Prof. Dr. Claudia Müller-Birn
Institute for Computer Science, Networked Information Systems

Freie Universität Berlin

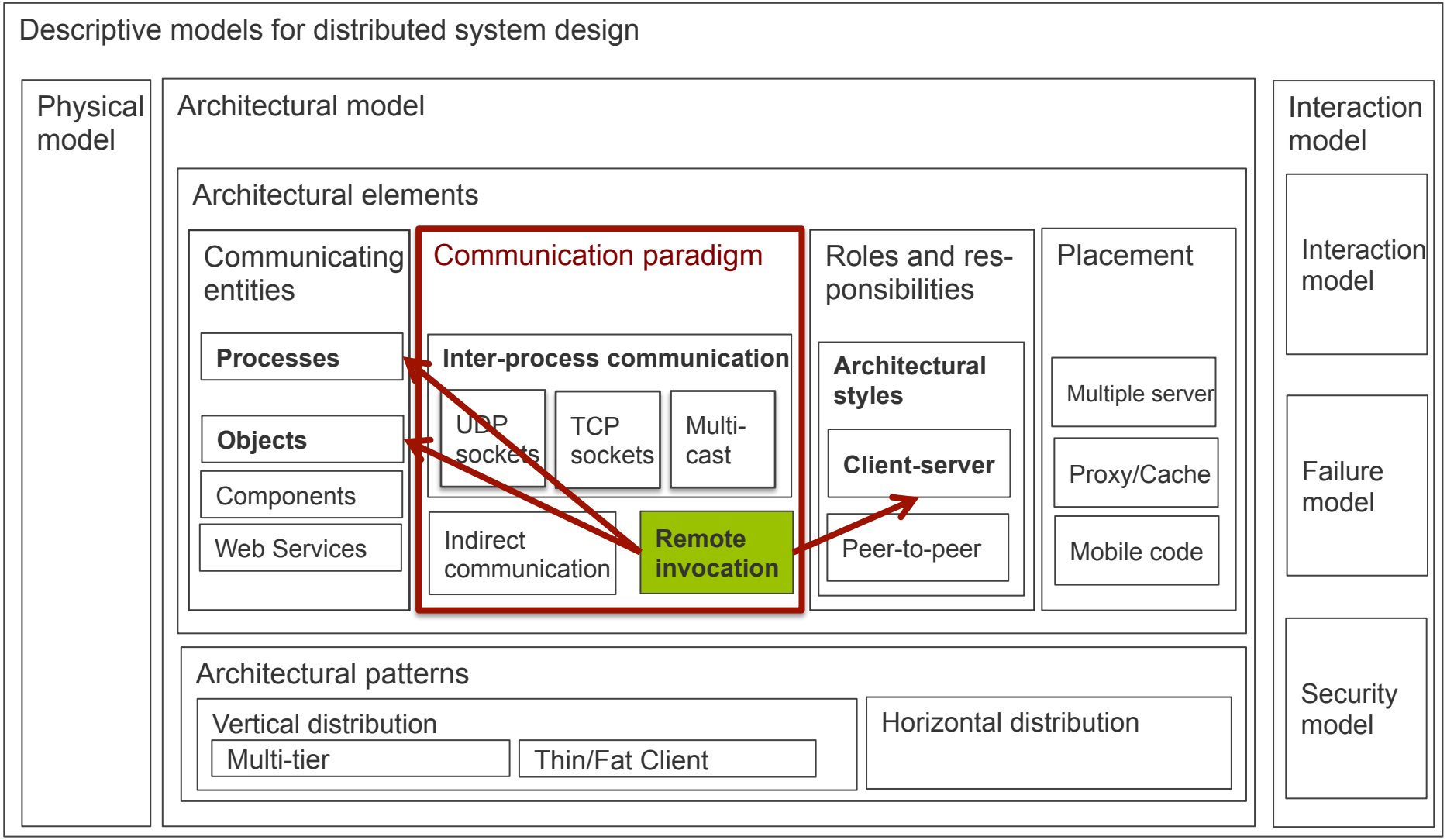
Distributed object component middleware I - Java RMI

Nov 15th, 2011

Netzprogrammierung

(Algorithmen und Programmierung V)

Our topics last week



Our topics today

Implementation of RMI

- The process of remote method invocation
- Communication modules and remote reference module
- RMI software

Generation of classes for proxies, dispatcher and skeleton

Dynamic invocation: An alternative to proxies

Distributed garbage collection algorithm

Java RMI

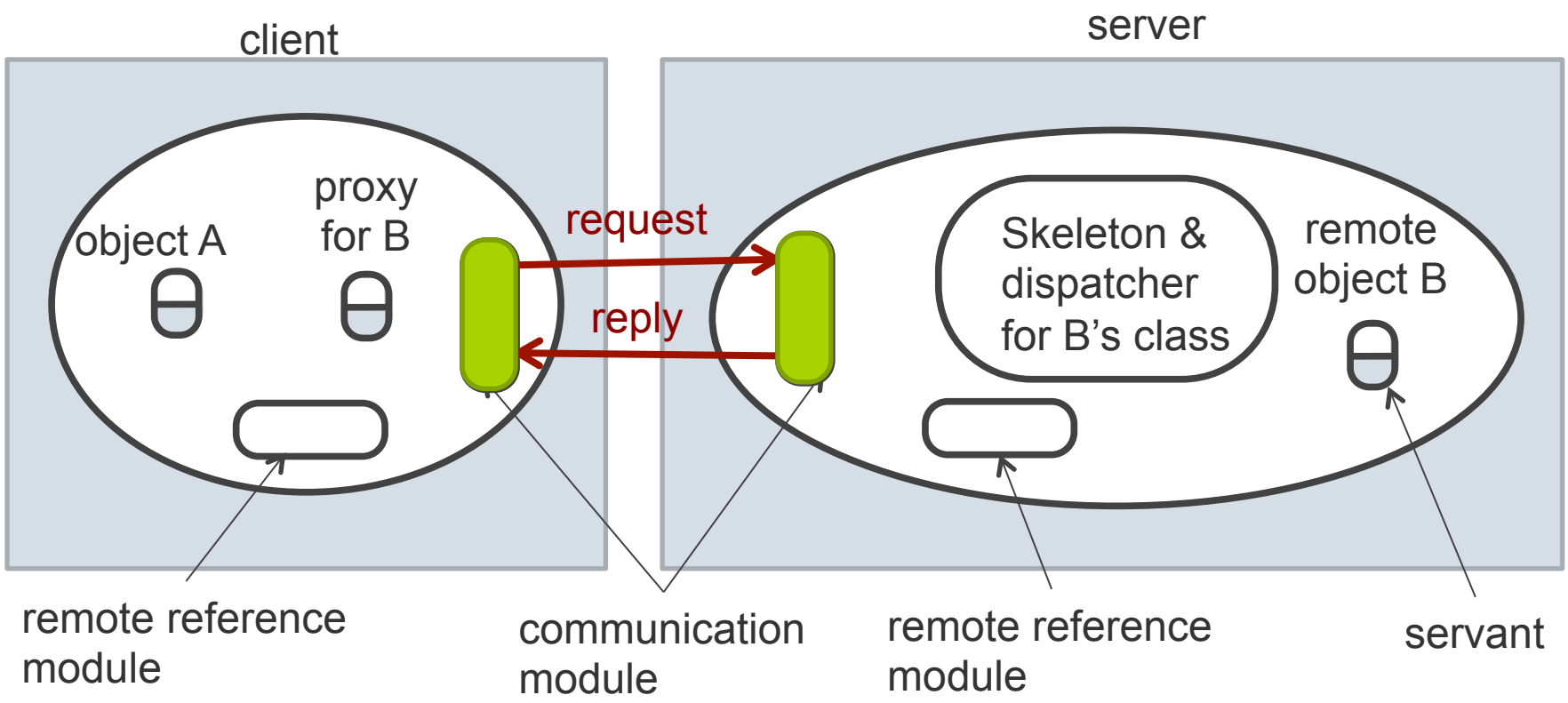
- Introducing a case study
- Parameter and result passing and RMI registry
- Building a client and server programs

Freie Universität Berlin

Distributed object component middleware I - Java RMI

Implementation of RMI

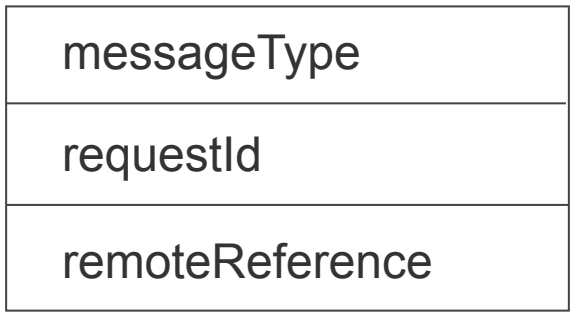
The process of remote method invocation



What does the communication module do?

Two cooperating communication modules carry out the request-reply protocol.

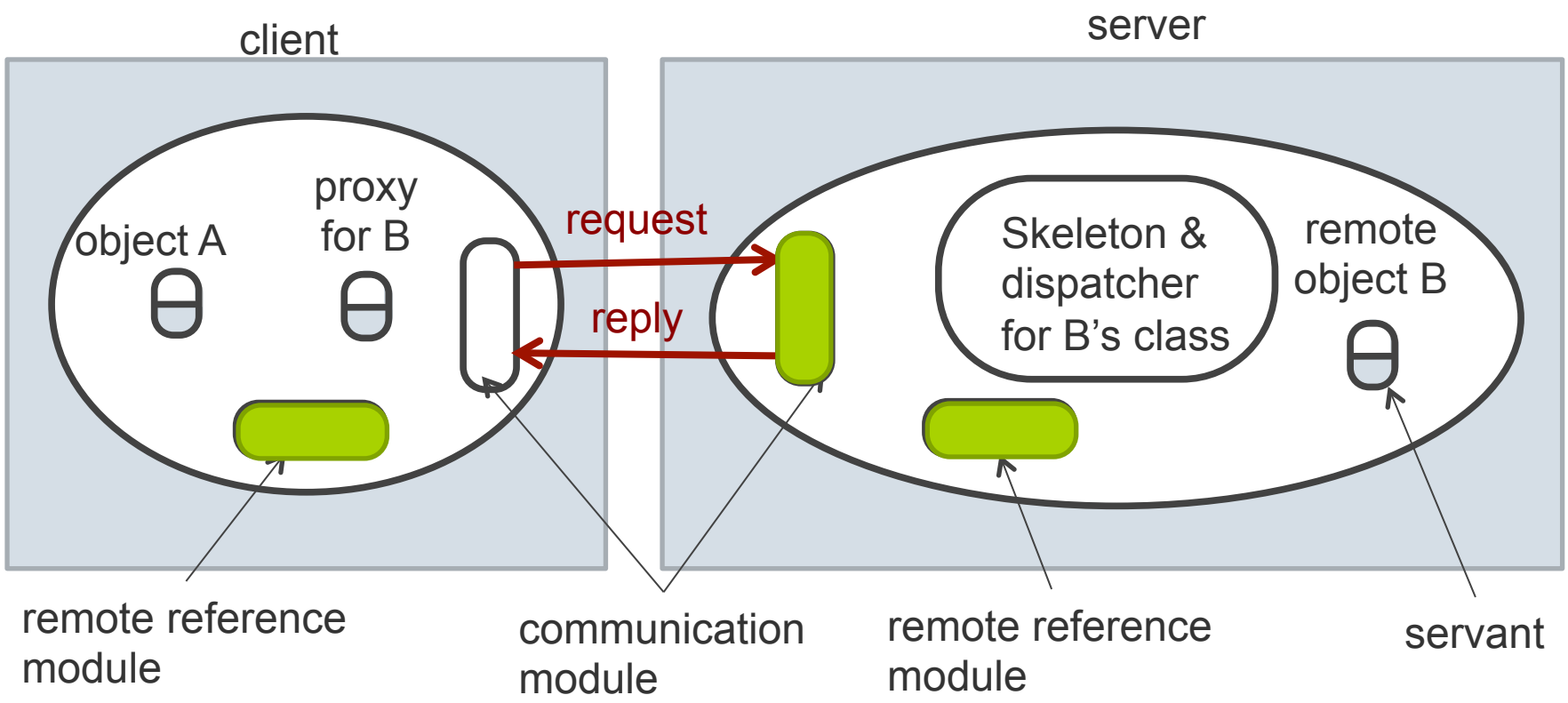
Content of request and reply messages



Communication modules provide together a specified invocation semantics.

The communication module in the server selects the dispatcher for the class of the object to be invoked, passing on the remote object's local reference.

Remote reference module



What does the remote reference module do?

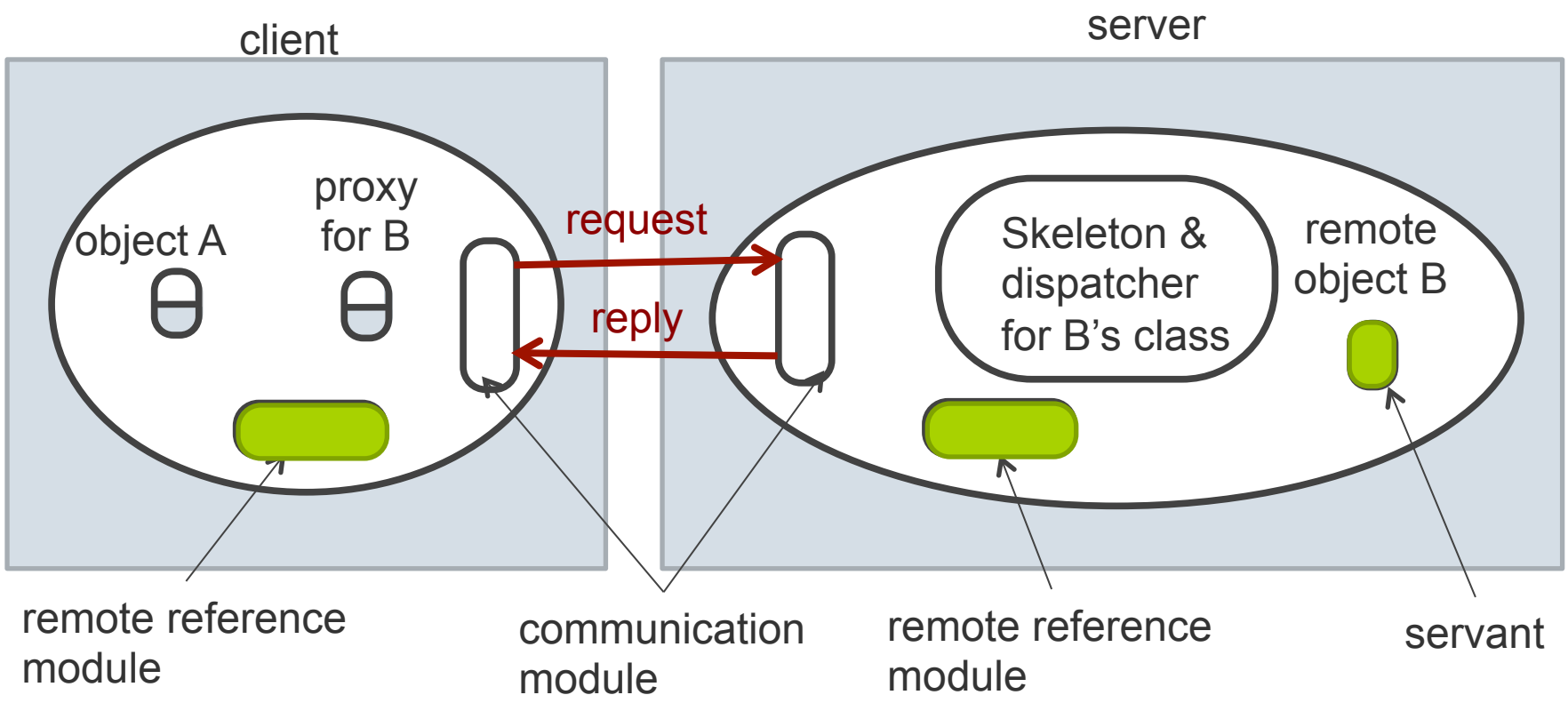
It is responsible for translating between local and remote object references and for creating remote object references.

The remote reference module holds a (*remote object*) table that records the correspondence between local object references in that process and remote object references (which are system-wide).

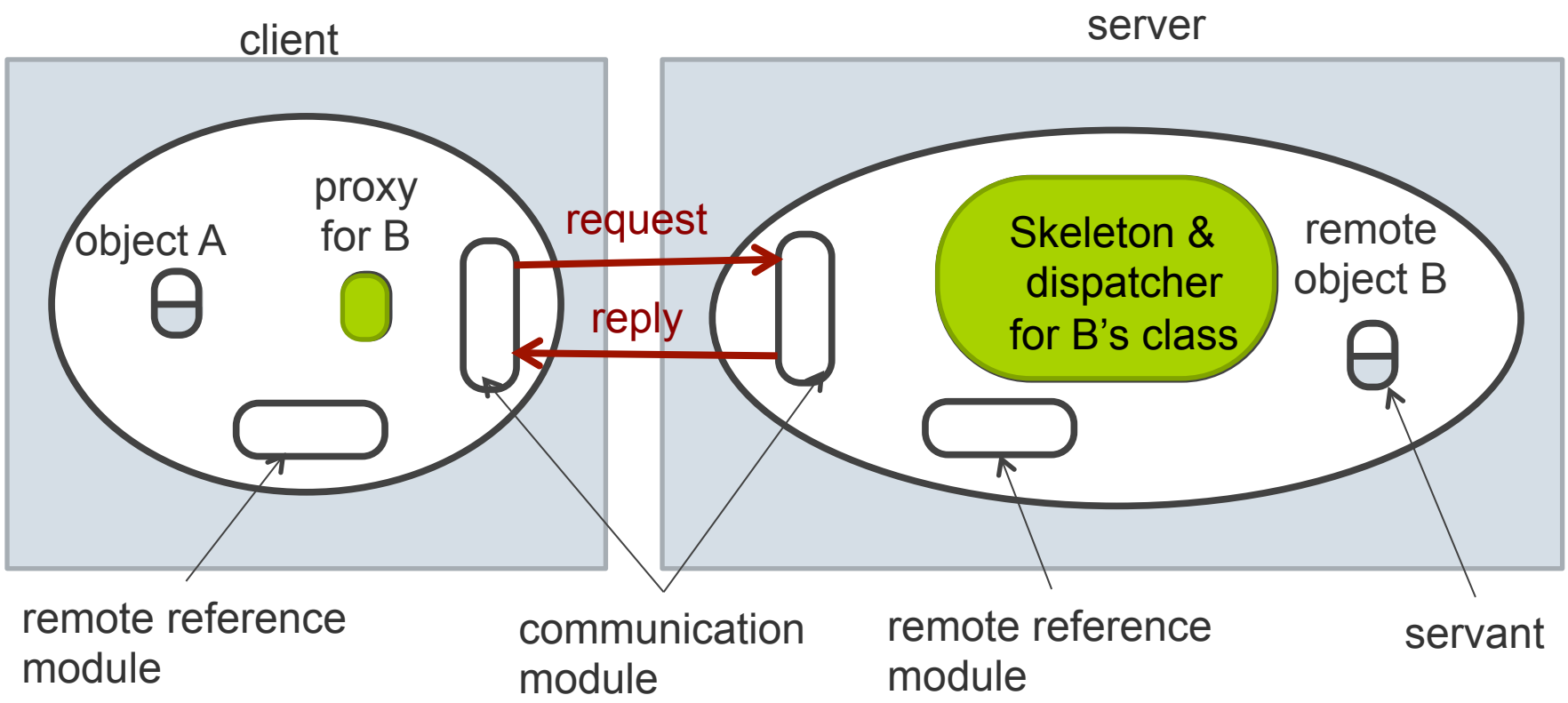
Table includes

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

Remote reference module/servant



RMI software



Generation of classes for proxies, dispatcher and skeleton

Classes for proxies, dispatcher and skeleton are generated automatically by an interface compiler.

In Java RMI

- Set of methods offered by a remote object is defined as a Java interface that is implemented within the class of the remote object
- Java RMI compiler generates the proxy, dispatcher and skeleton classes from the class remote object

Dynamic invocation: An alternative to proxies

Dynamic invocation gives the client access to a generic representation of a remote invocation.

In order to make a dynamic invocation not only information (e.g., name) about the interface of the remote object are included in the remote object reference. Additionally the names of the methods and the types of the argument are required.

When is it useful?

In applications, where some of the interfaces of the remote objects cannot be predicted at design time.

Server and client programs

Server program

- Contains classes for the dispatcher and skeletons, together with the implementations of the classes of all of the servants
- Contains a *initialization* section (responsible for creating and initializing at least one of the servants to be hosted by the server)
- Generally allocates a separate thread for the execution of each remote invocation -> designer of the remote object implementation must allow concurrent executions

Client program

- Contain the classes of the proxies for all of the remote objects that it will invoke
- Require a means of obtaining a remote object reference for at least one of the remote objects held by the server -> binder

Factory methods

Servants are created either in the initialization section or in methods in a remote interface designed for that purpose

Factory method: used to refer to a method that creates servants

Factory object: object with factory methods

Activation of remote objects

A remote object is described as active when it is available for invocation from a running process, whereas it is called passive if it is not currently active but can be made active.

Activation consists of creating an active object from the corresponding passive object by creating a new instance of its class and initialize its instance variables from the store state.

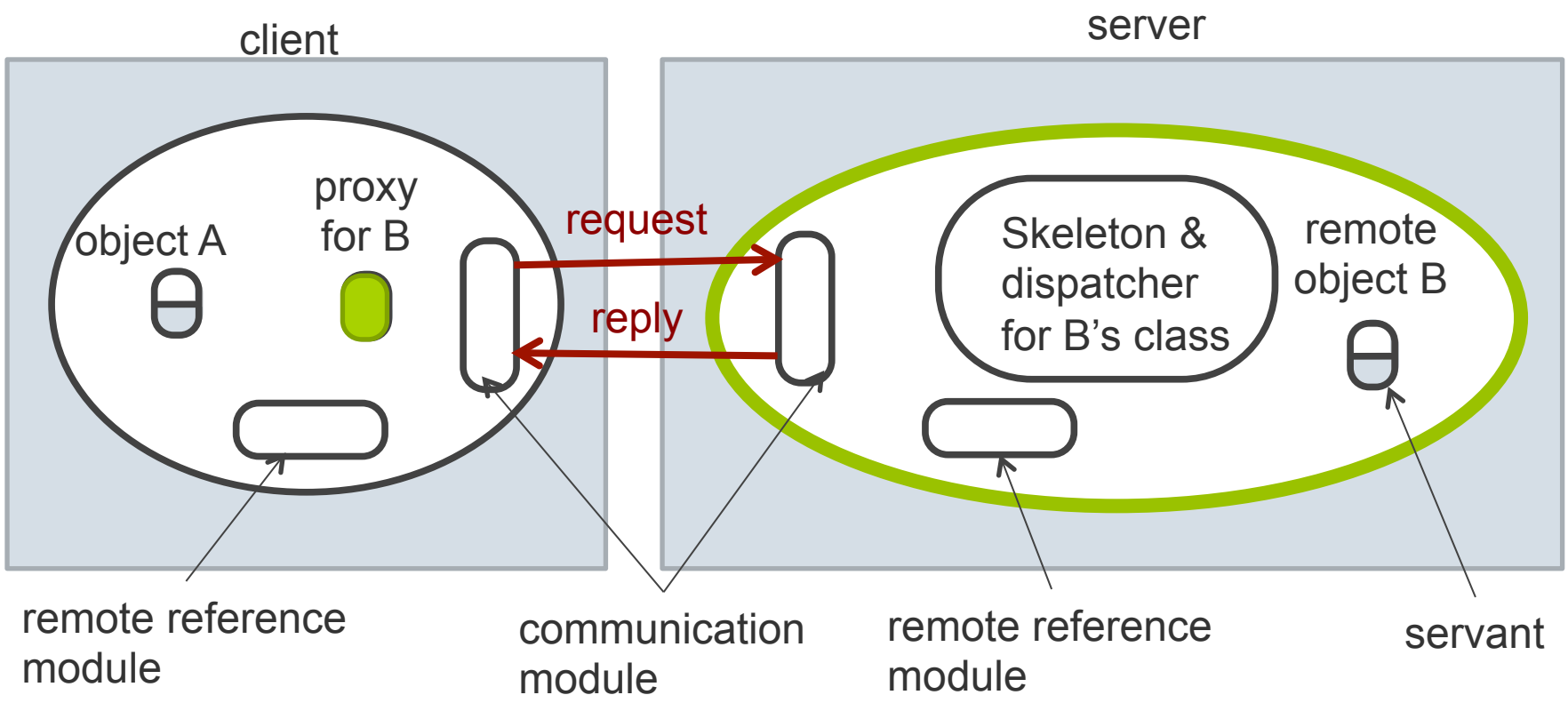
Activator is responsible for

- Registering passive objects that are available for activation
- Starting names server processes and activating remote objects in them
- Keeping track of the locations of the servers for remote objects that it has already activated

Implementation of RMI

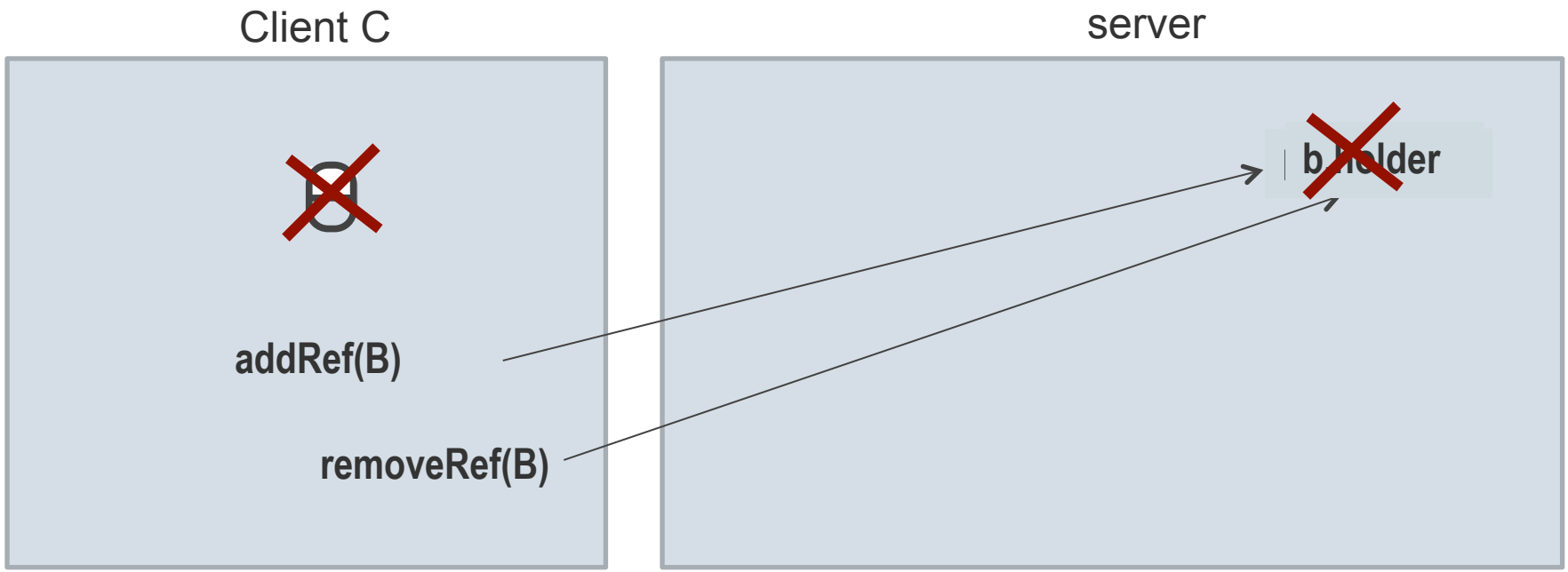
Distributed garbage collection

Java distributed garbage collection algorithm



Java distributed garbage collection algorithm (*cont.*)

Each server process contains a set of names of the processes hold remote object references for each of its remote objects.



Freie Universität Berlin

Distributed object component middleware I - Java RMI

Java RMI

Case study: shared whiteboard

http://www.flickr.com/photos/36567420@N06/

Java Remote interfaces *Shape* and *ShapeList*

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

public interface Shape extends Remote {
      int getVersion() throws RemoteException;
      GraphicalObject  getAllState() throws RemoteException;
}

public interface ShapeList extends Remote {
      Shape newShape(GraphicalObject g) throws RemoteException;
      Vector allShapes() throws RemoteException;
      int getVersion() throws RemoteException;
}
```

Parameter and result passing

In Java RMI, the parameters of a method are assumed to be *input* parameters and the result of a method is a single *output* parameter. Any object that is serializable can be passed as an argument or results in Java RMI.

Passing remote objects

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

Passing non-remote objects

All serializable non-remote objects are copied and passed by value. When a object is passed by value a new object is created in the receiver's process.

Downloading classes

As you know, non-remote objects are passed by value and remote objects are passed by reference as arguments and results of RMI's.

→ If the recipient does not already posses the class of an object passed by value, its code is downloaded automatically.

→ If the recipient of the remote object reference does not already posses the class for a proxy, its code is downloaded automatically.

Advantages:

1. There is no need for every user to keep the same set of classes in their working environment.
2. Both client and server programs can male transparent use of instances of new classes whenever they added.

RMIregistry

The RMIregistry is the binder for Java RMI.

It maintains table mapping textual, URL-styled names to references to remote objects hosted on that computer.

It is accessed by methods of the *Naming* class, whose methods take as an argument a URL-formatted string of the form:

//computerName:port/objectName

The *Naming* class of Java RMIregistry

void rebind (String name, Remote obj)

This method is used by a server to register the identifier of a remote object by name.

void bind (String name, Remote obj)

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

void unbind (String name, Remote obj)

This method removes a binding.

Remote lookup(String name)

This method is used by clients to look up a remote object by name. A remote object reference is returned.

String [] list()

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

RMIregistry (*cont.*)

It is possible to set up a system-wide binding service.

How?

- An instance of the RMI registry must run in the networked environment
- The class *LocateRegistry* (in *java.rmi.registry*) must be used to discover this registry
  - Contains a *getRegistry* method that returns an object of type *Registry* representing the remote binding service:

    public static Registry getRegistry() throws RemoteException

  - After discovery it is necessary to issue a call of *rebind* on this returned *Registry* object to establish a connection with the remote RMIregistry

Freie Universität Berlin

Java RMI

Building a client and server programs

Server program

The server is a whiteboard server which

- represents each shape as a remote object instantiated by a servant that implements the *Shape* interface
- holds the state of a graphical object as well as its version number

- represents its collection of shapes by using another servant that implements the *ShapeList* interface
- holds a collection of shapes in a *Vector*

Java class *ShapeListServer* with *main* method

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

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

Java class *ShapeListServant* implements interface *ShapeList*

Freie Universität Berlin

```
import java.util.Vector;

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

Java client of *ShapeList*

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

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

Callbacks

The clients creates a remote object that implements an interface that contains a method for the server to call. We refer to this as a callback object.

The server provides an operation allowing interested clients to inform it of the remote object references of their callback objects. It records these in a list.

Whenever an event of interest occurs, the server calls the interested clients.

Disadvantages

1. The performance of the server may be degraded by constant polling.
2. Clients cannot notify users of updates in a timely manner.

Remote method invocation

Summary

Freie Universität Berlin

We have we learned?

Freie Universität Berlin

Next class

Distributed object component middleware II (Java RMI)

References

Main resource for this lecture:

George Coulouris, Jean Dollimore, Tim Kindberg: *Distributed Systems: Concepts and Design*. 5th edition, Addison Wesley, 2011