Remote Procedure Calls

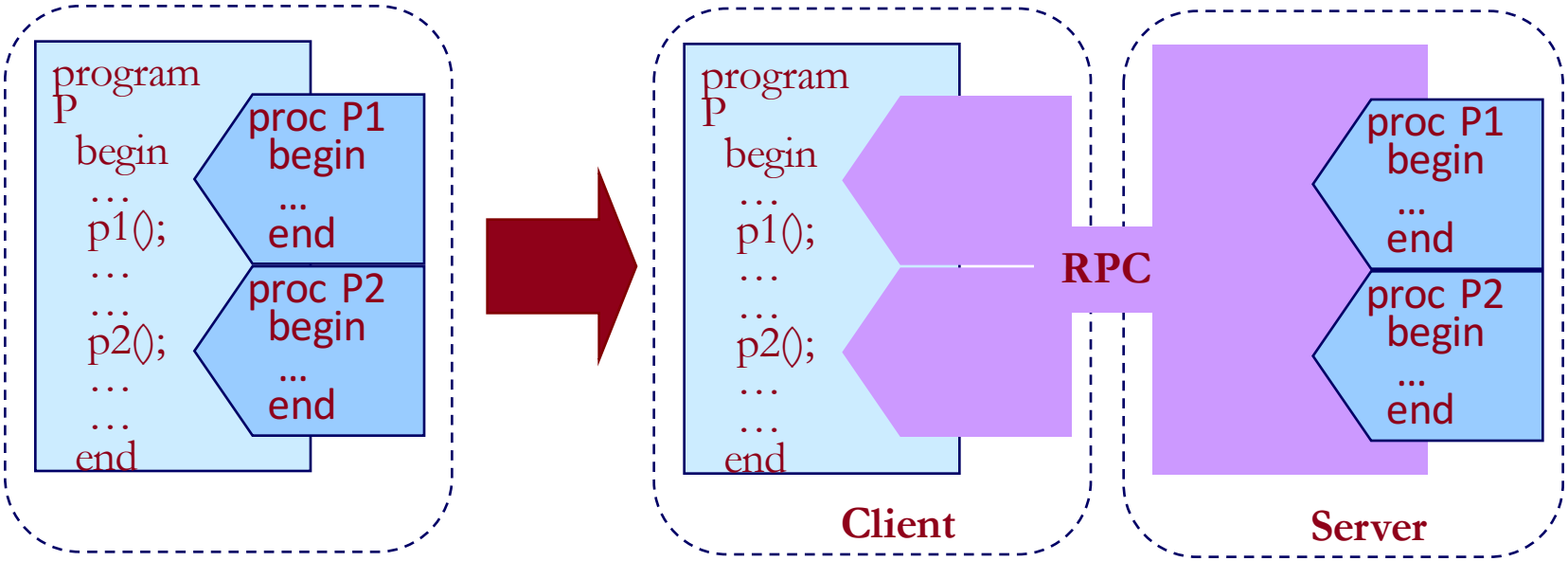
Contents

- Remote procedure call
  - Fundamentals
  - RPC in C (under Unix/Linux)
- Remote Method Invocation
  - Fundamentals
  - Java RMI

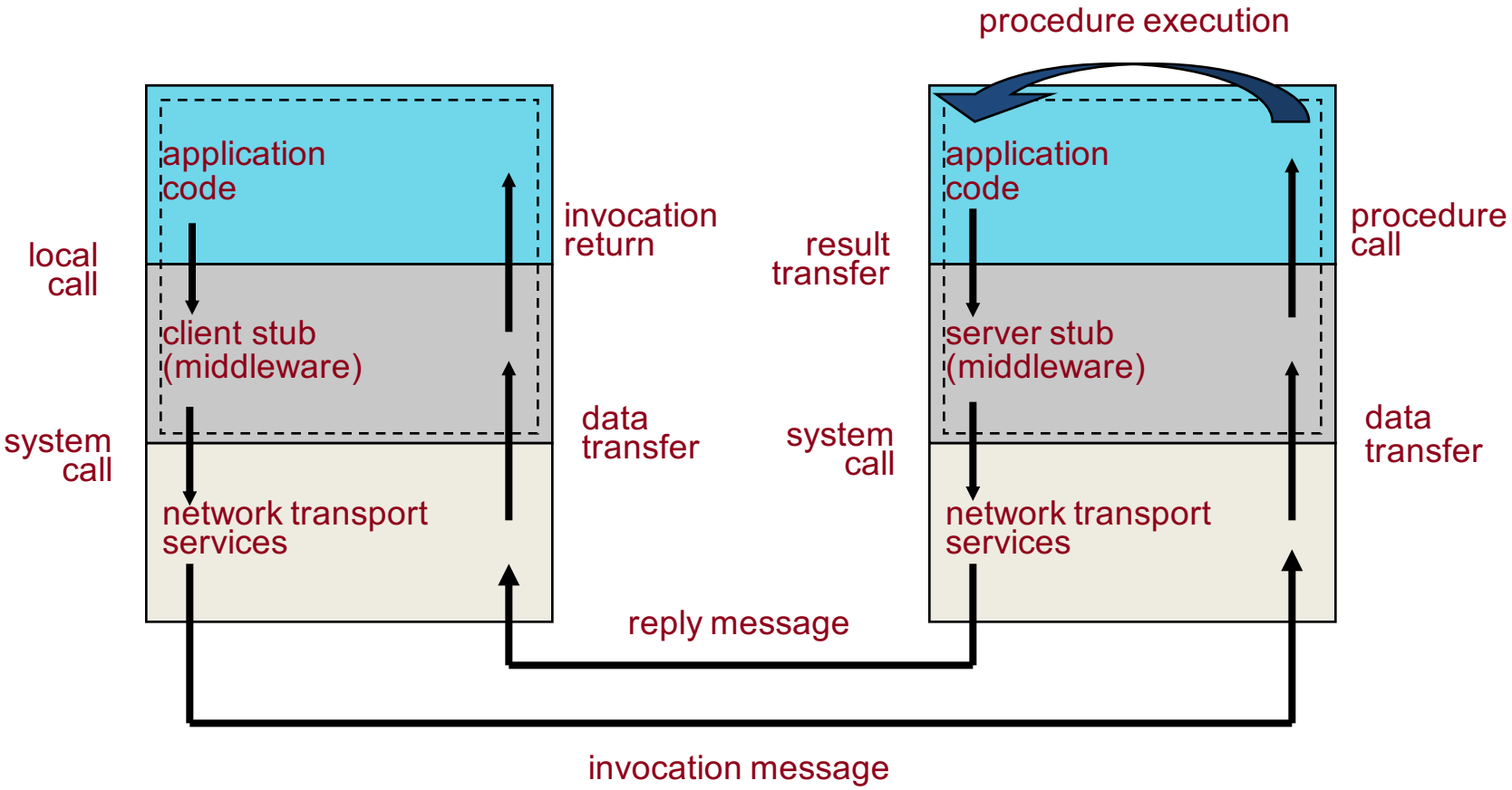
REMOTE PROCEDURE CALLS (RPC)

RPC: Fundamentals

- Problem: client-server interaction is handled through the OS primitives for I/O ➔ difficult to develop applications
- Idea (Sun Microsystems in the early 80s): enable remote access through the well-known procedure call programming model



RPC: How does it work



Parameter passing: Marshalling and serialization

- Passing a parameter poses two problems:
  - Structured data (e.g., structs/records, objects) must be ultimately flattened in a byte stream
    - Called *serialization* (or pickling, in the context of OODBMSs)
  - Hosts may use different data representations (e.g., little endian vs. big endian, EBCDIC vs. ASCII) and proper conversions are needed
    - Called *marshalling*
- Middleware provides automated support:
  - The marshalling and serialization code is automatically generated from and becomes part of the stubs
  - Enabled by:
    - A language/platform independent representation of the procedure's signature, written using an *Interface Definition Language* (IDL)
    - A data representation format to be used during communication

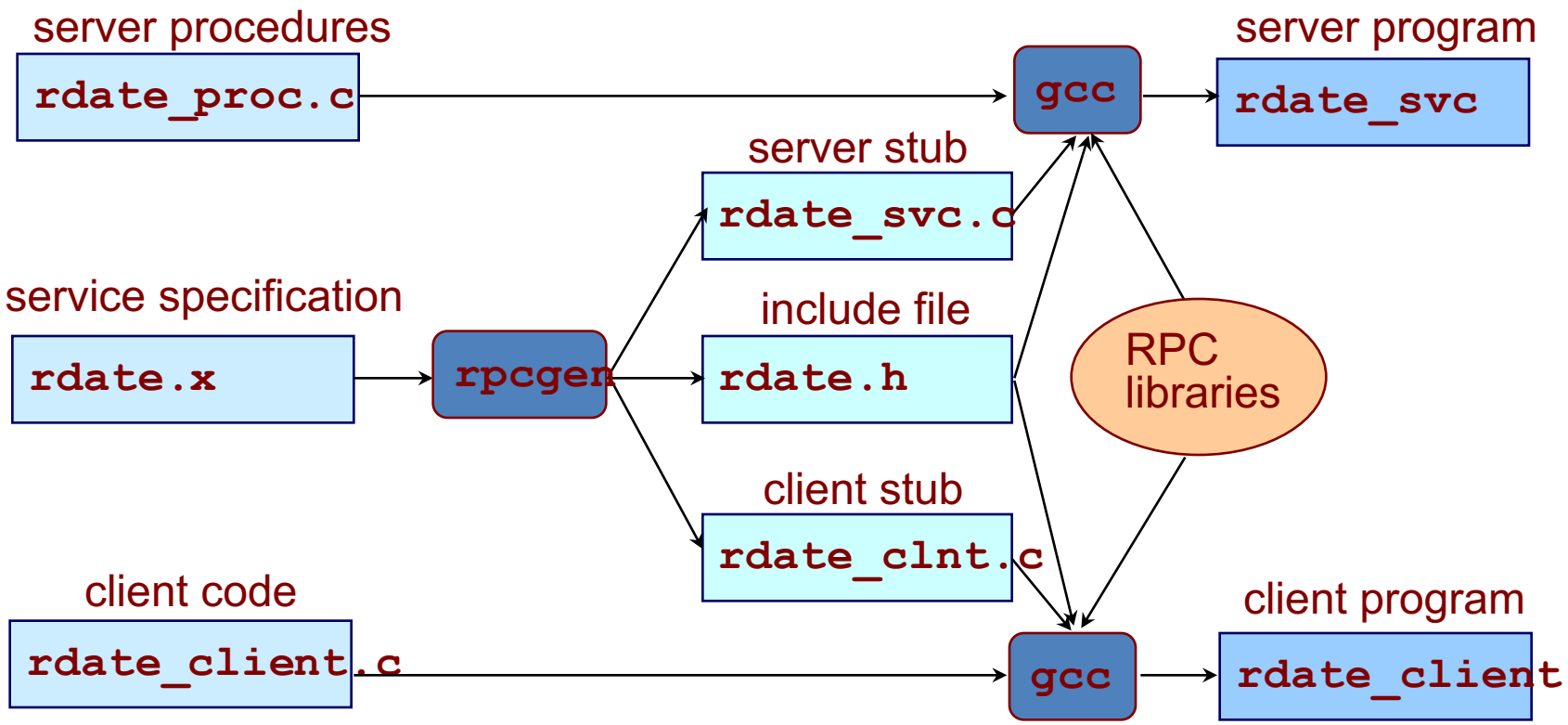
The role of IDL

- The *Interface Definition Language* (IDL) raises the level of abstraction of the service definition
  - It separates the service *interface* from its *implementation*
  - The language comes with "mappings" onto target languages (e.g., C, Pascal, Python…)
- Advantages:
  - Enables the definition of services in a language-independent fashion
  - Being defined formally, an IDL description can be used to automatically generate the service interface code in the target language

Sun RPC (ONC RPC)

- Sun Microsystems' RPC (also called Open Network Computing RPC, ONC RPC) is the *de facto* standard over the Internet
  - At the core of NFS, and many other services
  - Found in modern Unix systems (e.g., Linux)
- Data format specified by XDR (eXternal Data Representation)
  - Initially only for data representation, then extended in a proper IDL
- Transport can use either TCP or UDP
- Parameter passing:
  - Only pass by copy is allowed (no pointers)
  - Only one input and one output parameter
- Provision for DES security

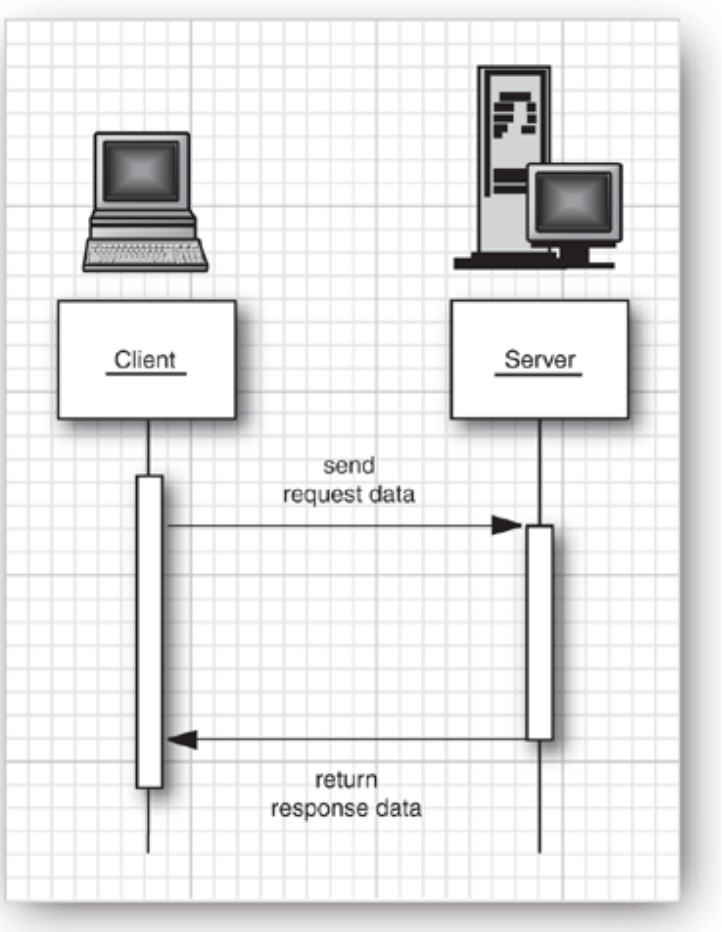
Sun RPC: Development cycle



REMOTE METHOD INVOCATION (RMI)

Towards RMI…

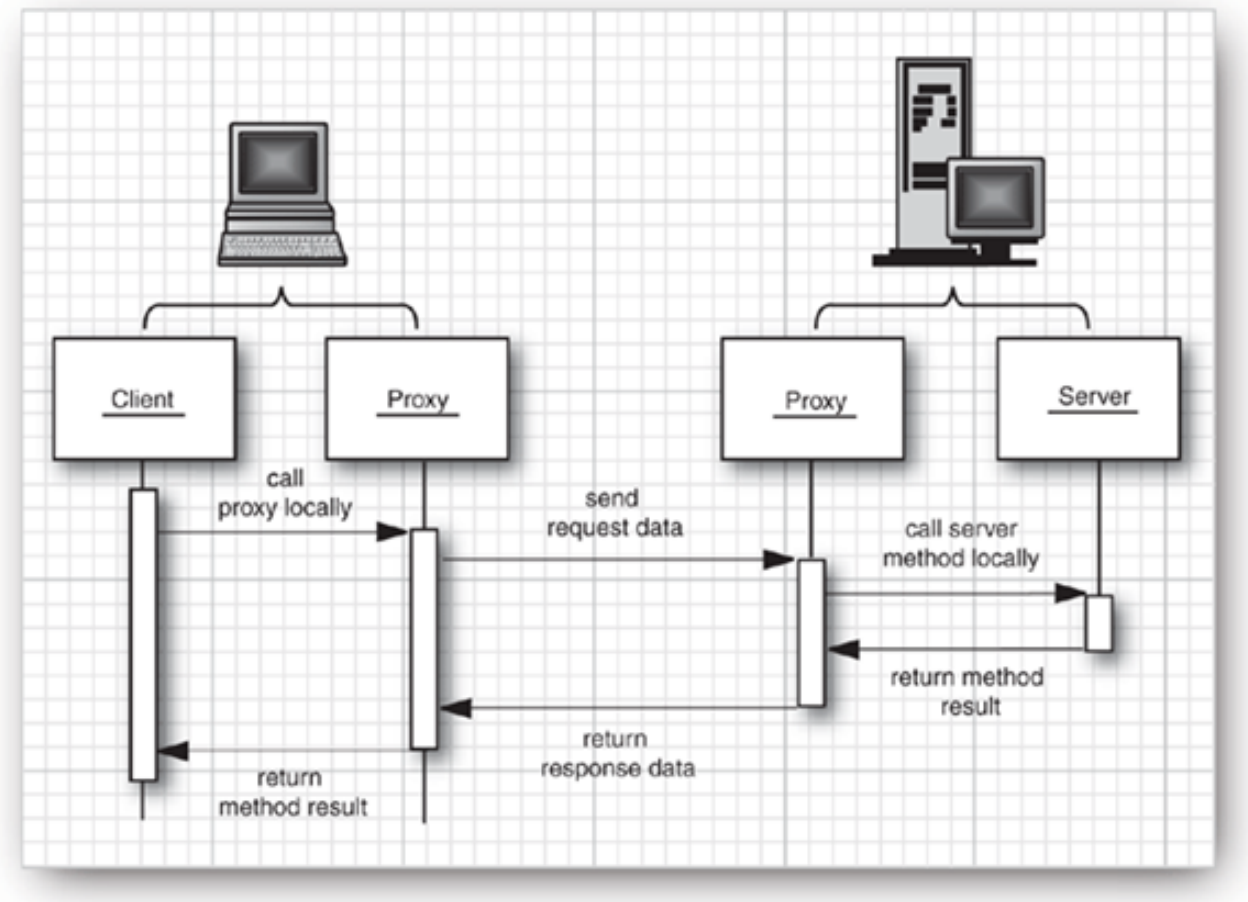
- The basic idea is simple
  - A client executes a request. This request travels over the network towards a specific Server.
  - The server processes the requests and sends back a result.
  - If we were to work with sockets we would have to explicitly deal with message formats and connections.



Towards RMI…

- We want a mechanism through which the developer can simply execute a method call.
  - Without having to deal with network issues!
- The technical solution is to install a proxy on the client side and have the proxy hide away all the complexity.
  - The proxy appears to the client as a regular object.
- At the same time the server-side developer also does not want to have to deal with low-level network technicalities.
  - He/She too can install a proxy…
  - This creates a new level of abstraction

Towards RMI…


Client
Proxy
Proxy
Server
call
proxy locally
send
request data
call server
method locally
return method
result
return
response data
return
method result

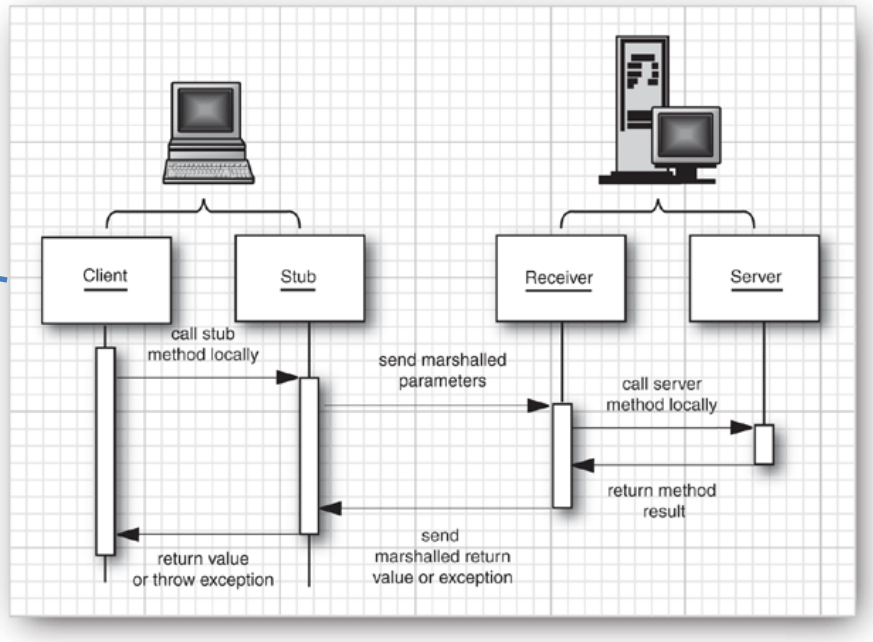
The Basics of RMI

- *Remote Object*
  - Objects whose methods can be called from a JVM that is different from the one they exist in.
- *Remote Interface*
  - The interface defines the methods that can be invoked from a different JVM.
- *Server*
  - The Server is a set of “one or more” remote objects that offer resources (data and computation) , through remote interfaces, to external machines that are distributed on the network.
- *Remote Method Invocation (RMI)*
  - Invocation of a method offered by a remote object that implements a remote interface. The syntax is identical to a local method invocation.

RMI Stubs

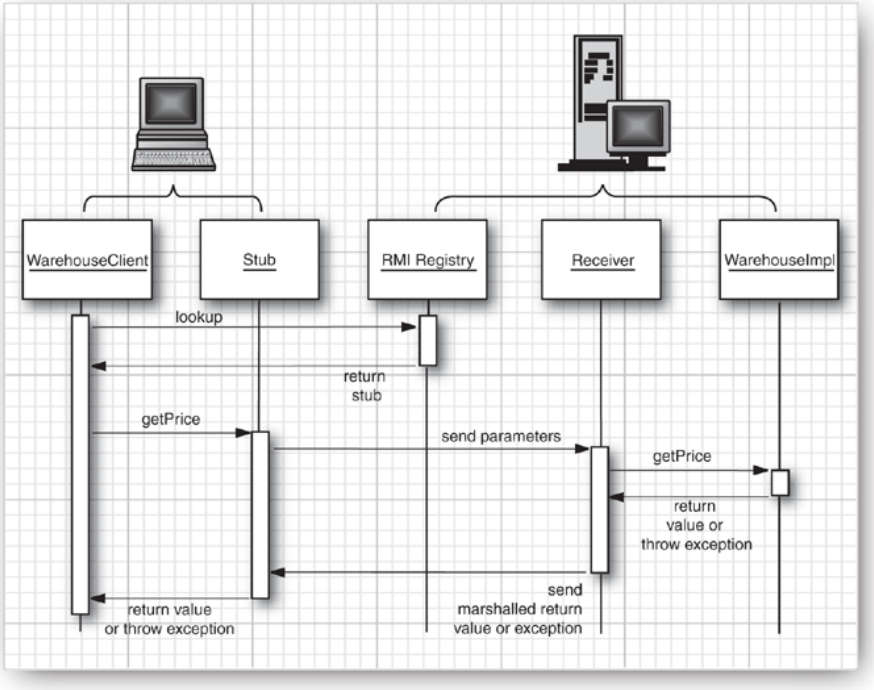
- RMI workflow
  - The server instantiates remote objects and makes them visible
  - The clients obtain references to the remote objects called stubs.
  - Through these stubs they make their calls.

But how do I know the stubs???



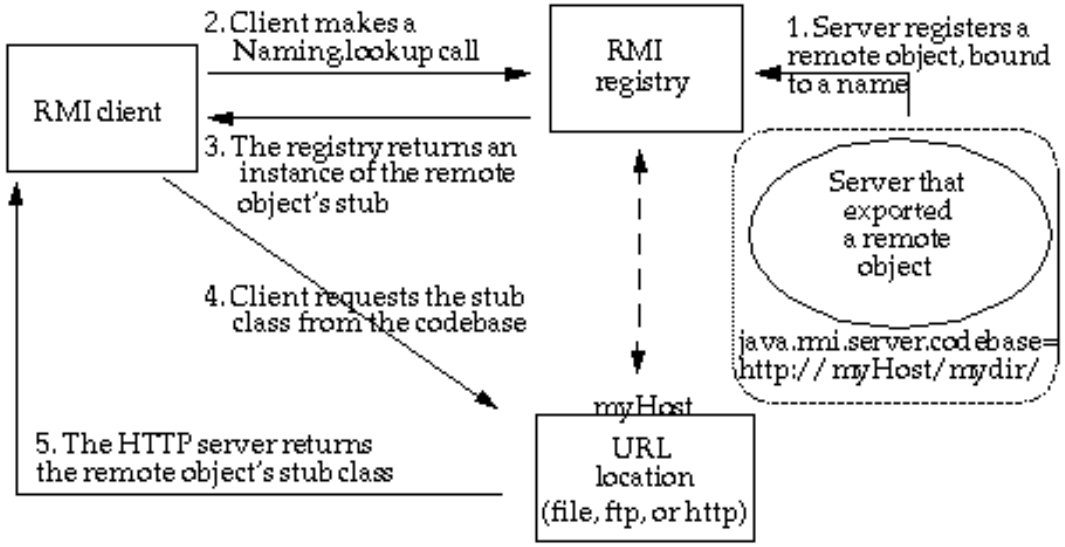
RMI Registry

- The RMI Registry provides the stubs to the clients
  - During registration the server must provide a canonical name for the remote object. This is a unique identifier.
  - The client must know this canonical name before hand.



Downloading the stub

- The RMI Registry can send the client the stub in different ways…
  - If the two JVMs reside on the same machine they can use the local filesystem
  - If they reside on different machines they must use an external and well-identified HTTP server.



Summary

- **Client side**
  1. Request the stub needed to make the remote method call
  2. Serialize the parameters you need to pass to the remote object
  3. Send the request to the server

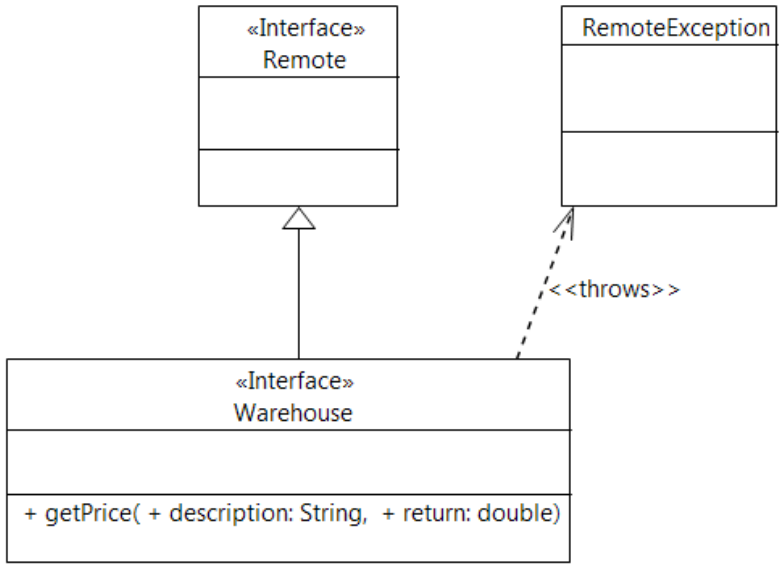
- **Server side**
  1. The server receives the serialized data and localizes the remote object that needs to be invoked
  2. The server calls the desired method passing it the de-serialized input parameters
  3. The server captures the return parameter and any eventual exceptions
  4. The server sends everything back to the client

Running Example

- We will implement the well-known Warehouse example
- The warehouse contains a set of products and each product has:
  - A string identifier which is to be unique
  - A price

Shared interface

- The Warehouse <<interface>> clarifies what methods are being exposed
- It extends the Remote <<interface>>
- All methods need to throw a *RemoteException*



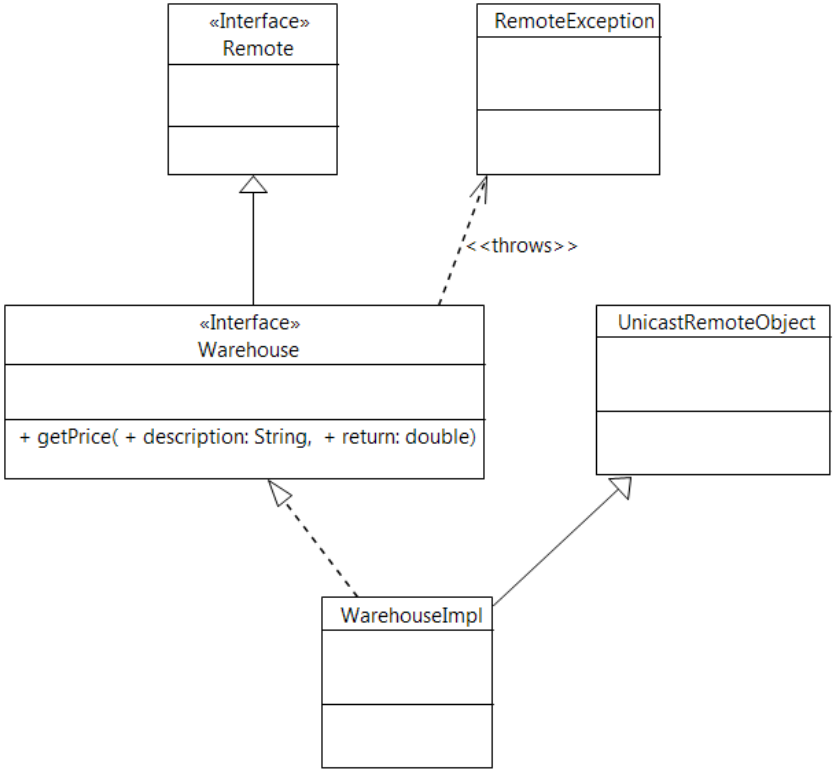
Let’s implement the shared interface

Shared Interface

```
import java.rmi.*;

public interface Warehouse extends Remote{

    double getPrice(String description) throws RemoteException;

}
```

Warehouse Server

- The server implements the remote interface
- It must extend the class *UnicastRemoteObject* that makes the object accessible from a remote location



Let's implement the server side component

Warehouse Server-side implementation

Makes the object remotely accessible

```
import java.rmi.*;
import java.rmi.server.*;
import java.util.*;

public class WarehouseImpl extends UnicastRemoteObject implements Warehouse{
    private Map<String, Double> prices;

    public WarehouseImpl() throws RemoteException
    {
        prices = new HashMap<String, Double>();
        prices.put("Blackwell Toaster", 24.95);
        prices.put("ZapXpress Microwave Oven", 49.95);
    }

    public double getPrice(String description) throws RemoteException
    {
        Double price = prices.get(description);
        return price == null ? 0 : price;
    }

}
```

Alternative Server-side implementation

```
import java.rmi.*;
import java.rmi.server.*;
import java.util.*;

public class WarehouseImpl implements Warehouse{
    public WarehouseImpl() throws RemoteException
    {
        prices = new HashMap<String, Double>();
        prices.put("Blackwell Toaster", 24.95);
        prices.put("ZapXpress Microwave Oven", 49.95);
        UnicastRemoteObject.exportObject(this, 0);
    }

    public double getPrice(String description) throws RemoteException
    {
        Double price = prices.get(description);
        return price == null ? 0 : price;
    }

    private Map<String, Double> prices;
}
```

Publishing the remote object (1)

- The server starts by publishing the remote object to the RMI Registry
- The registry must be online before this occurs
  - We will see how to do this.

Assumes the registry is in the default location -> localhost:1099

```
WarehouseImpl centralWarehouse = new WarehouseImpl();
Registry registry = LocateRegistry.getRegistry();
registry.bind("central_warehouse", centralWarehouse);
```

binding the name “central_warehouse” to the remote object centralWarehouse

Let’s publish it…

Publishing the remote object (2)

```
import java.rmi.*;
import javax.naming.*;

public class WarehouseServer
{
    public static void main(String[] args) throws RemoteException, NamingException
    {
        System.out.println("Constructing server implementation...");
        WarehouseImpl centralWarehouse = new WarehouseImpl();

        System.out.println("Binding server implementation to registry...");
        Registry registry= LocateRegistry.getRegistry();
        registry.bind("central_warehouse", centralWarehouse);

        System.out.println("Waiting for invocations from clients...");
    }
}
```

Technical notes on bind()

- For security reasons bindings, unbindings or rebindings can only be performed by an application that is running on the same registry host.
- This allows us to avoid malicious behaviour.
- Lookups are open to external clients…

```
String[] remoteObjects = registry.list();
```

Technical notes on the Registry

- The registry is also a Remote Object
- The bind() method is part of the object's remote interface

```
void bind(String name, Remote obj)
        throws RemoteException, AlreadyBoundException, AccessException;
```

- The parameters will need to be serialized/deserialized
- The Registry will dynamically download the definition of the remote interface we are using to be able to serialize the object we are passing.

Warehouse Client

- This is how we get a reference to a stub on the client side:
  - Registry must be online
  - The object needs to be already on the server

```
String remoteObjectName = "central_warehouse";
Warehouse centralWarehouse = (Warehouse)
registry.lookup(remoteObjectName);
```

Casting to Warehouse and not WarehouseImpl beacuse this is all the client knows

Let’s implement the client code…

Warehouse Client

```
import java.rmi.*;
import java.util.*;
import javax.naming.*;
public class WarehouseClient
{
    public static void main(String[] args) throws NamingException, RemoteException
    {
        Registry registry= LocateRegistry.getRegistry();

        System.out.print("RMI registry bindings: ");
        String[] e = registry.list();

        for (int i=0; i<e.legth; i++)
            System.out.println(e[i]);

        String remoteObjectName = "central_warehouse";
        Warehouse centralWarehouse = (Warehouse) registry.lookup(remoteObjectName);

        String descr = "Blackwell Toaster";
        double price = centralWarehouse.getPrice(descr);
        System.out.println(descr + ": " + price);
    }
}
```

How to deploy the RMI Application

- What do we need?
  1. Launch the HTTP server.
  2. Launch the RMI Registry
  3. Launch the Server
  4. Launch the Client

Setup

- On the Server side:

```
server/
        WarehouseServer.class
        WarehouseImpl.class
download/
        Warehouse.class
```

- On the client side:

```
client/
        WarehouseClient.class
```

Setup

- Let's launch the server on localhost:8080
  - This is the location we will need to declare our java.rmi.server.codebase for the dynamic downloading.

- Linux and OSX

```
$ rmiregistry –J-Djava.rmi.server.codebase=http://localhost:8080/Warehouse
```

- Windows

```
$ start rmiregistry –J-Djava.rmi.server.codebase=http://localhost:8080/Warehouse
```

- The *rmiregistry* is distributed with Java

Setup

- Let's go to the server dir and type

```
$ java WarehouseServer
```

- What do we see?

```
Constructing server implementation...
Binding server implementation to registry...
Exception in thread "main" javax.naming.CommunicationException [Root exception is
java.rmi.ServerException: RemoteException occurred in server thread; nested exception is:
        java.rmi.UnmarshalException: error unmarshalling arguments; nested exception is:
        java.lang.ClassNotFoundException: Warehouse]
        at com.sun.jndi.rmi.registry.RegistryContext.bind(RegistryContext.java:143)
        at com.sun.jndi.toolkit.url.GenericURLContext.bind(GenericURLContext.java:226)
        at javax.naming.InitialContext.bind(InitialContext.java:419)
        at WarehouseServer.main(WarehouseServer.java:13)
```

```
$ java -Djava.rmi.server.codebase=http://localhost:8080/Warehouse/  WarehouseServer
```

Logging the application

- Logging can be very important in a distributed application.
- The easiest way to do it id:
  - -Djava.rmi.server.logCalls=true
  - All RMI calls and exceptions are saved to System.err.
- For more complex needs
  - http://download.oracle.com/javase/1.4.2/docs/guide/rmi/logging.html

PASSING PARAMETERS

Remote and Non-remote objects

- A **non-remote** object (passed or received as a method parameter) is passed by copy
  - It is serialized and put into the stream
  - It is then deserializzed on the other side into a *new* copy
  - Changes made to a non-remote object have no effect on the original copy

- A **remote** object (already exported,or passed or received as a method parameter) is passed using a stub
  - A remote object passed as a parameter can only provide a remote interface

Let’s implement a new method that does this…

Warehouse V2

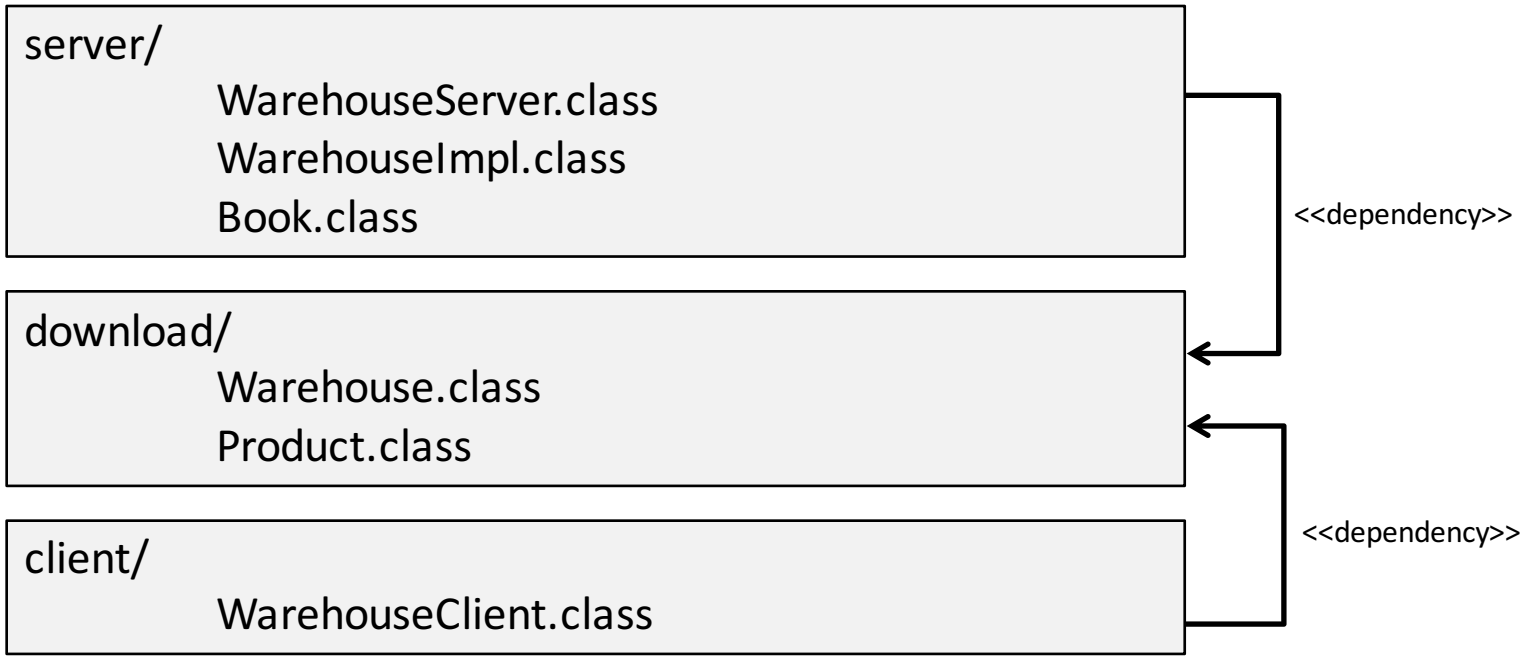
```
import java.rmi.*;
import java.util.*;

public interface Warehouse extends Remote
{
    double getPrice(String description) throws RemoteException;
    Product getProduct(List<String> keywords) throws RemoteException;
}
```

Product

```
import java.io.*;

public class Product implements Serializable{
    private String description;
    private double price;
    private Warehouse location;

    public Product(String description, double price){
        this.description = description;
        this.price = price;
    }
    public String getDescription(){
        return description;
    }
    public double getPrice(){
        return price;
    }
    public Warehouse getLocation(){
        return location;
    }
    public void setLocation(Warehouse location){
        this.location = location;
    }
}
```

New Project Structure

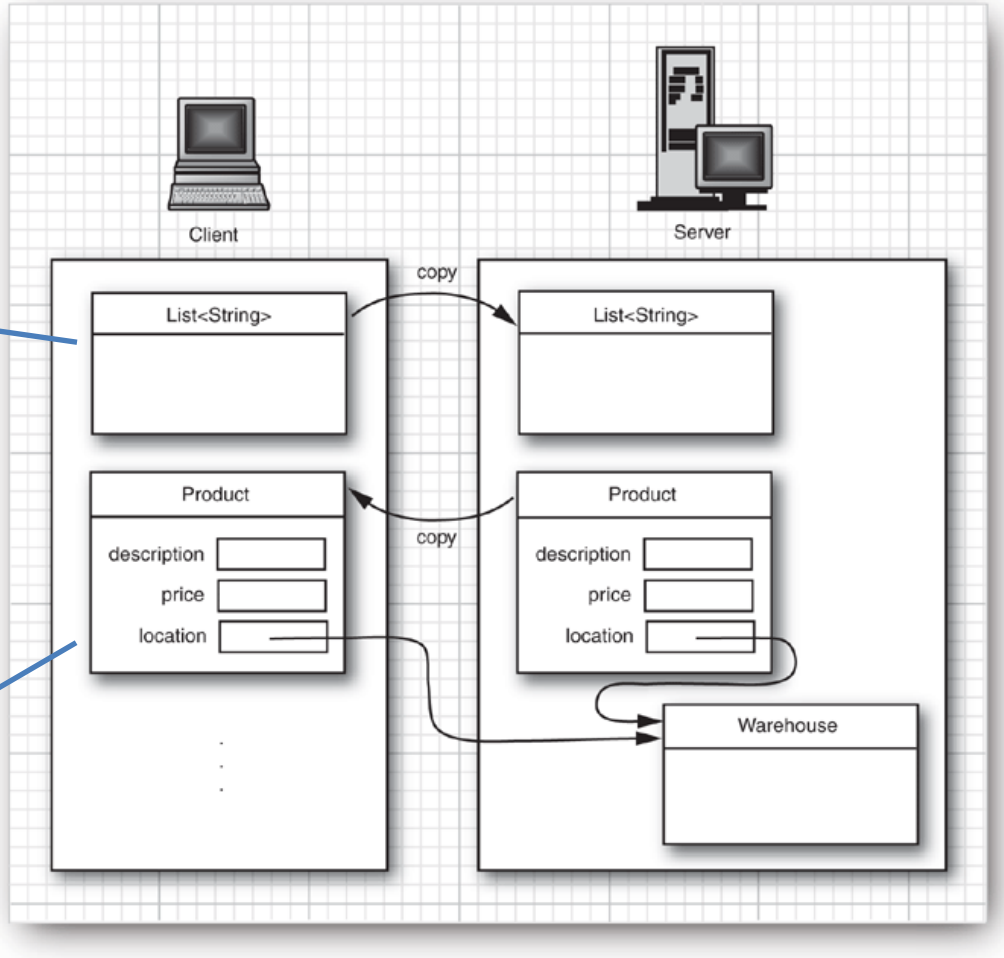
- Three different compilation units
- The server and the client both depend on the common interface artifacts



Application Workflow

Use a list of strings as a set of keywords

The returned product can have a reference to the warehouse



The Book is a subclass of Product, with a different implementation of getDescription

Book

```
public class Book extends Product{
    public Book(String title, String isbn, double price){
        super(title, price);
        this.isbn = isbn;
    }

    public String getDescription(){
        return super.getDescription() + " " + isbn;
    }

    private String isbn;
}
```

Dynamic Class Loading

- Thanks to *Dynamic Class Loading* in Java it is possible to load Java class definitions at run time.
  - The client can receive previously unknown classes from the server
- The server communicates the codebase URL to the client
  - *java.rmi.server.codebase*
- The client contacts the HTTP server and gets the Book.class file so that its code can actually be executed.
  - This is all transparent

Security Policy

- Dynamic Class Loading requires security policies
  - Executing an externally defined piece of code is never nice
- The most simple example…

```
grant {
   permission java.security.AllPermission "", "";
};
```

- This policy needs to be used on the RMIRegistry, on the server so that it can access the rmi registry

Let’s implement the code

WarehouseImpl

```
public class WarehouseImpl extends UnicastRemoteObject implements Warehouse{

    private Map<String, Product> products;
    private Product backup;

    public WarehouseImpl(Product backup) throws RemoteException{
        products = new HashMap<String, Product>();
        this.backup = backup;
    }

    public void add(String keyword, Product product){
        product.setLocation(this);
        products.put(keyword, product);
    }

    public double getPrice(String description) throws RemoteException{
        for (Product p : products.values())
            if (p.getDescription().equals(description)) return p.getPrice();
        if (backup == null) return 0;
        else return backup.getPrice(description);
    }

    public Product getProduct(List<String> keywords) throws RemoteException{
        for (String keyword : keywords){
            Product p = products.get(keyword);
            if (p != null) return p;
        }
        return backup;
    }//getProduct
}//class
```

WarehouseServer

```
import java.rmi.*;
import javax.naming.*;

public class WarehouseServer{
    public static void main(String[] args) throws RemoteException, NamingException{

        System.out.println("Constructing server implementation...");
        WarehouseImpl centralWarehouse = new WarehouseImpl(
            new Book("BackupBook", "123456", 66.99));

        centralWarehouse.add("toaster", new Product("Blackwell Toaster", 23.95));

        System.out.println("Binding server implementation to registry...");
        Registry registry= LocateRegistry.getRegistry();
        registry.bind("central_warehouse", centralWarehouse);

        System.out.println("Waiting for invocations from clients...");
    }
}
```

WarehouseClient

```
import java.rmi.*;
import java.util.*;
import javax.naming.*;
import java.util.ArrayList;
public class WarehouseClient
{
    public static void main(String[] args) throws NamingException, RemoteException
    {
        …

            Warehouse centralWarehouse = (Warehouse) registry.lookup (“central_warehouse”);

            ArrayList<String> l=new ArrayList<String>();
            l.add(”pluto");
            Product p=centralWarehouse.getProduct(l);
            System.out.println("Description: " + p.getDescription());

            String location = null;
            try {
                location = p.getWarehouse().getLocation();
                System.out.println("Product  download  from -> " + location);
            } catch (RemoteException e) {
            // TODO Auto-generated catch block
                        e.printStackTrace();
            }
    }
}
```

ACTIVATABLE OBJECTS

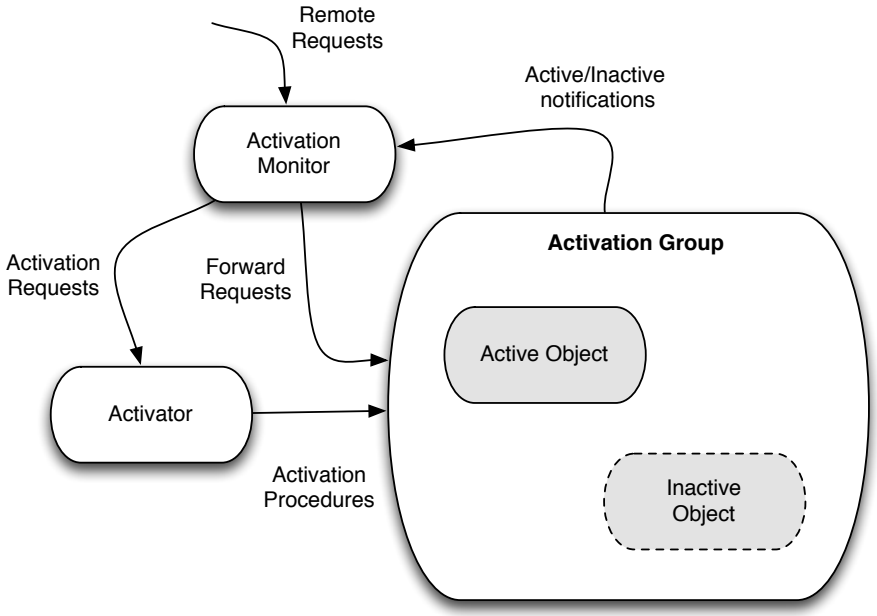
Activatable Objects

- Some times we don't want to have the server remote objects just sitting there waiting to be called
- It would be more efficient if we could set them up so that they
  - Automatically get instantiated every time one is needed.
  - Automatically get torn down when nobody else needs them

- RMI provides a solution for this.
  - The RMI daemon

How does it work?

- With the introduction of the class Activatable and the RMI daemon, **rmid**, programs can be written to register information about remote object implementations that should be created and execute "on demand“, rather than running all the time
- The RMI daemon, **rmid**, provides a Java virtual machine from which other JVM instances may be spawned



Code changes?

- No changes in the common part…
- No changes in the client part…

- There are four steps on the server-side…
  - Make the appropriate imports in the implementation class
  - Extend your class from *java.rmi.activation.Activatable*
  - Declare a two-argument constructor in the implementation class
  - Implement the remote interface methods (you should already have this)

Let’s implement the server-side remote object…

Activatable Server-side

```
public class ActiveWarehouseImpl extends Activatable implements ActiveWarehouse {

private Map<String, ActiveProduct> products;
private String loc;

public ActiveWarehouseImpl(ActivationID id, MarshalledObject<?> data) throws
RemoteException {
                        super(id, 0);
                        //add extra code for initialization from incoming data
                        products = new HashMap<String, ActiveProduct>();
                        this.loc = "Server number 1";
            }
```

The Setup Class

- The job of the "setup" class is to create all the information necessary for the activatable class
  - without necessarily creating an instance of the remote object

- The setup class
  - Makes the appropriate imports
  - Installs a security manager
  - Creates an ActivationGroup instance
  - Creates an ActivationDesc instance
  - Declares an instance of your remote interface and registers it with rmid
  - Binds the stub to a name in the rmiregistry

```
rmid -J-Djava.security.policy=myrmi.policy  -J-Djava.rmi.server.codebase=http://localhost:8888/.../
```

Let’s implement the Setup Class

The Setup Class

```
Properties props = new Properties();
props.put("java.security.policy", "myrmi.policy");

ActivationGroupDesc groupDesc = new ActivationGroupDesc(props, null);

try {
  ActivationGroupID gid = ActivationGroup.getSystem().registerGroup(groupDesc);

  String location = "http://localhost:8888/Warehouse/";
  MarshalledObject<ActiveWarehouseConfig> data = null;

  ActivationDesc activationDescription = new ActivationDesc(gid,
  "warehouse.server.ActiveWarehouseImpl", location, data);
  ActiveWarehouse activeWarehouse =
        (ActiveWarehouse)Activatable.register(activationDescription);
  System.out.println("Stub created for the activatable object...");

  Naming.rebind("warehouse", activeWarehouse);

}
catch (RemoteException | MalformedURLException | ActivationException e) { … }
```