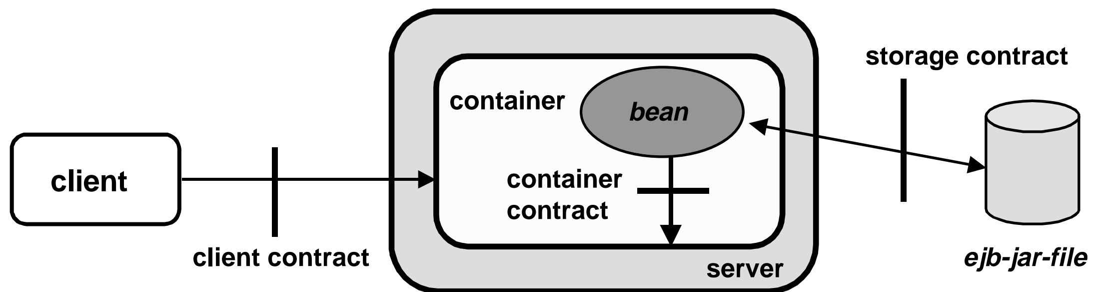
Enterprise Java Beans Related Roles



source : documentation JOnAS : http://www.objectweb.org/jonas/

EJB Contracts

- "Contracts" are associated with each *bean*
  - should be respected by the client, the *bean* and the container
  - client-side contract
    - provides a uniform view of the bean to the client (this view is independent of the deployment platform)
  - container contract
    - allows *beans* to be ported on different servers
  - storage contract (*packaging*)
    - defines a standard file format (*ejb-jar-file*) for archiving *beans*. This format should be respected by all tools



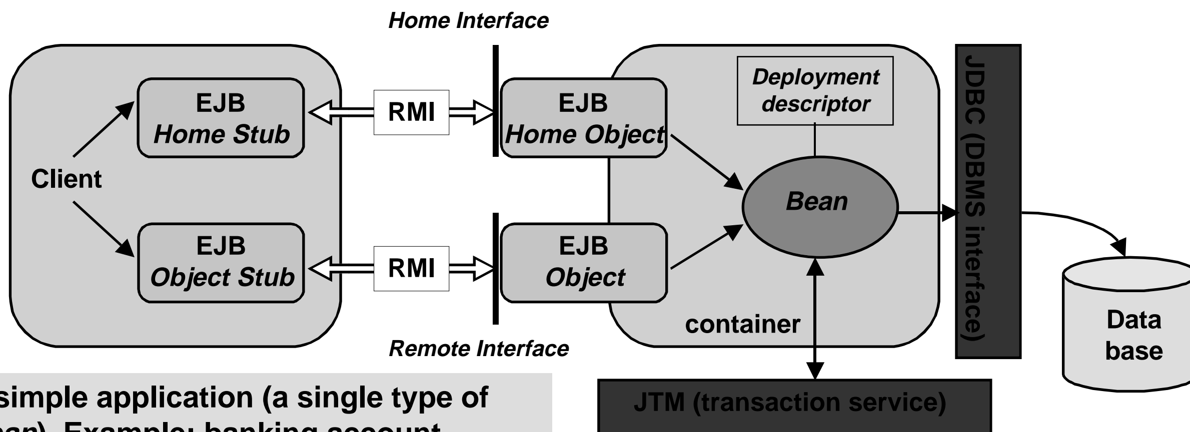
© 2002, S. Krakowiak

Client-side Contract

- **Locating the *bean***
  - use JNDI Java (interface to directory service)
- **Using the *bean***
  - via the standard interfaces provided by the *beans* developer
    - *Home Interface* (methods for *bean* management : *create*, *remove*, *find*, etc.)
    - *Remote Interface* (application specific methods)
- **The container implements a delegation mechanism**
  - the client does not directly call the *bean*, but the container
  - the container "forwards" the calls

© 2002, S. Krakowiak

Executing EJBs: a closer view



A simple application (a single type of *Bean*). Example: banking account

An *Object EJB* and a *Home Object EJB* are associated with each bean
The *Home Object EJB* is in charge of the *Bean*'s lifecycle : creation, identification lookup, destruction. It is located through a directory (JNDI interface)
The *Object EJB* implements the interface to the *Bean*'s services. It intercepts the calls to the *Bean* and performs tasks related to transactions, persistence, state management, security, as specified in the *deployment descriptor*.

Container Contract

- A container implements functions for the enclosed *beans*
  - lifecycle management, state management, security, transactions, concurrency, etc.
  - These services call methods provided by the *bean* (*callback methods*)
    - example : *ejbCreate*, *ejbPostCreate*, *ejbLoad*, *ejbStore*, etc.
- The containers manage two types of *Beans*
  - *Entity Beans* : implement the application's objects
  - *Session Beans* : implement operations sequences for a client
  - specific contracts are defined for for each of these types (with some variations according to the degree of involvement of the container)

EJB *Entity Beans*

Properties

- represent persistent entities (stocked in a database) as objects ; the *entity beans* are themselves persistent
  - persistence may be managed by the *bean* itself (*bean managed persistence*) or delegated to the container (*container managed persistence*)
- shared by several clients
  - concurrency management
- may be involved in transactions
- survive to shutdowns or failures of the EJB servers
- how are they created?
  - explicit instance creation
  - Inserting an entity in the database

EJB *Session Beans*

Properties

- Represent a sequence of operations for a specific client
  - Processing in memory
  - Access to DBMS
- Created and destroyed by a client
- Non persistent
- Do not survive failures or server shutdown
- Two modes of state management
  - *stateless*: no internal data; may be shared between several clients; no passivation (see later)
  - *stateful* : preserves its state throughout a sequence of method calls (for a single client); must include passivation and activation primitives (see later)

Lifecycle Managed by the Container

- Administration (via *Home Interface*)
  - Allows a client to create, destroy, lookup a *bean*
  - The *bean* must implement the corresponding methods (*callback*) : *ejbCreate*, *ejbPostCreate* , etc
- Management of a *bean*'s state by the container
  - Passivation
    - Stores the *bean's state* in persistent storage
    - Deactivates the *bean* (it may no longer be called)
  - Activation
    - Stores the *bean's state* in persistent storage
    - Reactivates the *bean* (it may then be called again)
  - The *bean* must implement the (*callback*) methods *ejbPassivate and ejbActivate*

Persistence Management

- Container-managed Persistence
  - The container is responsible for saving the beans' state
  - The fields to be saved and the storage support must be specified separately (in a deployment descriptor)
- *Bean*-Managed Persistence
  - The *bean* is responsible for saving its state
  - (must explicitly insert the operations that perform persistence management in appropriate *callback* functions)
  - Less adaptable than container managed persistence, since persistence management operations are "hardwired" in the code

Transaction Management

- Conforms to the XA architecture (X/Open Architecture)
  - in Java : Java Transaction API (JTA)
- Flat transaction model
- Declarative specification
  - for a whole *bean* or for each method
  - transactional attributes defined in the deployment descriptor
  - also possible explicit control by the *bean*

Transactional attribute associated with a *bean* method	Client Transaction	Transaction executing the method of the *bean*
NotSupported	--	--
	T1	--
Required	--	T2
	T1	T1
Supports	--	--
	T1	T1
RequiresNew	--	T1
	T1	T2
Mandatory	--	Error
	T1	T1
Never		

Security Management

- **Security management is delegated to the container**
  - simplifies the programmer's task, enhances portability
- **Basic security principles**
  - Java security API (*javax.security*)
  - security-related methods in the container (*javax.ejb.EJBContext*: container interface, base of *EntityContext* and *SessionContext*)
    - *getCallerPrincipal*
    - *isCallerInRole*
  - security attributes in the *Bean*'s deployment descriptor
    - specification of "roles"
    - specification of methods executable under a role

Deployment Descriptor

- **Function**
  - declarative specification of some properties of a *bean*
    - Identification, transactional attributes, persistent fields, environment, container- or bean- managed, roles for security
  - used by the container to perform its task
- **Form**
  - Since EJB 1.1 : descriptor written in XML
  - For each property, a specific tag is defined by the DTD
  - Example

```
…
<persistence-type>Container</persistence-type>          container managed
                                                        persistence
<container-transaction>
        <method>
                <ejb-name>MyBean</ejb-name>             container managed
                <method-name>*</method-name>            transactions
        </method>
        <trans-attribute>Required</ trans-attribute>    transactional execution
</container-transaction>                                Mandatory for all methods
…
```


Developing an application with EJB (1)

- **Developing a *Session Bean***
  - create the *Home Interface* (extend *ejb.EJBHome*)
    - methods *create*, *remove*
  - create the *Remote Interface* (extend *ejb.EJBObject*)
    - application specific methods
  - the implementation of the creation methods (*Create*, *PostCreate*)
  - write the implementation of the interface
    - specific methods
    - *callback* methods
      - *setSessionContext*, *ejbActivate*, *ejBPassivate*, etc
      - if the *bean* has a container-managed state: *afterBegin*, *afterCompletion*, *beforeCompletion*, etc

Developing an application with EJB (2)

- **Developing an *Entity Bean***
  - **create the *Home Interface* (extend *ejb.EJBHome*)**
    - **methods *create*, *remove***
  - **create the *Remote Interface* (extend *ejb.EJBObject*)**
    - **application specific methods**
  - **write a *Serializable* class "primary key" (*BeanNamePK*) to be used for *bean lookup* and access**
  - **write the implementation of creation methods (*Create*, *PostCreate*)**
  - **write the implementation of the interface**
    - **specific methods**
    - ***callback* methods**
      - ***setEntityContext*, *ejbActivate*, *ejbPassivate*, etc**
      - ***ejbLoad*, *ejbStore*, etc. For container managed persistence, *ejbLoad* and *ejbStore* may be empty.**

Steps for developing an EJB application (3)

- **Write a deployment descriptor**
  - **one per *bean***
  - **defines the behavior for transactions, persistence, security, link with the databases, environment (placement of the servers), etc.**
- **Set up the server**
  - **compile the programs of the *beans***
  - **generate the container classes (implementation of the *Home and Remote* interfaces) with the adequate tool (*GenIC* in JOnAS) using the *bean* classes and the deployment descriptor**
- **Develop and start the client program**
  - **the client gets the reference to the *beans* via a name service**
  - **the client may create and start sessions, or directly call the *Entity beans***

Examples

See example in the JOnAS documentation : http://objectweb.org/jonas/

Simple example : bank server

an *Entity Bean* in two versions

AccountImpl : implicit persistence (container managed)

AccountExpl : explicit persistence (*bean* managed)

a client program (without session) *ClientAccount*

a deployment descriptor *ejb-jar*

a *Makefile* file

More complex example : electronic commerce server

Conclusion on EJB

- A component-based model for server side programming
  - widely used; influences the normalization process (OMG)
- Benefits
  - simplifies the development of complex applications by freeing the developer from the aspects not directly relevant to the application
    - declarative transaction management
    - persistence management
    - security management
    - distribution management
  - increases the independence between platform and applications
    - separation of the providers' roles
    - openness, competition, improved quality
  - extensible model

A Brief Conclusion

- Patterns and Frameworks are important
  - To improve our understanding of software architecture
  - To help reusing proven designs
  - To help reusing code bases
- Open Source Sofware is important
  - To improve the quality of code
    - Wide community
    - Critical reviews
    - Reactivity and evolution
  - To improve our understanding of software architecture
- Good documentation is important
  - Using patterns and frameworks to document open source
  - Strong implications to education

This is a new and exciting area for research!

© 2002, S. Krakowiak