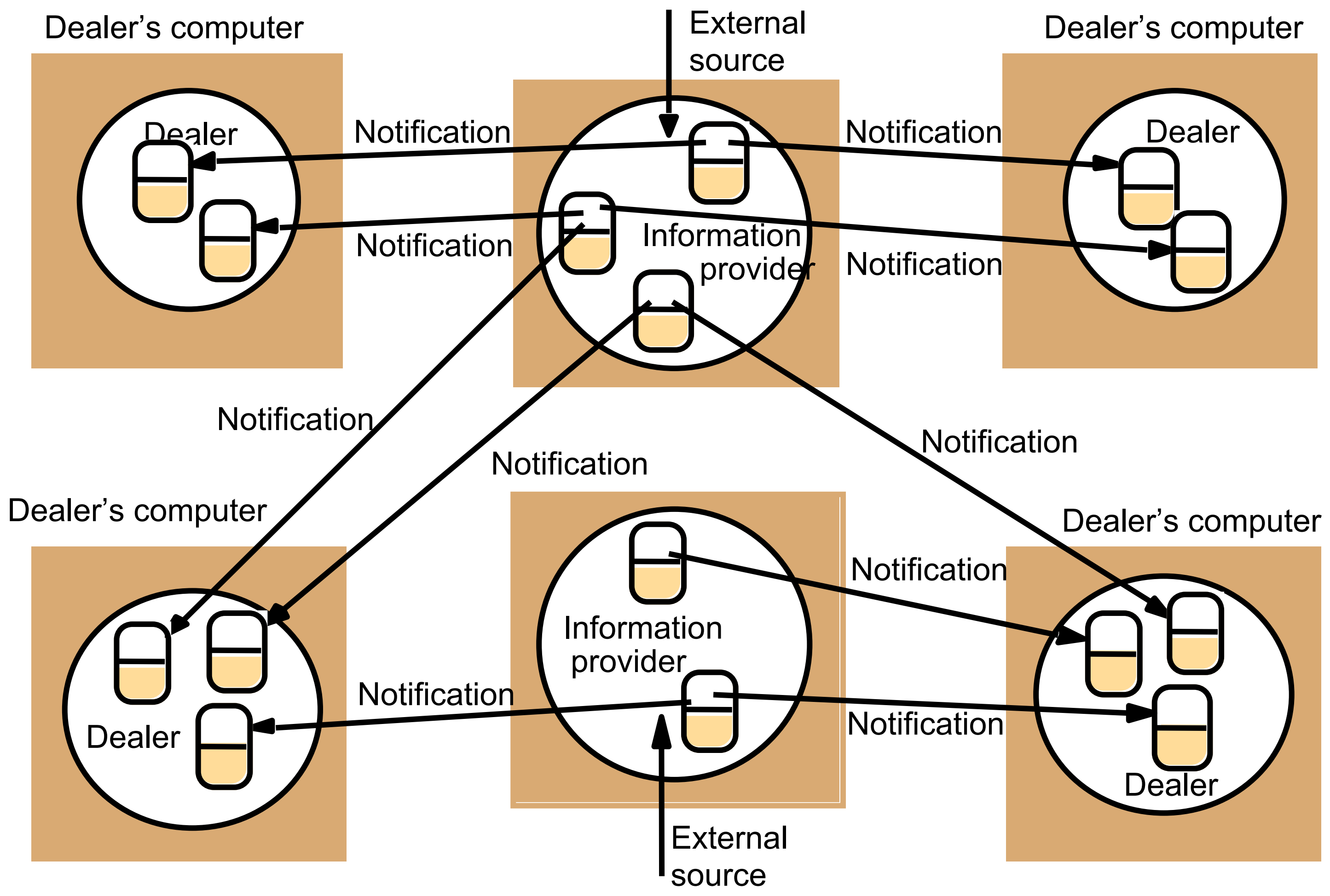
5.4. Events and notifications

- Distributed event-based systems extend local event model
  - Allowing multiple objects at diff. locations to be notified of events taking place at an object
  - Two characteristics: heterogeneous, asynchronous
- Such systems can be useful: stock dealing room
  - Allow dealers using computers to see latest stock prices
  - Information provider process: continuously receives new trading information from external source and applies to the stock object
  - Each such update is regarded as an event
  - The stock object notifies all the dealers who have subscribed
  - Dealer process: creates local object to represent stock to display, local object subscribes to stock object in info. provider process, receives notifications and display to user

5.4. Events and notifications

Dealing room system



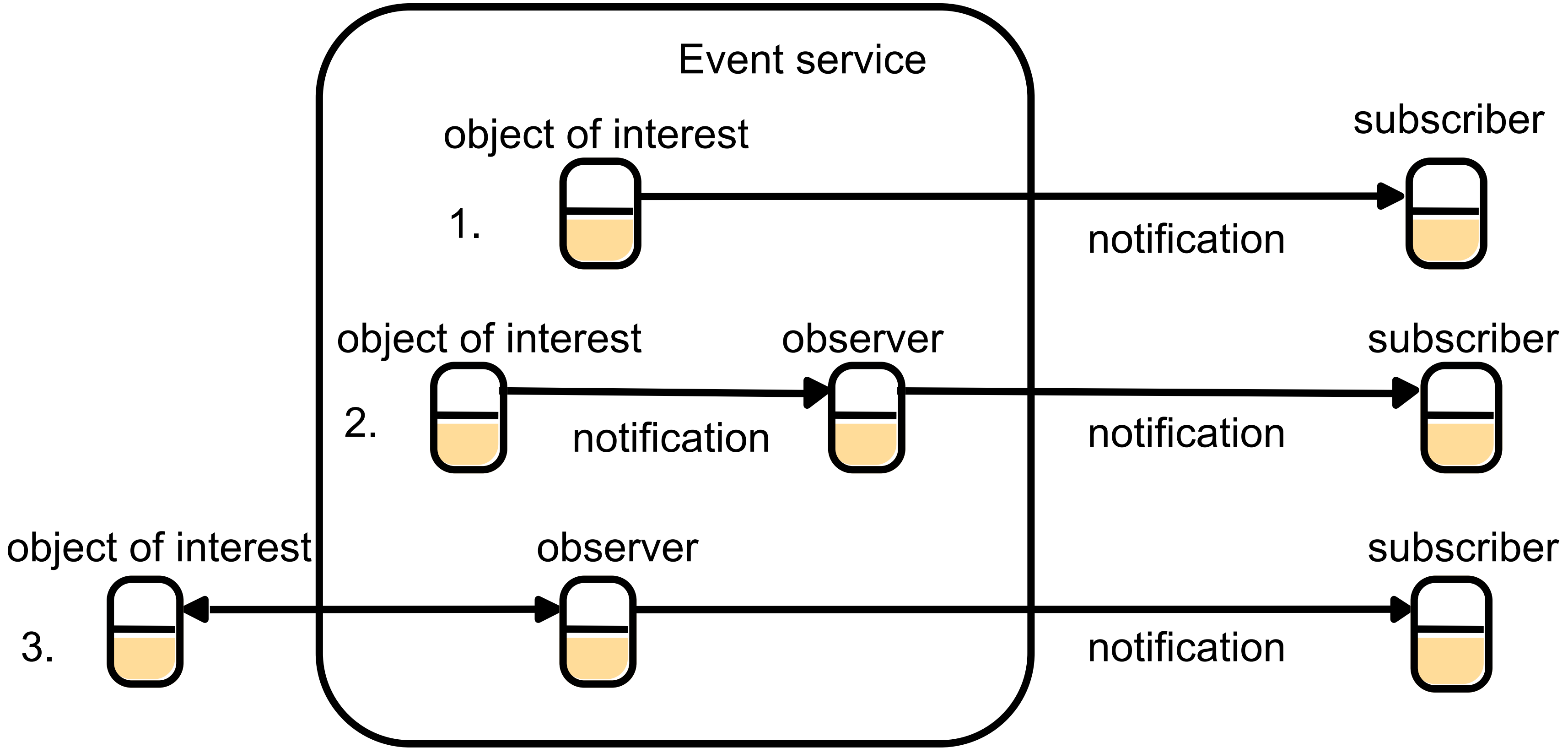
5.4. Events and notifications

- Publish-subscribe paradigm: publisher sends *notifications*, i.e. objects representing events
    - Subscriber *registers interest* to receive notifications

- The object of interest: where events happen, change of state as a result of its operations being invoked
- Events: occurs in the object of interest
- Notification: an object containing information about an event
- Subscriber: registers interest and receives notifications
- Publisher: generate notifications, usually an object of interest
- Observer objects: decouple an object of interest from its subscribers (not important)

5.4. Events and notifications

Architecture for distributed event notification



5.5. Case study: Java RMI

- Extends Java object model to support distributed objects
- Allows objects to invoke methods in remote objects using the same syntax as for local invocations
- Distributed object model is integrated into Java naturally, but the semantics of parameter passing differ because invoker and target are remote
  - *Input* parameters and *output* parameters
- Single language advantage: remote interfaces are defined in Java, do not have to learn IDL (CORBA)

5.5. Case study: Java RMI

- Shared whiteboard example:
  - Allows a group of users to share a common view of a drawing surface containing graphical objects, such as rectangles, lines, circles.
  - Each of the graphical objects is drawn by one of the users
  - Server maintains current state of a drawing
    - provides an operation for clients to inform it about the latest shape
    - allows user to retrieve the latest shapes drawn by other users
    - allows enquiry about version number so as to avoid fetching shapes they already have
      - each shape has a version number attached

5.5. Case study: Java RMI

Java Remote interfaces *Shape* and *ShapeList*

```
import java.rmi.*;  //package providing Remote interface, must throw RemoteException
import java.util.Vector;
public interface Shape extends Remote {  //Shape is a remote interface
    int getVersion() throws RemoteException; //returns version number
    GraphicalObject  getAllState() throws RemoteException; //GraphicalObject serializable
}
public interface ShapeList extends Remote { //ShapeList is a remote interface
    Shape newShape(GraphicalObject g) throws RemoteException; // return remote object
    Vector allShapes() throws RemoteException;
    int getVersion() throws RemoteException;
}
```

5.5. Case study: Java RMI

- Both ordinary and remote objects can be passed as arguments and results
  - Ordinary object (*GraphicalObject*) is passed by value: all non-remote objects
    - New object is created in the receiver’s process, whose methods can be invoked locally, possibly having states changed
  - Remote object (***Shape***) is passed by reference (remote object reference), on which remote methods can be invoked
    - *newShape*: client uses the remote object reference for a remote object of *ShapeList*, calls its method *newShape*, passes to it a *GraphicObject* instance, the server makes a remote object of type *Shape* containing the state of the graphical object and return a remote object reference to it… so that …

5.5. Case study: Java RMI

- Downloading of classes: Java is designed to allow classes to be downloaded from one virtual machine to another
  - Non-remote object: passed by value. If the recipient does not already have the class of the object, its code is downloaded automatically
  - Remote object: passed by reference. If the recipient does not already have the class for a proxy, its code is downloaded automatically
    - Recall, we need a proxy as the local representative for each remote object such that remote methods can be invoked
    - And, proxy implements the remote interface of the remote object, for local invokers, the proxy looks as if it is the remote object
- Advantage: (1) No need for every user to keep the same set of classes in their working environment (2) both client and server can make transparent use of instances of new classes (automatically)

5.5. Case study: Java RMI

- How Java object serialization (chp4) and automatic class downloading work together?
- Section 4.3.2 (Java object serialization), p148 (4th edition):

… Therefore, some information about the class of each object is included in the serialized form. This information enables the recipient to load the appropriate class when an object is deserialized.

The information about a class consists of the name of the class and a version number…

5.5. Case study: Java RMI

- RMIregistry: binder for Java RMI
  - Binder: a separate service in DS, maintains a table containing mappings from textual names to remote object references
- Each server hosting remote objects must run an instance of RMIregistry, which is accessed by methods of *Naming* class
- Methods in the Naming class takes as argument a URL-formatted string " //computerName:port/objectName"
  - computerName and port refer to the location of the RMIregistry; if omitted, refer to local computer and default port
  - The service is not system-wide, clients must direct their lookup enquiries to particular hosts
  - E.g. Naming.rebind("Shape List", aShapeList);

5.5. Case study: Java RMI

The Naming class of Java RMIregistry

void rebind (String name, Remote obj)
  //used by a server to register the identifier of a remote object by name

void bind (String name, Remote obj)
  //alternatively 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)
  //removes a binding.

Remote lookup(String name)
  //used by clients to look up a remote object by name. A remote object reference is returned.

String [] list()
  //returns an array of Strings containing the names bound in the registry.

5.5. Case study: Java RMI

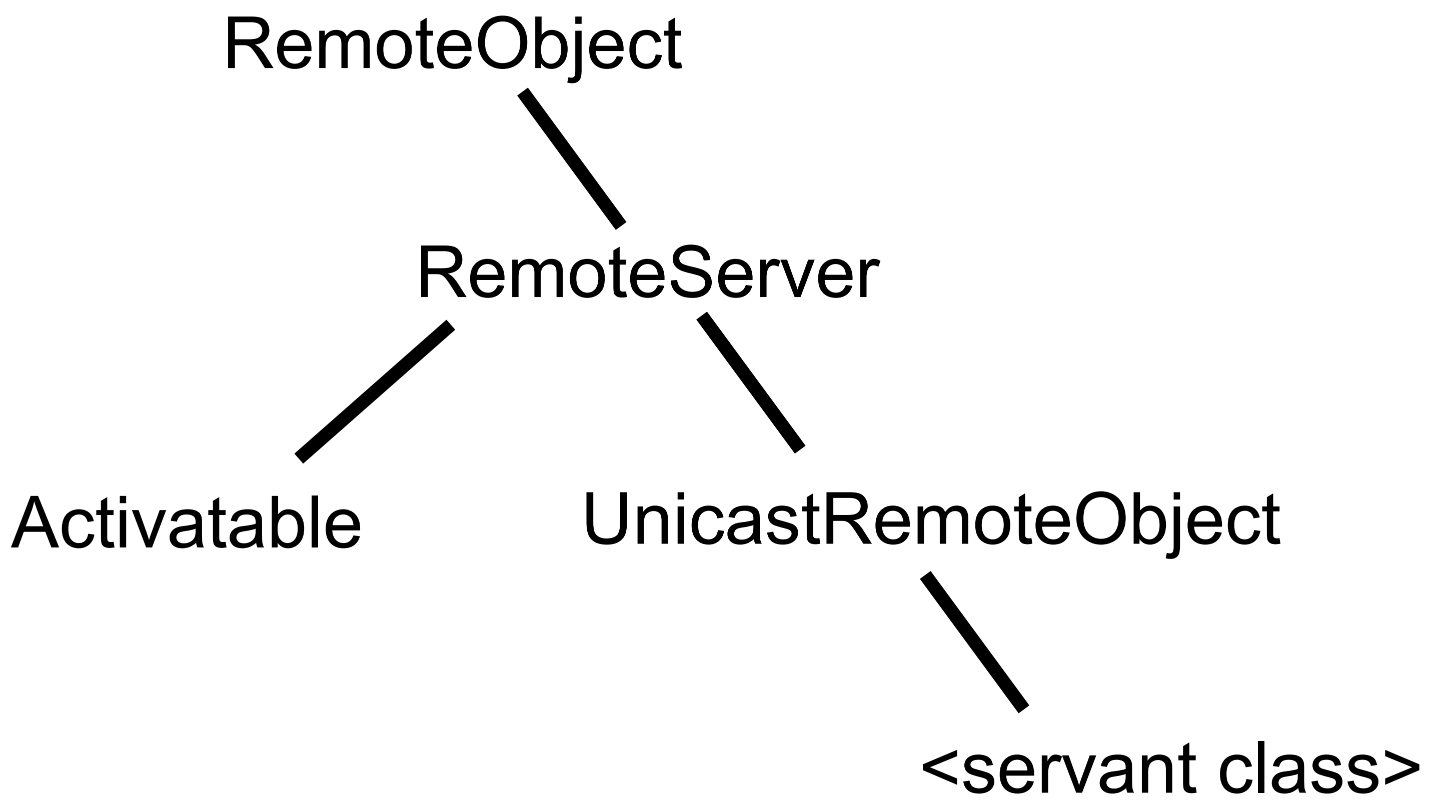
Building client and server programs

- server program
  - main program: binding instances of servant classes
  - main method needs to create a security manager to enable Java security. A default security manager, RMISecurityManager, is provided
  - Note: if an RMI server sets no security manager, proxies and classes can only be loaded from the local classpath, in order to protect the program from code that is downloaded as a result of remote method invocations.
  - servant classes: ShapeListServant and ShapeServant, implementing ShapeList and Shape interfaces respectively
  - servant classes need to extend UnicastRemoteObject, which provides remote object that live only as long as the process in which they are created
  - implementation of servant classes are straightforward, no concern of communication details

5.5. Case study: Java RMI

Classes supporting Java RMI:

Every single servant class needs to extend UnicastRemoteObject



5.5. Case study: Java RMI

- UnicastRemoteObject
  - Automatically creates socket and listens for network requests, and make its services available by exporting them.
- RMISecurityManager ()
  - Needed to download objects from network. The downloaded objects are allowed to communicate only with sites they came from.
  - Default security manager, when none is explicitly set, allows only loading from local file system

5.5. Case study: Java RMI

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();
            Naming.rebind("Shape List", aShapeList );
            System.out.println("ShapeList server ready");
        }catch(Exception e) {
            System.out.println("ShapeList server main " + e.getMessage());}
        }
}
```

5.5. Case study: Java RMI

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
            private int version;
           public ShapeListServant()throws RemoteException{...}
           public Shape newShape(GraphicalObject g) throws RemoteException {
                     version++;
                     Shape s = new ShapeServant( g, version);
                     theList.addElement(s);
                     return s;
           }
           public  Vector allShapes()throws RemoteException{...}
           public int getVersion() throws RemoteException { ... }
}
```

5.5. Case study: Java RMI

- Client program: Java client of ShapeList
  - Set security manager
  - Lookup a remote object reference using the lookup operation of RMIregistry to obtain initial remote object reference
  - Sending RMIs to it or to others discovered during its execution according to needs
  - In the example, to display all the shapes, how would the client program continue?

```
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());}
    }
}
```

5.5. Case study: Java RMI

- Callbacks: server's action of notifying clients about an event
  - Client creates a remote object with method for the server to call
    - callback object
  - Server provides an operation allowing interested client to inform it of the remote object references of their callback objects
  - When event occurs, the server notifies the interested client
- Use of reflection: used to make a generic dispatcher and to avoid the need for skeletons.
  - **Deprecated.** *Skeletons are no longer required for remote method calls in the Java 2 platform v1.2 and greater.*

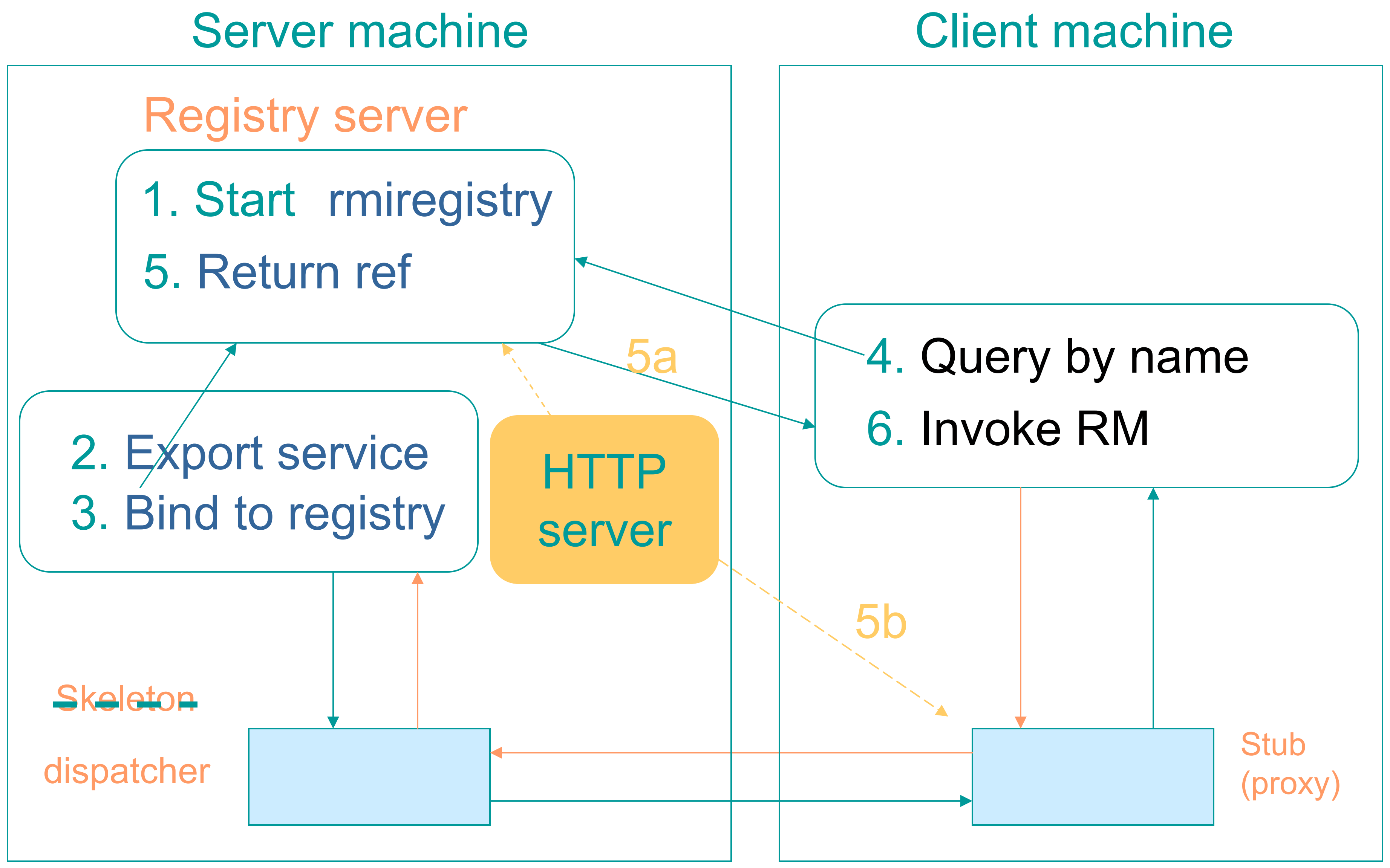
    public interface **Skeleton**

    The Skeleton interface is used solely by the RMI implementation…
- Client proxies are generated by a compiler called *rmic* from the compiled server classes

5.5. Case study: Java RMI

- Reflection: the class of an object can be determined at runtime, and this class can be examined to determine which methods are available, and even invoke these methods with dynamically created arguments
- The key to reflection is the *java.lang.Class*, which allows much information to be determined about a class. This leads onto the other reflection classes such as *java.lang.reflect.Method*. For instance, one can:
  - Determine the class of an object.
  - Get information about a class's modifiers, fields, methods, constructors, and superclasses.
  - Find out what constants and method declarations belong to an interface.
  - Create an instance of a class whose name is not known until runtime.
  - Get and set the value of an object's field, even if the field name is unknown to your program until runtime.
  - Invoke a method on an object, even if the method is not known until runtime.
  - Create a new array, whose size and component type are not known until runtime, and then modify the array's components.
- Reflection tutorial: http://java.sun.com/docs/books/tutorial/reflect/

Java Registry overview



5.5. Case study: Java RMI

- Heterogeneity is an important challenge to designers :
  - Distributed systems must be constructed from a variety of different networks, operating systems, computer hardware and programming languages.
    - The Internet communication protocols mask the difference in networks and middleware can deal with the other differences.
- External data representation and marshalling
  - CORBA marshals data for use by recipients that have prior knowledge of the types of its components. It uses an IDL specification of the data types
  - Java serializes data to include information about the types of its contents, allowing the recipient to reconstruct it. It uses reflection to do this.
- RMI
  - each object has a (global) remote object reference and a remote interface that specifies which of its operations can be invoked remotely.
  - local method invocations provide exactly-once semantics; the best RMI can guarantee is at-most-once
  - Middleware components (proxies, skeletons and dispatchers) hide details of marshalling, message passing and object location from programmers.