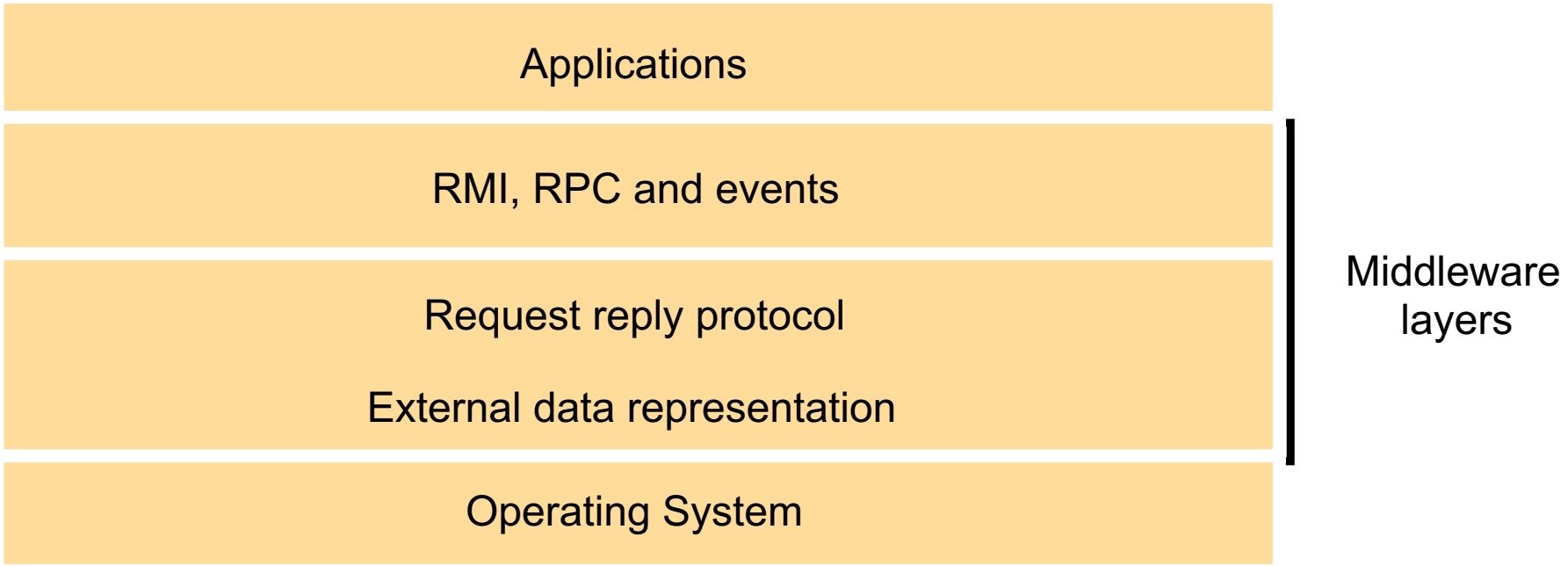
Distributed Systems

Distributed Objects and Remote Invocation

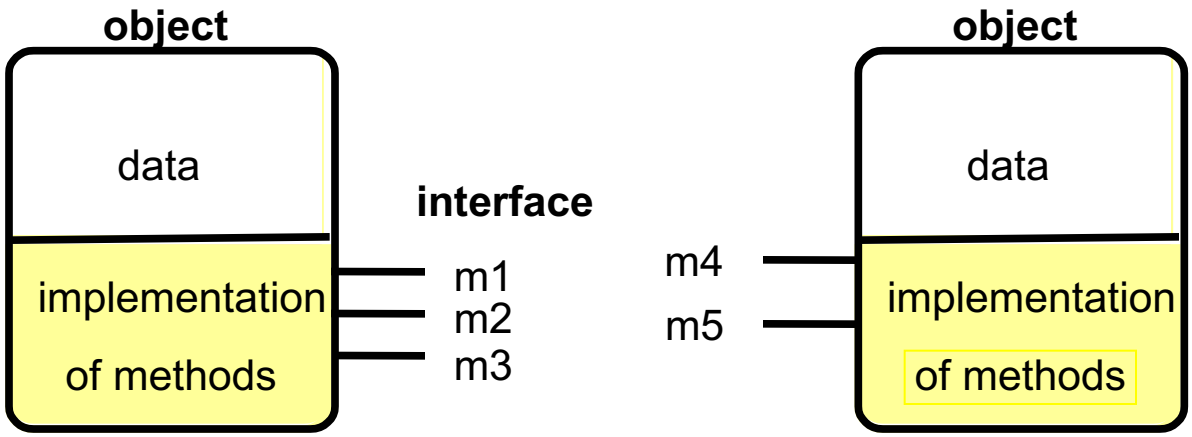
Overview

- Distributed applications programming
  - distributed objects model
  - RMI, invocation semantics
  - RPC
  - events and notifications
- Products
  - Java RMI, CORBA, DCOM
  - Sun RPC

Middleware layers



Objects

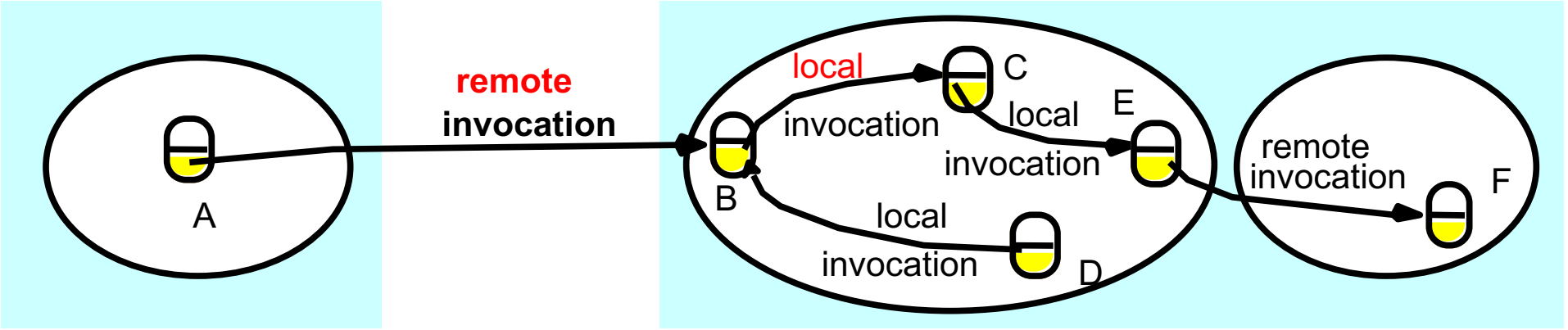


- Objects = Data (attributes) + Operations (methods)
  - encapsulating Data and Methods
  - State of Objects: value of its attributes
- Interact via interfaces:
  - define types of arguments and exceptions of methods

The object (local) model

- OO Programs:
  - (state is distributed in) a collection of interacting objects
- Interfaces
  - the only means to access data, make them remote?
- Actions
  - via method invocation
  - The state of the receiver may be changed
  - A new object may be instantiated
  - interaction, chains of invocations
  - may lead to exceptions, specified in interfaces
- Garbage collection
  - reduced effort, error-free (Java, not C++)

In contrast: distributed object model



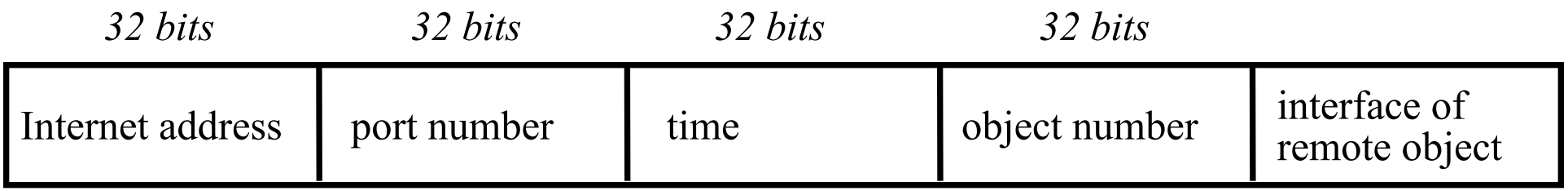
- Objects distributed (client-server models)
- Extend with
  - Remote object reference
  - Remote interfaces
  - Remote Method Invocation (RMI)

Remote object reference

- Object references
  - used to access objects which live in processes
  - can be passed as arguments, stored in variables,...
- Remote object references
  - object identifiers in a distributed system
  - must be unique in space and time
  - error returned if accessing a deleted object
  - can allow relocation (as in CORBA)

Remote Object Reference

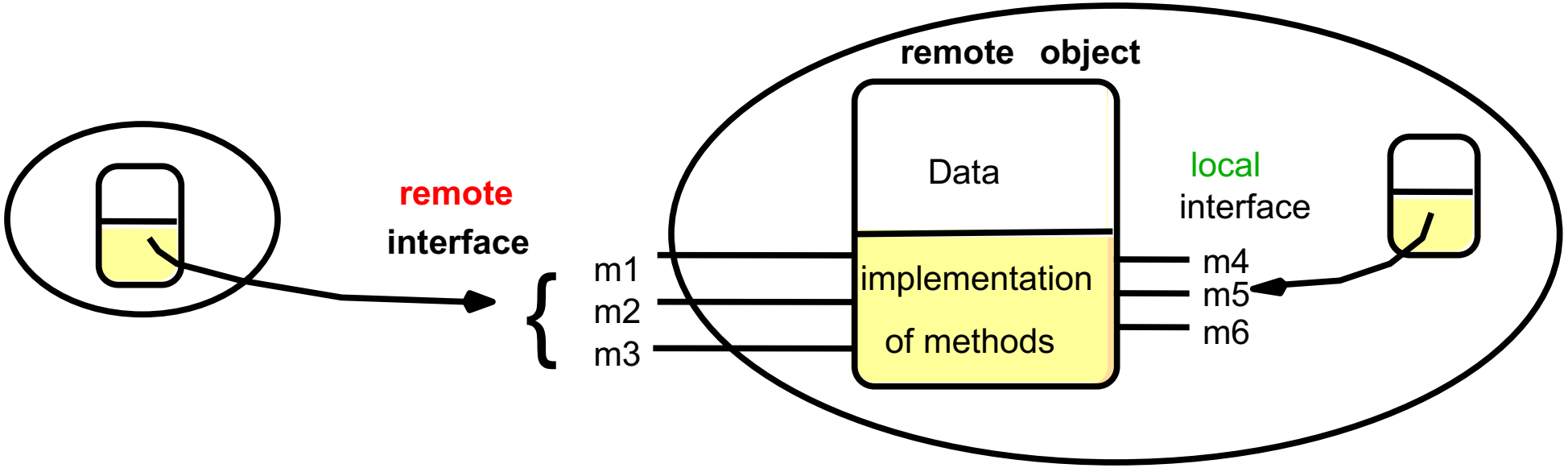
- An identifier for an object that is valid throughout the distributed system
  - must be unique
  - may be passed as argument, hence need external representation



Remote interfaces

- The class of a remote object implements the methods of its remote interface (e.g. as public instance methods in Java)
- Interfaces specify externally accessed
  - variables and procedures
  - no direct references to variables (no global memory)
  - local interface separate
- Parameters
  - input, output or both,
  - instead of call by value, call by reference
- No pointers
- No constructors

Remote object and its interfaces



- CORBA: Interface Definition Language (IDL)
- Java RMI: as other interfaces, keyword *Remote*

Handling remote objects

- Exceptions
  - raised in remote invocation
  - clients need to handle exceptions
  - timeouts in case server crashed or too busy
- Garbage collection
  - distributed garbage collection may be necessary
  - combined local and distributed collector
  - cf Java reference counting

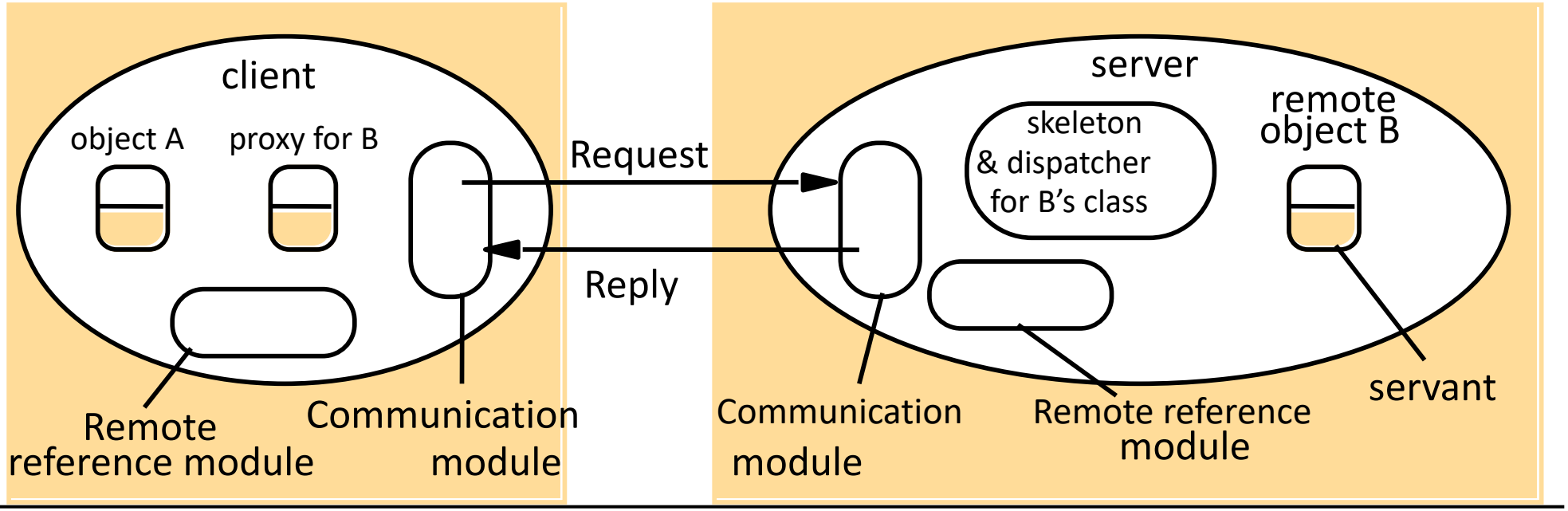
Design issues for RMI

- Invocation semantics
  - Local invocations are executed only once. This is not the case for remote invocations
- Maybe: invocation not guaranteed
- At least once: either a result or an exception (retransmission of request messages): Sun RPC
- At most once: Java and CORBA

Fault tolerance measures			*Invocation semantics*
Retransmit request message	*Duplicate filtering*	*Re-execute procedure or retransmit reply*	
No	Not applicable	Not applicable	*Maybe*
Yes	No	Re-execute procedure	*At-least-once*
Yes	Yes	Retransmit reply	*At-most-once*

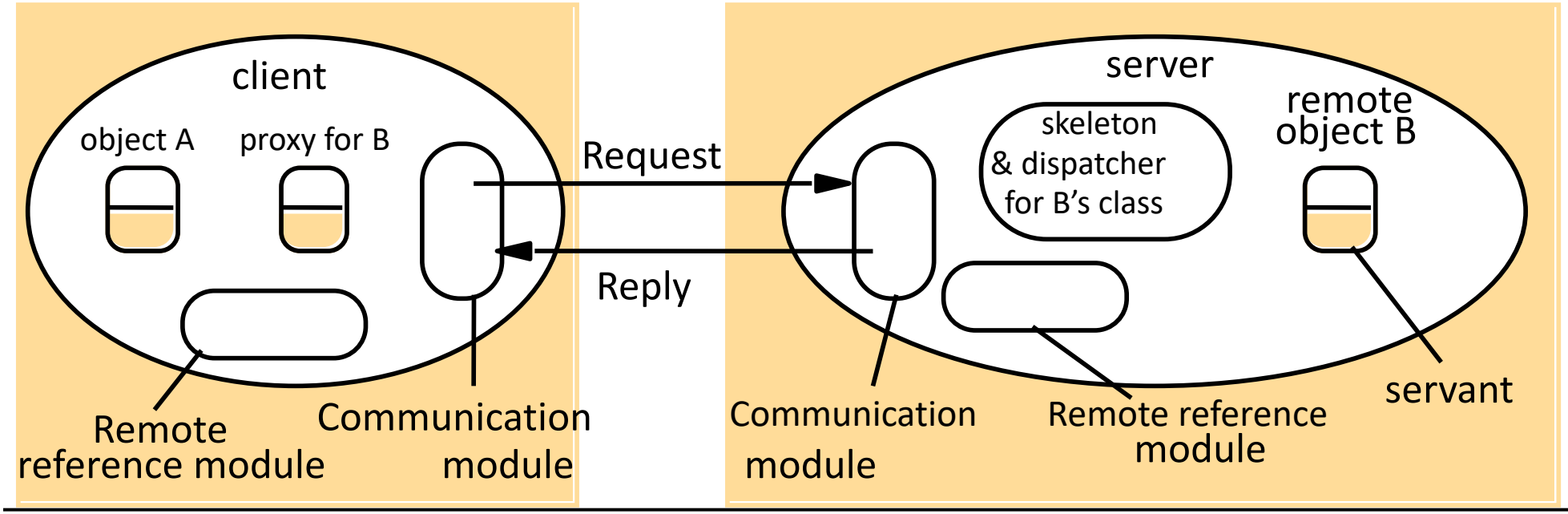
Implementation of RMI

- Application-level Object A invokes a remote method in application-level object B
- Communication module: implements the request-reply protocol
  - use unique message ids (new integer for each message)
  - implement given RMI semantics
  - Receiver module selects dispatcher for the class of the object to be invoked passing on its local reference which it gets from the remote reference module in return for the remote object id in the request message



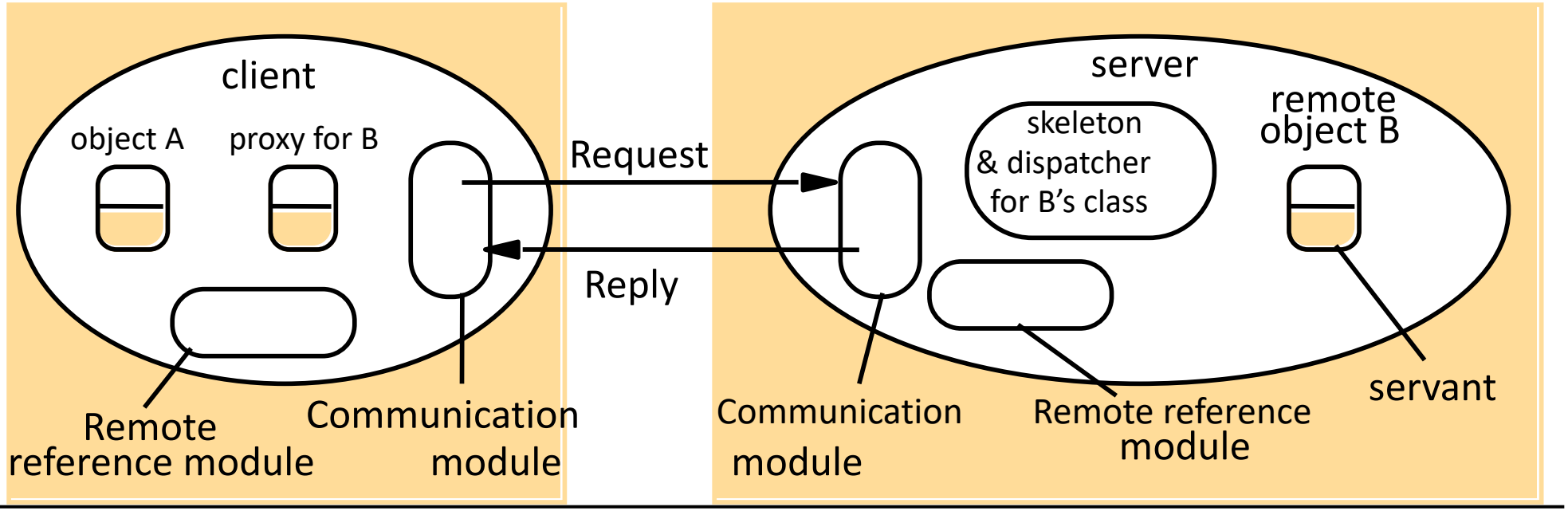
Implementation of RMI

- Proxies: makes RMI transparent to clients by behaving like a local object to the invoker. Instead of executing an invocation, it sends a message
  - One proxy for each of the remote objects



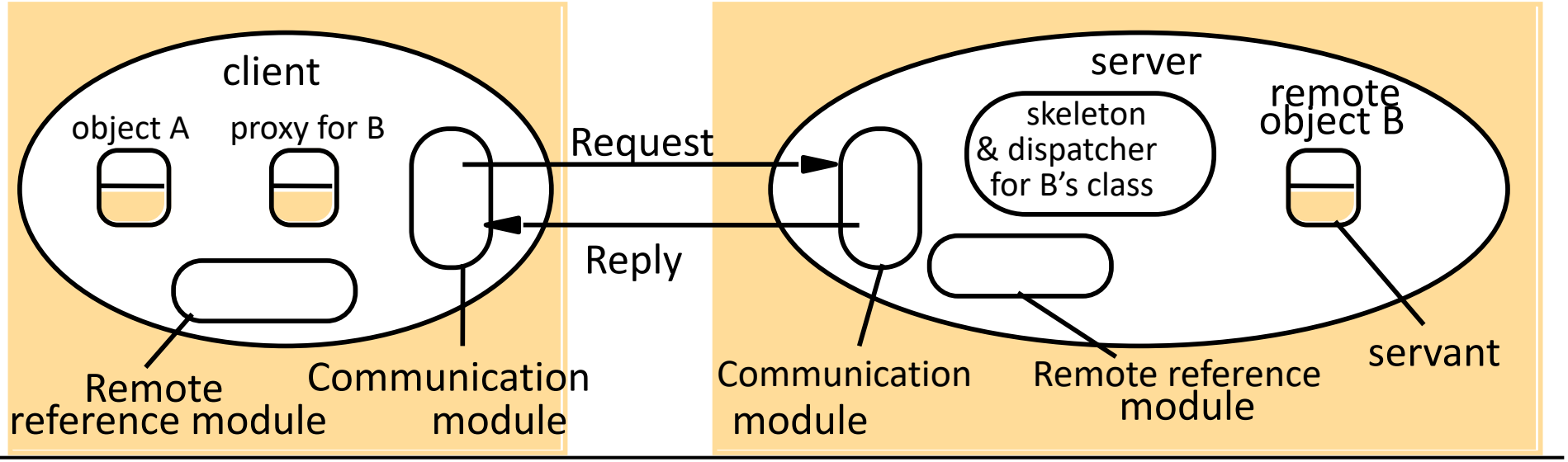
Implementation of RMI

- Remote reference module: translates between local and remote object references and creates remote object references and proxies
  - Remote object table: entries for all remote objects held by the process and entries for all local proxies
- Servants: an instance of a class which provides the body of a remote object. This eventually handles the remote requests passed on by the corresponding skeleton. Lives in a server process.



Implementation of RMI

- A server has one dispatcher and one skeleton for each class representing a remote object.
- Dispatcher receives the request from communication module. Uses the *methodid* to select the appropriate method in the skeleton. Dispatcher and proxies use the same allocation of *methodIds* to the methods of a remote interface
- Skeleton: The class of a remote object has a skeleton which implements the methods in the remore interface. Unmarshalls the arguments in the request and invokes the corresponding method in the servant. Then sends reply to the sending proxy's method



Binding and activation

- The binder
  - mapping from textual names to remote object references
  - used by clients as a look-up service (cf Java RMIregistry)
- Activation
  - objects active (within running process) and passive (=implementation of methods + marshalled state)
  - activation = create new instance of class + initialise from stored state
- Activator
  - records location of passive and active objects
  - starts server processes and activates objects within them

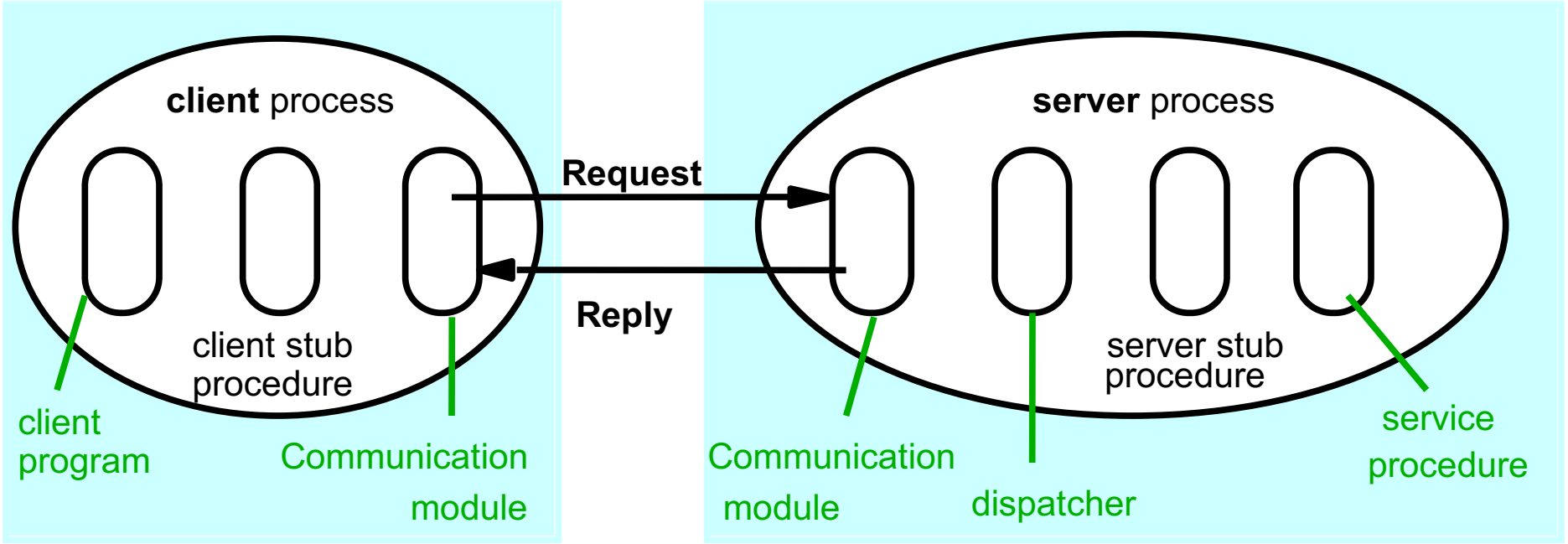
Object location issues

- Persistent object stores
  - stored on disk, state in marshalled form
  - readily available
  - cf Persistent Java
- Object migration
  - need to use remote object reference and address
- Location service
  - assists in locating objects
  - maps remote object references to probable locations

Remote Procedure Call (RPC)

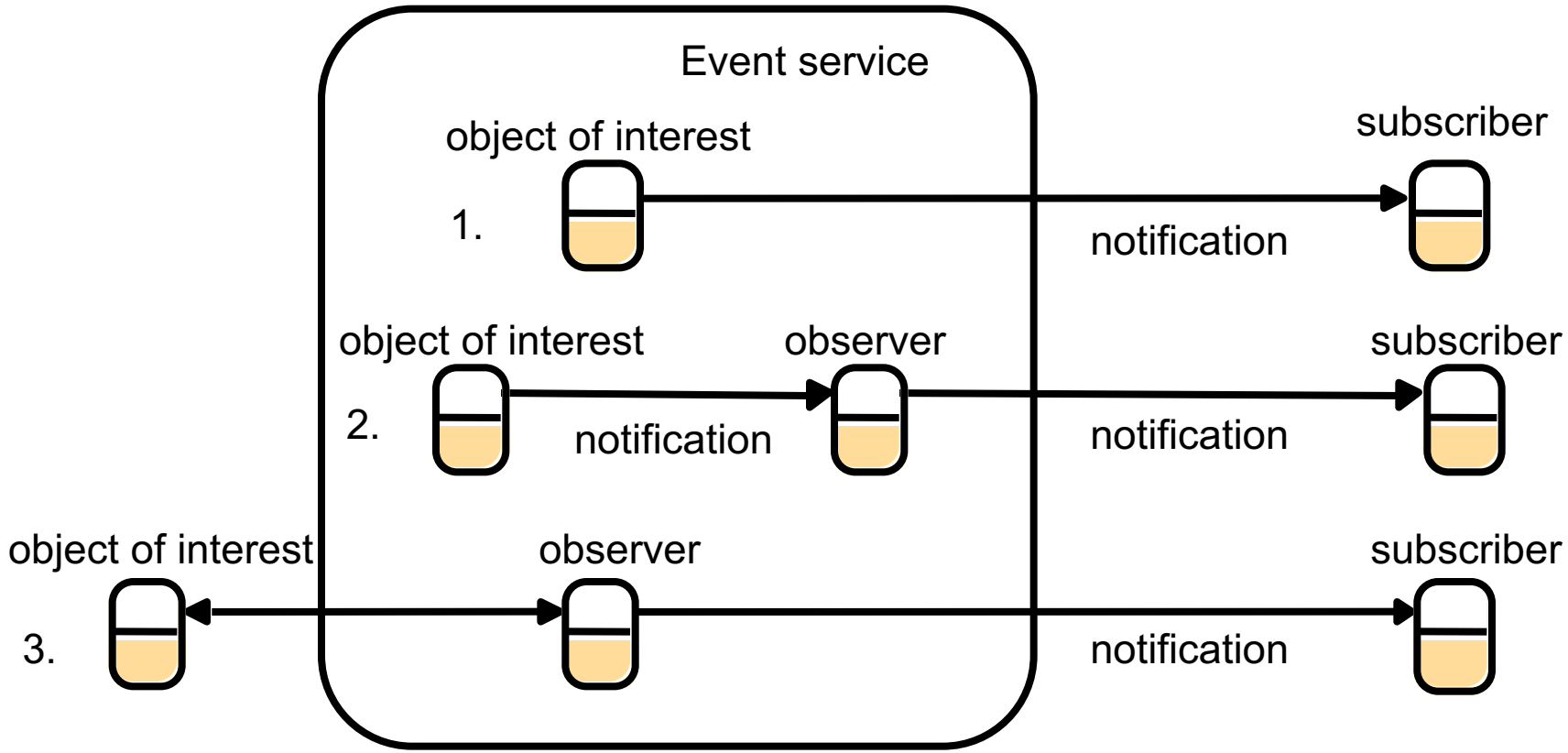
- RPC
  - historically first, now little used
  - over Request-Reply protocol
  - usually at-least-once or at-most-once semantics
  - can be seen as a restricted form of RMI
  - cf Sun RPC
- RPC software architecture
  - similar to RMI (communication, dispatcher and stub in place of proxy/skeleton)

RPC client and server



Implemented over Request-Reply protocol.

Events and Notifications



Java RMI

- Java RMI extends the Java object model to provide support for distributed objects.
- Example: Shared Whiteboard:
  - A group of users sgjhare a common view of the drawing surface containing graphical objects.
  - Server maintains state. Clients inform it about the latest shape their users have drawn. Server allows clients to retrieve the latest drawings by other users
  - Each time a new shape arrives, it increments its id (version number)

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;          1
}
public interface ShapeList extends Remote {
    Shape newShape(GraphicalObject g) throws RemoteException; 2
    Vector allShapes() throws RemoteException;
    int getVersion() throws RemoteException;
}
```

The *Naming* class of Java RMIregistry

void rebind (String name, Remote obj)

This method is used by a server to register the identifier of a remote object by name, as shown in Figure 1 , line 2.

void bind (String name, Remote obj)

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

void unbind (String name, Remote obj)

This method removes a binding.

Remote lookup(String name)

This method is used by clients to look up a remote object by name, as shown in Figure 3 line 1. A remote object reference is returned.

String [] list()

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

Java class *ShapeListServer* with *main* method

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

Java class *ShapeListServant* implements interface *ShapeList*

```
import java.rmi.*;
import java.rmi.server.UnicastRemoteObject;
import java.util.Vector;
public class ShapeListServant extends UnicastRemoteObject implements ShapeList {
      private Vector theList;              // contains the list of Shapes          1
      private int version;
     public ShapeListServant()throws RemoteException{...}
     public Shape newShape(GraphicalObject g) throws RemoteException {      2
          version++;
               Shape s = new ShapeServant( g, version);                        3
               theList.addElement(s);
               return s;
     }
     public  Vector allShapes()throws RemoteException{...}
     public int getVersion() throws RemoteException { ... }
}
```

Java client of *ShapeList*

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

Classes supporting Java RMI