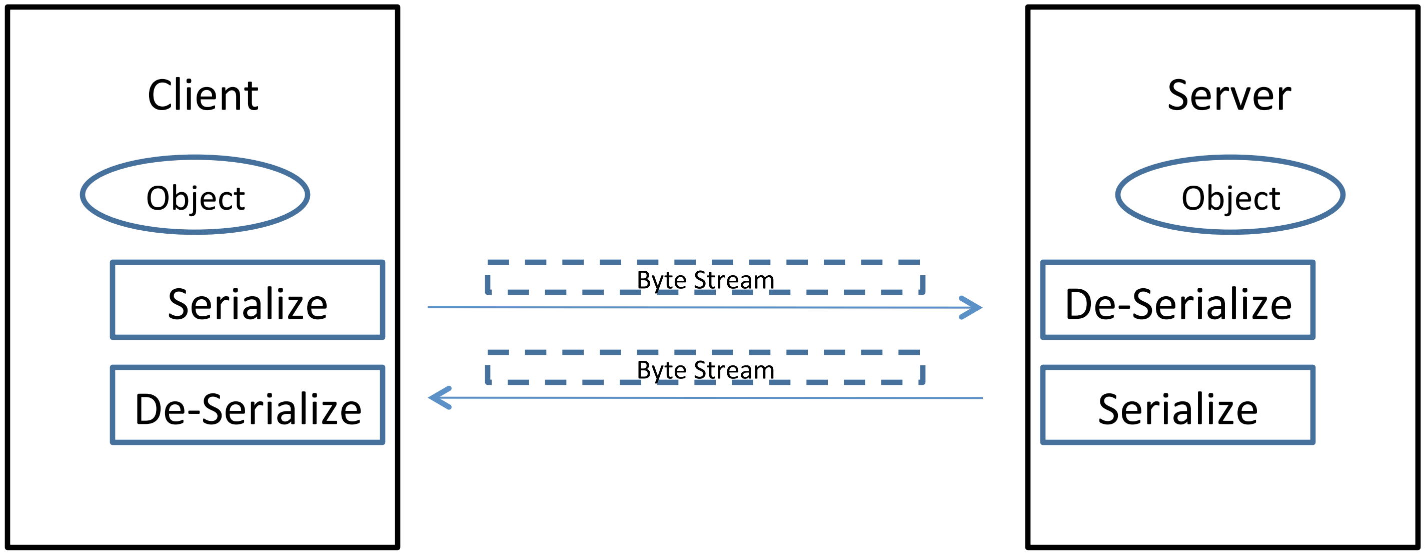
Serialisation, Callbacks, Remote Object Activation

CS3524 Distributed Systems

Lecture 03

Serialization



- How to pass information between client and server?

Serialization

- How to pass information between client and server?
  - So far, we have passed instances of java.lang.String from client to server
- How to pass more complex, programmer-defined data structures?
  - We need mechanisms to convert such data structures into and from byte streams – these byte streams can then be passed between client and server

Serialisation

- Java provides a simple means for object serialisation:
  - The interface **`java.io.Serializable`**

```
public class MyClass implements java.io.Serializable {
    . . .
}
```

- If Java classes implement this interface, then objects instantiated from these classes can be translated into a byte stream that is sent across a network
- This byte stream is sufficient for the reconstruction of the serialized object when it is read by the receiving program

Java Streams

- Java uses input/output “streams” to handle streams of data:
  - Byte Streams: e.g. **`FileInputStream`**, **`FileOutputStream`**
  - Character Streams: e.g. **`FileReader`**, **`FileWriter`**
  - Buffered Streams: e.g. **`BufferedReader`**, **`BufferedWriter`**
  - Object Streams: e.g. **`ObjectInputStream`**, **`ObjectOutputStream`**
- Java provides means to read / write objects from/to “streams”
  - Provided methods: readObject(), writeObject() etc.
- Usage:
  - Scanning and Formatting
  - I/O from the Command Line
  - Sending / receiving values and objects via a network

- Detailed Information
  - http://docs.oracle.com/javase/tutorial/essential/io/

Writing Objects

Nested Streams

- Example: write objects to a file
  - First, create a FileOutputStream to a file called “tmpfile.dat”
  - Second, construct an ObjectOutputStream over the FileOutputStream – this is needed in order to serialize any Java objects written to the file
  - **Note the use of nested Streams** !
  - With the stream objects in place
    - write a String to the file
    - construct a Date object and write it to the same file – the ObjectOutStream helps to serialize this Date object

```
FileOutputStream   out = new FileOutputStream ( “tmpfile.dat” ) ;
ObjectOutputStream s   = new ObjectOutputStream ( out ) ;
s.writeObject ( “Today” ) ;
s.writeObject ( new Date() ) ;
s.flush() ;
```

  - Primitive data types may also be written through methods such as writeInt(), writeFloat(), etc

Reading Objects

Nested Streams

- Example: read objects from a stream
  - This reads in the String and the Date objects previously written to the file called “tmpfile.dat” via a file output stream:

```
FileInputStream   in = new FileInputStream ( ‶tmpfile.dat″ ) ;
ObjectInputStream s  = new ObjectInputStream ( in ) ;
String today = (String) s.readObject();
Date   date  = (Date)   s.readObject();
```

  - The **`readObject()`** method is used to read in objects from the file returning a **`java.lang.Object`** each time.
  - It deserializes objects from the stream and traverses its reference to other objects recursively to deserialise all objects that can be reached from it
  - You can also use **`readInt()`**, **`readFloat()`** etc.

Customising Serializable

- An object is serializable if it implements the Serializable interface

```
public class MyClass implements java.io.Serializable {
    . . .
}
```

- How can we customise serialisation – write our own serialiser:
  - In case of the ObjectOutputStream, the method **defaultWriteObject()** can be implemented (overwritten) to influence serialization behaviour
  - In case of the ObjectInputStream, the method **defaultReadObject()** can be implemented (overwritten) to influence serialization behaviour
  - Methods readObject() and writeObject() can be implemented to customise read/write behaviours

```
public class MyClass implements java.io.Serializable {
    private void writeObject (java.io.ObjectOutputStream out) throws IOException
    { …. }
    private void readObject (java.io.ObjectInputStream in) throws IOException, ClassNotFoundException
    { …. }
}
```

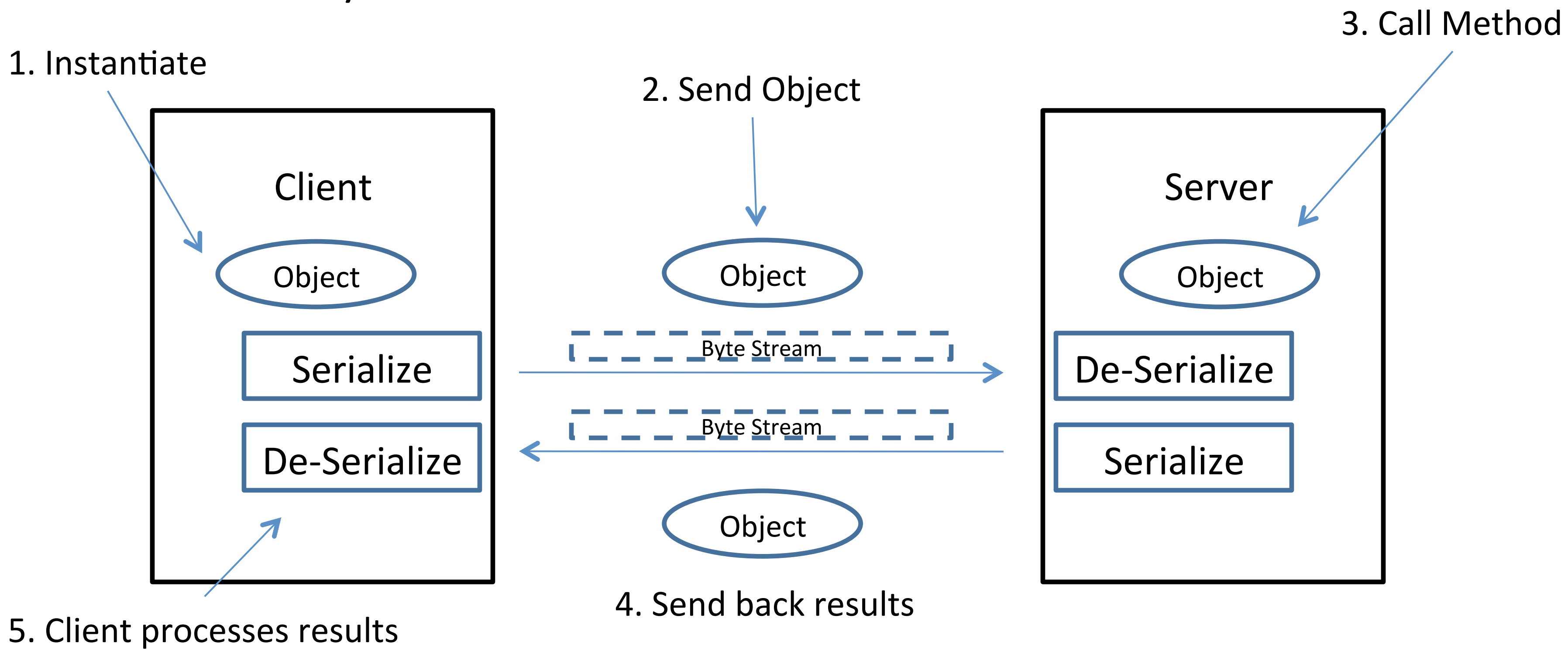
Using Serialisation

Writing Mobile Code

Parameter Passing

Serialisation: Mobile Code

- We can use serialisation to create “mobile code” – we can send whole Java objects as “intelligent” messages from a client to a server
  - The serialised message object may contain methods that can then be called by the server



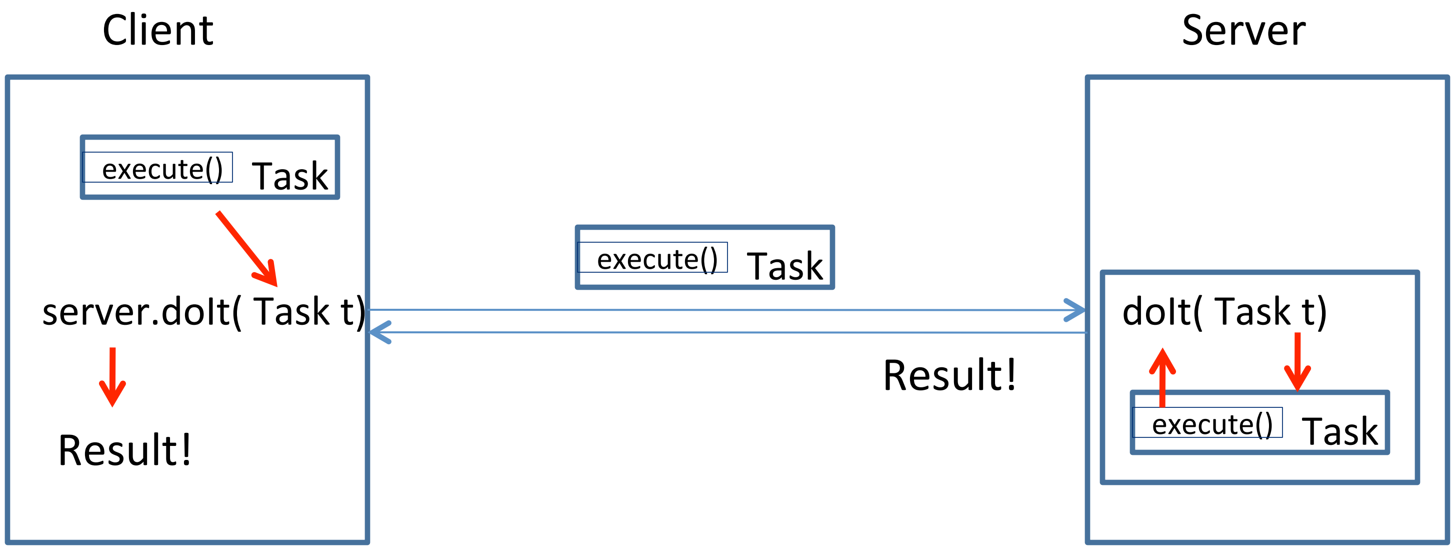
Serialisation: Mobile Code

- For example:
  - Client sends “intelligent message to server that contains code to execute a query
  - Server calls a method on the received message object, which executes this query on the server
  - The result of such a query can be collected by the message object
  - The message object is then sent back to the client with the processed data as its payload
  - The message may visit even multiple servers before being returned to the client

Exercise “Task”

Send a Task Object, return a result

- (See also http://docs.oracle.com/javase/tutorial/rmi/)
- We want to specify messages that can be executed by a server
  - These messages encode particular actions or “tasks” that the server should perform, e.g.:
    - A “shout task”: transform a string into uppercase
    - A “repeat lines task”: the message object transports a String of text to the server side, the server invokes “execute()”, which will create a new String of text that contains the original string a specified number of times



Exercise: “Task”

- We start by defining a serialisable superclass of messages, providing a method by which the message can be called upon to execute its particular task

```
import java.io.* ;
public abstract class Task implements Serializable
{
   public Object execute() { return null ; }
}
```

- By extending this superclass and overriding the execute() method, we can specify a number of different tasks, which are “requests” to the server
- The execute() method returns Object – this is the Superclass of all Java objects, therefore any object can be returned (even Task itself)

Shout Task

- The ShoutTask:
  - Is instantiated by the client – the constructor requires a String as an input parameter – this is the message
  - The server will receive this Task object and invoke the "execute()" method
  - when executed, it will change the message transported within the Task object to upper case
  - A new String is returned as an object by the execute() method – this String object will be received by the client as a result of the task execution

```
public class ShoutTask extends Task
{
   String _msg = null ;
   public ShoutTask ( String msg )
   {
      this._msg = msg ;
   }
   public Object execute()
   {
      return new String ( _msg.toUpperCase() ) ;
   }
}
```

Lines Task

- We create a new String object lines that contains a concatenation of multiple _msg Strings.

```
public class RepeatLinesTask extends Task
{
   String _msg = null ;
   int    _n   = 0 ;

   public RepeatLinesTask ( String msg, int n )
   {
      this._msg = msg ;
      this._n   = n   ;
   }

   public Object execute()
   {
      String lines = new String ( _msg ) ;
      for ( int i = 1; i < _n; i++ ) lines = lines + "\n" + _msg ;
      return lines ;
   }
}
```

Task Service: Interface

- We have to develop a remote service object that can process these tasks (we adapt the rmishout example):

```
public interface TaskServiceInterface extends Remote
{
   public Object doit ( Task t ) throws RemoteException ;
}
```

Task Service: Implementation

- The Task service implementation contains the remote method doIt() that returns an Object

```
public class TaskServiceImpl
             extends java.rmi.server.UnicastRemoteObject
             implements TaskServerInterface
{
   public TaskServiceImpl() throws RemoteException
   { . . . }

   public Object doit ( Task t ) throws RemoteException
   {
      return t.execute() ;
   }
}
```

Parameter Passing “by Value”

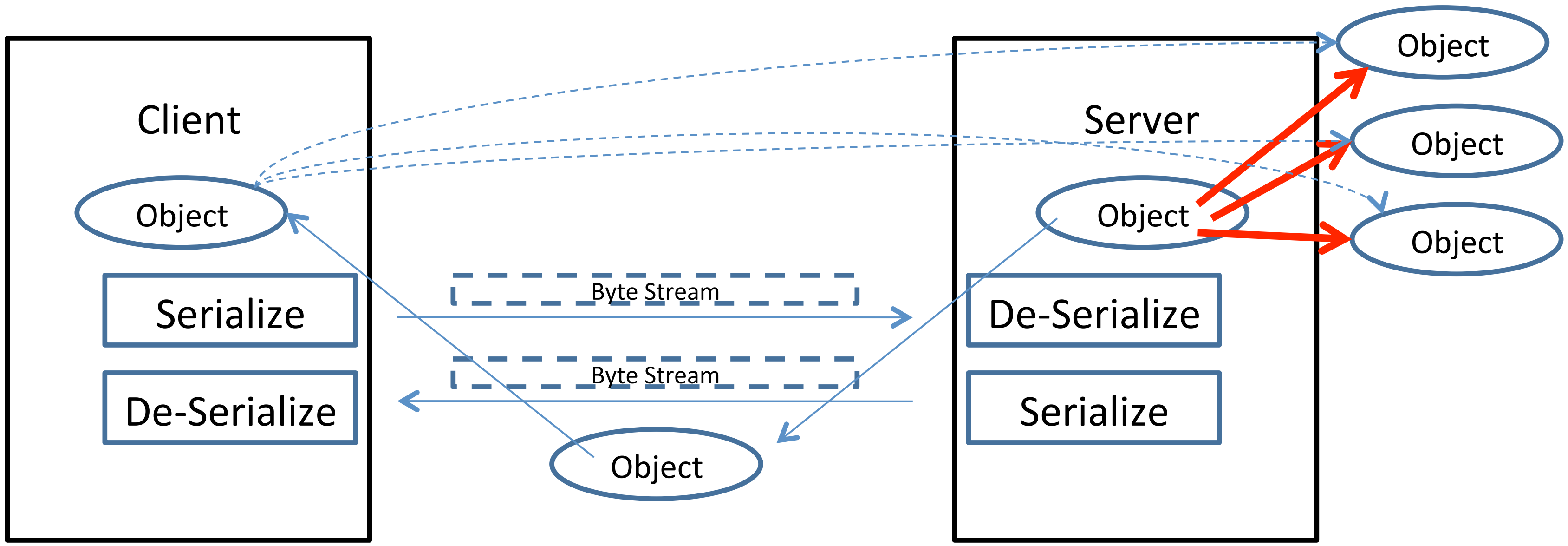
- Parameter passing “by value”:
  - A *copy (the “value”)* of a remote object is transferred between client and server
- As we have seen, any Serializable Java object can be passed *by copy* using RMI.
- It enables:
  - Easy passing of complex-structured information
  - After passing an object to a remote server/client, they can work with a local copy of passed objects
- Please note: manipulating a copy of an object does not affect the original one
- Passing *by copy* leads to object replication and caching (a copy exists both at the client and server side), therefore manipulations on one copy (by the client or server) does not affect the other copy

Parameter Passing “by Reference”

- This is passing *remote object references* as parameters
  - Instead of passing a copy, a reference to a remote object is passed
- Please note: if a reference to an object is passed on to a server / client, any manipulation on the object by one party can be seen by all other parties – remote object becomes “shared”, danger of race conditions
- Example use:
  - a remote object can serve as a “broker” for other remote objects – it can contain references to other remote objects, which can be accessed by those who have a reference to the “broker”
- See, e.g.: rmiregistry – it is a remote object that operates as a “broker” for other remote objects.

Parameter Passing "by Reference"

- This is passing *remote object references* as parameters
  - Instead of passing a copy, a reference to a remote object is passed
- Both client and server see the **same** object in memory



Example: Parameter Passing by Reference

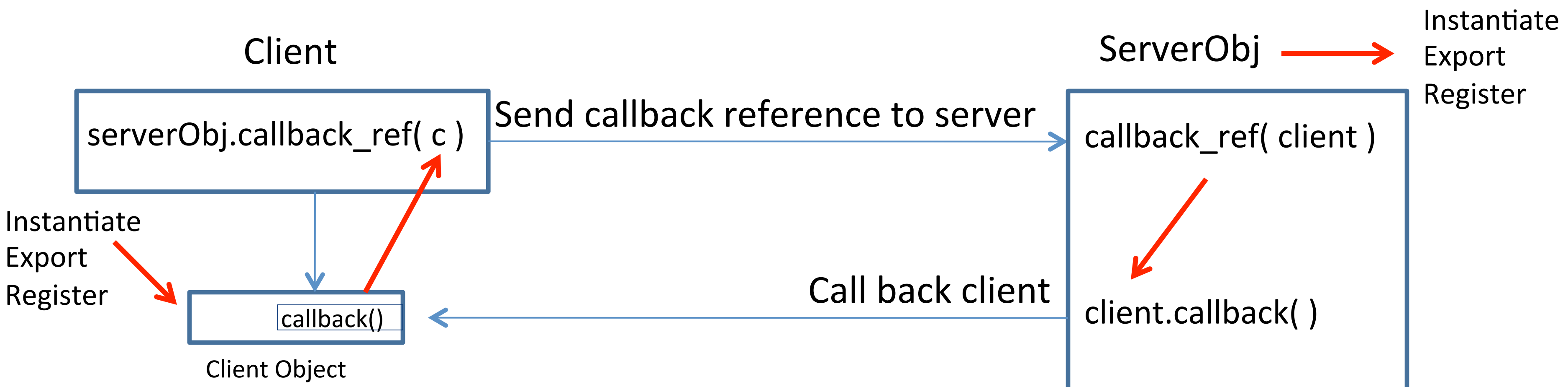
```
RemoteBank    bank    = Naming.lookup ( " … " ) ;
RemoteAccount account = bank.getAccount ( "87654321") ;
Balance       bal     = account.getBalance() ;
```

- With this model, we use the RMI Registry only to bootstrap our distributed system:
  - A reference to the bank is obtained from the Registry
  - Any further object reference can then be obtained directly from the bank server; e.g. an object reference to a RemoteAccount object
  - This may subsequently be used to obtain a *copy* of a Serializable Balance object

Callbacks

The server executes Client code

Callback



- How can the server call the client?
  - Instead of clients frequently calling remote methods() on a server to find out whether some event has occurred (this is called "polling"), the server should inform clients about such events – this is called "callback"
  - This time, the **server** invokes a remote method on the client
- Please note: not only servers, but also clients can register and export remote objects!

Callback

- Implementation in RMI:

  Client:
  - We define a remote interface for a remote client object that can be called remotely by a server and an implementation for it - these are the definitions for a client “callback object”
  - Client instantiates, exports and registers a remote object that implements this interface – this is the “callback object”

  Server:
  - Server provides a method in its remote interface that clients can use to inform the server of the remote object reference of their callback object – the server will record this reference
  - Whenever an event of interest for a client occurs, the server “calls the interested client” – it will call a remote method on the callback object

Callbacks

- Advantage
  - Usually, when a client object invokes a remote method, the client (the calling thread) waits (is blocked) until the remote method returns
  - When server-side computation are lengthy, the client would be blocked during that time – it can be more convenient for the client to pass a remote object reference to the server for call-back
  - The server will invoke a method on the client's call-back object to pass the results of a computation
- Call-backs can also be used to implement extended messaging conversations between client and server

RMI Activation Service

Generate Remote Objects on Demand

Making Distributed Systems Robust and Scalable

Robustness in Distributed Systems

- How can we construct distributed systems that are
  - Long-lived
  - Resilient against system failure
- The distributed system should be able to handle and recover from disturbances and system failure, without loss of information
  - What if a remote server crashes due to a system failure?
  - Ideally, we would like the remote object to be regenerated when it is next required
  - And, its remote reference should be persistent

Scalability in Distributed Systems

- Let’s consider one aspect of scalability
  - Suppose we have a server providing access to an important data resource
  - We could have a data access object running on a server 24/7 (and we guarantee this 100% uptime!)
  - What if the access to the database is only required infrequently? Can we start a data access object “on demand”?

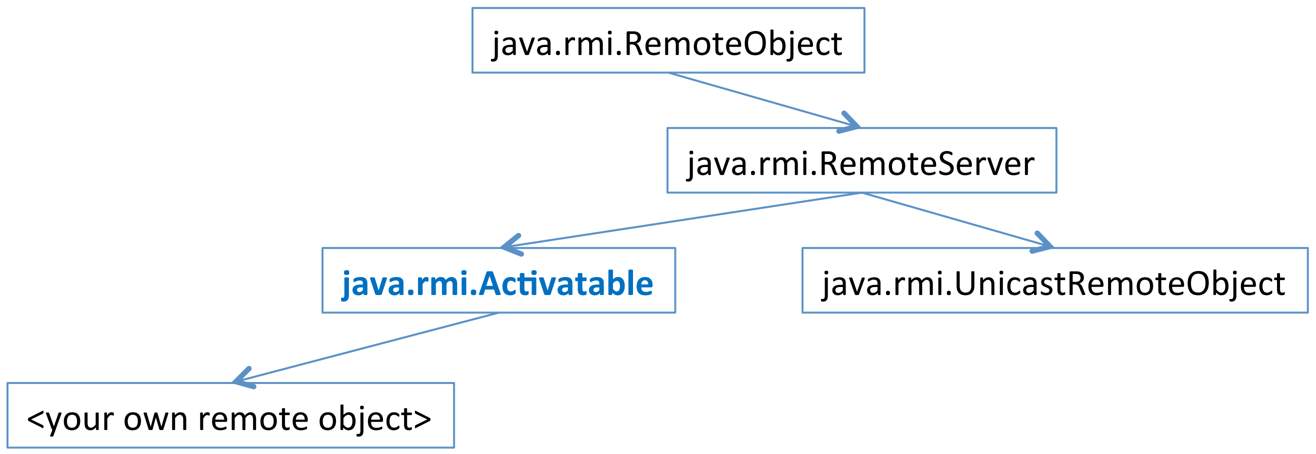
Robustness and Scalability

- The RMI Activation Service can be used to support Robustness and Scalability in the design of distributed systems
- Robustness:
  - The distributed system is constructed in a way to handle and recover from disturbances and system failure, without loss of information, e.g. by
    - Restarting remote objects that were destroyed due to a system failure
- Scalability
  - A distributed system should be able to absorb high system load, increased/decreased service requests , e.g. by
    - Starting services/remote objects on demand

RMI Activation Service

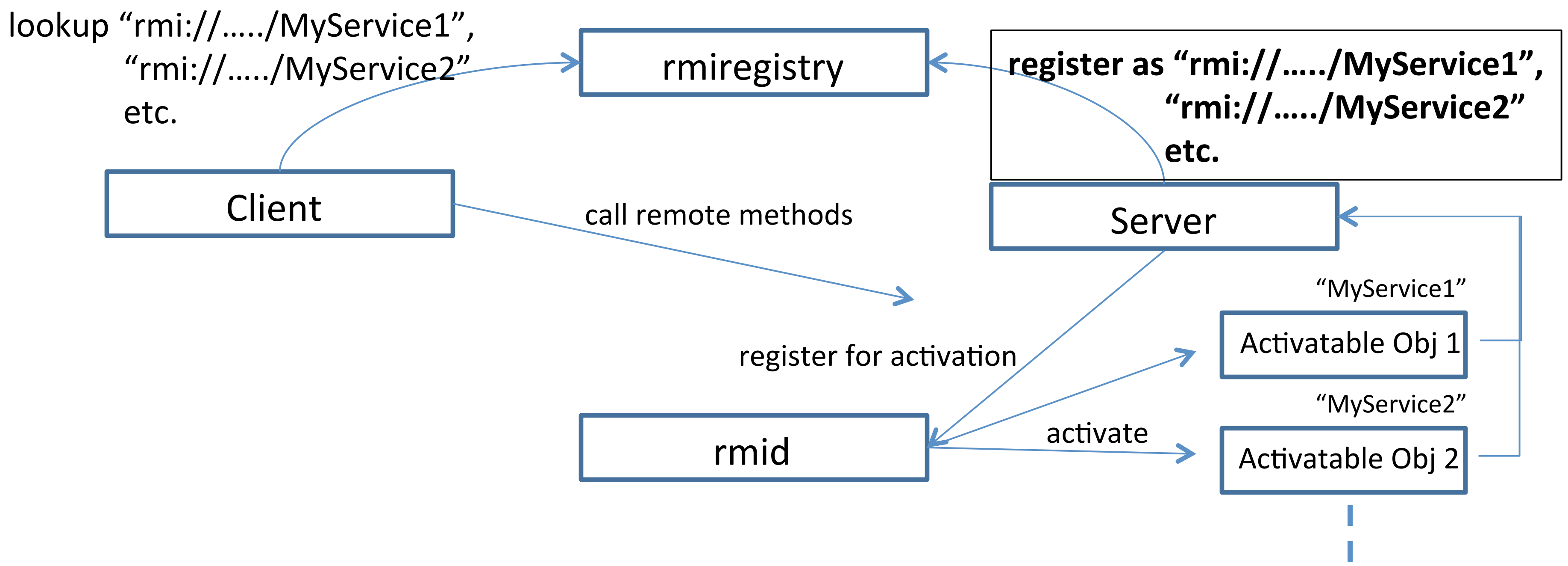
- The key features of the RMI activation service include:
  - The ability to automatically generate remote objects, triggered by requests for references to these objects
  - The ability to restart remote objects if they exist or are destroyed due to a system failure
- So far
  - We used UnicastRemoteObject for exporting Remote objects – these objects live as long as a reference to them exists locally (due to instantiation in the server mainline)
  - We want to our remote objects to be started ("activatable") "on demand"

Creating Activatable Remote Objects



- To create an activatable remote object:
  - The remote object implementation should be a subclass of (extend) **`java.rmi.activation.Activatable`**
  - Activation constructors should be specified in the server implementation
  - The remote object and its activation method must be registered with the activation service (the rmid RMI System daemon)
  - If you want clients to directly access the activatable objects, you need to register the object with the naming service (rmiregistry)

The rmid Activation Service



- In order to provide means for activating remote objects on demand, RMI provides rmid, the RMI Activation System Daemon
- It can be started with the following command:
  - rmid -J-Djava.security.policy=<some policy file> -port <some port number>

Persistent Remote References

- A reference to an activatable remote object does not have to have a “live” object behind it:
  - The object may not have been created yet
  - It may have been garbage collected by its own Java VM
  - Its Java VM may have exited
- The reference remains valid no matter how many times the remote object goes down and is reactivated – it is “persistent”

Client Invocation

- Clients obtain a remote reference in the normal way:
  - Through a naming service (rmiregistry), or
  - From some other (remote) object
- Clients obtain the class specification in the normal way
  - From the local CLASSPATH, or
  - Via HTTP
- When the client invokes the remote object:
  - The activation service (rmid) notices that the remote object is not running and starts it
  - If there is no VM for the remote object, the activation service creates a VM
- As long as the remote object is running, subsequent requests are handled as usual