Script generated by TTT

Title: Distributed_Applications (22.05.2012)

Date: Tue May 22 14:30:20 CEST 2012

Duration: 87:39 min

Pages: 32

Developing RMI applications

The steps developing an RMI application differs slightly from the development steps of a traditional RPC application.

1. Defining a remote interface
2. Implementing the remote interface
3. Generating stubs and skeletons
4. Remote object registration
5. Client implementation

At the end the client must be started.

A remote interface is the set of methods that can be invoked remotely by a client.

- The remote interface must be declared public.
- The remote interface must extend the `java.rmi.Remote` interface.
- Each method must throw the `java.rmi.RemoteException` exception.
- If the remote methods have any remote objects as parameters or return types, they must be interfaces rather than implementation classes.
- **Example: remote interface definition**

```
public interface HelloInterface extends java.rmi.Remote {
    /* this method is called by remote clients and it is implemented by
    the remote object */
    public String sayHello ( ) throws java.rmi.RemoteException
}
```


Definition of an implementation class that defines the methods of the remote interface;

the abstract class `java.rmi.server.RemoteServer` provides the basic semantics to support remote references.

`java.rmi.server.RemoteServer` has subclasses

`java.rmi.server.UnicastRemoteObject` : defines a non-replicated remote object whose references are valid only while the server process is alive.

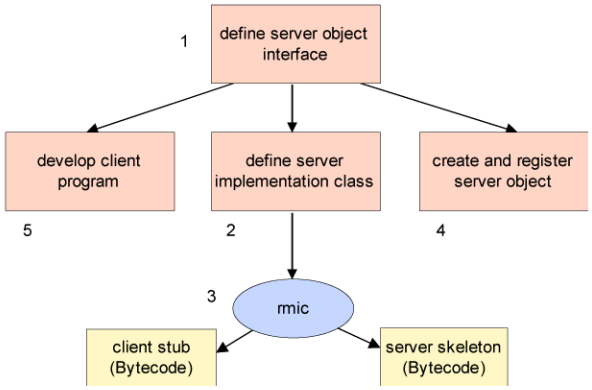
`java.rmi.activation.Activatable` : defines a remote object which can be instantiated on demand (if it has not been started already).

- **Example: Remote interface implementation**

```
import java.io.*;
import java.rmi.* ;
import java.rmi.server.* ;
import java.util.Date.* ;
public class HelloServer extends UnicastRemoteObject implements
HelloInterface{
    public HelloServer( ) throws RemoteException {
       super( );
       /* call superclass constructor to export this object */
    }
    public String sayHello( ) throws RemoteException {
       return "Hello World, the current system time is " + new Date( );
```

The tool rmic generates stub and skeleton from the implemented class (up to Java version 5).

```
rmic HelloServer
```



4 Remote object registration

Every remotely accessible object must be registered in a registry in order to make it available;

stubs are needed for registration.

the registry is started at the host of the remote object.

- **Example for object registration**

```
import java.rmi.* ;

public class RegisterIt {

    public static void main (String args [])
        try { // Instantiate the object
            HelloServer obj = new HelloServer( );
            System.out.println ("Object instantiated: " + obj);
            Naming.rebind("/HelloServer", obj);
            System.out.println("HelloServer bound in registry");
        } catch (Exception e) {
            System.out.println(e)
        }
    }

}
```


5 Client implementation

This step encompasses the writing of the client that uses remote objects.

The client must incorporate a registry lookup in order to obtain a reference to the remote object.

The client interacts with the remote interface, ***never*** with the object implementation.

- **Example: Client implementation**

```
import java.rmi.*;
public class HelloClient {
    public static void main (String args [ ]) {
        if (System.getSecurityManager( ) == null)
        System.setSecurityManager (new RMISecurityManager( ) );
        try { String name = "//" + args [0] + "/HelloServer";
            HelloInterface obj = (HelloInterface) Naming.lookup (name);
            String message = obj.sayHello( );
            System.out.println(message);

        } catch (Exception e) {
            System.out.println("HelloClient exception: " + e);
        }
    }
}
```

Missing access rights results in the exception:

Parameters with primitive data types are passed with their values between JVMs; for object parameters, a distinction is made between local and remote:

1. local object parameter

   RMI passes the object itself, rather than the object reference.

   The transmitted object must implement the interface `java.io.Serializable` or `java.io.Externalizable`.

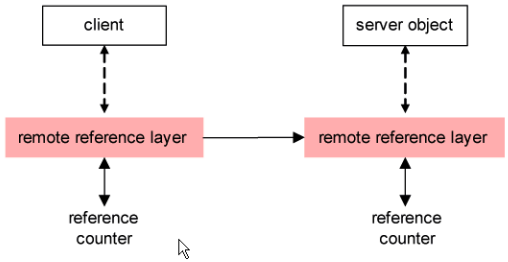
   Classes requiring special handling must implement

   ```
   private void writeObject(java.io.ObjectOutputStreamout) throws
   IOException;

   private void readObject(java.io.ObjectInputStreamin) throws
   IOException, ClassNotFoundException;
   ```

2. remote object parameter

   RMI transmits the stub of the remote object; the stub is a reference to the remote object.

Utilization of life references for each JVM; reference counter represents the number of life references.



The first client access creates a referenced message sent to the server.

If there is no valid client reference, then an unreferenced message is sent to the server.

Time limit of references ("lease time", e.g. 10 minutes); the connection to the server must be renewed by the client, otherwise the reference becomes invalid.

Remote Method Invocation (RMI)

RMI supports communication among objects residing on different Java virtual machines (JVM). RMI is an RPC of the object-oriented Java environment.

Definitions

RMI characteristics

RMI architecture

Locating remote objects

Developing RMI applications

Parameter Passing in RMI

Distributed garbage collection

Servlet Properties

execution of a servlet in the context provided by the servlet engine.

Apache Tomcat : free, open-source implementation of Java servlet technology.

methods specified within each servlet object and invoked by the servlet engine

init: when a servlet is initialized.

shutdown: when a servlet is no longer needed.

service: when a client request is forwarded to the servlet.

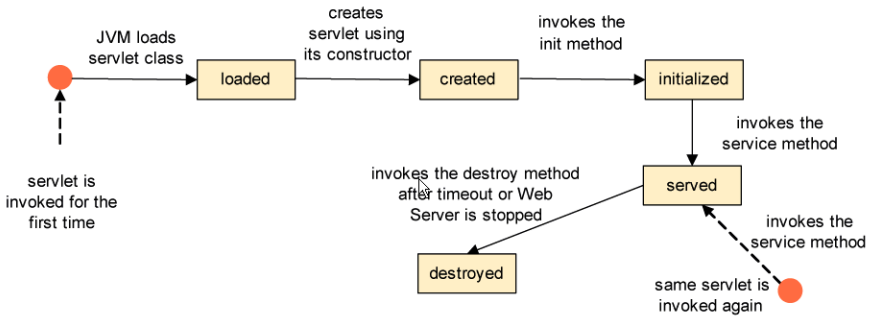
servlets are invoked via HTTP requests (get or post method), e.g.

```
<form method="post"
action="http://myhost:8080/servlet/formServlet">
.... arguments of the form .....
```


Servlet Lifecycle

Interface `javax.servlet.Servlet` specifies the methods to be implemented by the servlets.

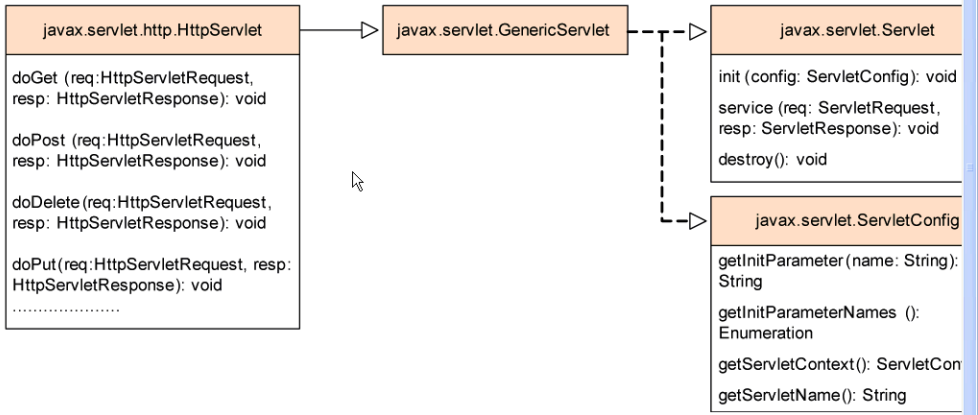
```
public void init() throws ServletException;

public void service(ServletRequest request, ServletResponse response)
throws ServletException, IOException;

public void destroy();
```



HttpServlet inherits abstract class GenericServlet which implements interfaces Servlet and ServletConfig.

GenericServlet defines a generic protocol-independent servlet

HttpServlet defines a servlet for the HTTP protocol

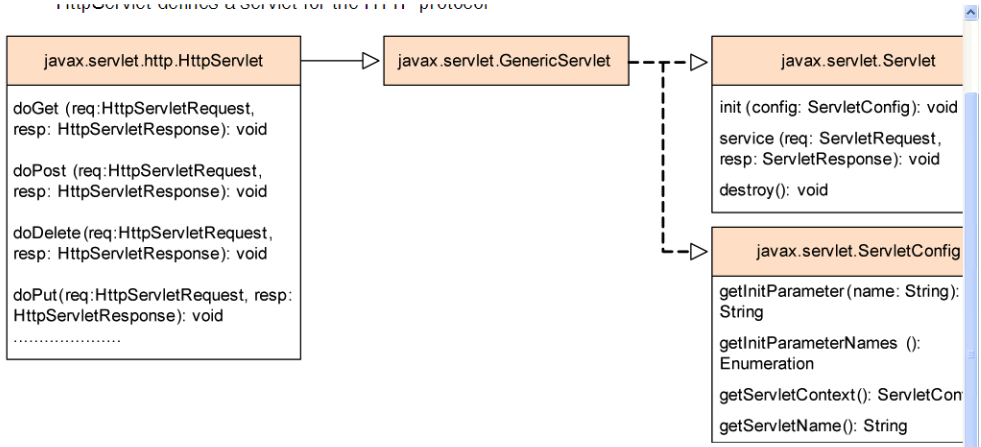


doGet is invoked to respond to a GET request

doPost is invoked to respond to a POST request

doDelete is invoked to respond to a DELETE request; normally used to delete a file on the server

HttpServlet Interface



doGet is invoked to respond to a GET request

doPost is invoked to respond to a POST request

doDelete is invoked to respond to a DELETE request; normally used to delete a file on the server

doPut is invoked to respond to a PUT request; normally used to send a file to the server

HttpServlet class provides a default implementation which must be overriden to process the requests.


```
import javax.servlet.*;
import javax.servlet.http.*
import java.io.*;

public class MyServlet extends HttpServlet {
    /** called by the servlet engine to initialize servlet */
    public void init() throws ServletException { .... }
    /** process the HTTP Get request */
    public void doGet(HttpServletRequest request, HttpServletResponse
    response) throws ServletException, IOEXception { .... }
    /** process the HTTP Post request */
    public void doPost(HttpServletRequest request, HttpServletResponse
    response) throws ServletException, IOEXception { .... }
    /** called by the servlet engine to release the resource */
    public void destroy () { .... }
    // other methods
}
```

[Example - CurrentTime]

[Example - Registration of Students]


```
import javax.servlet.*;
import javax.servlet.http.*
import java.io.*;

public class CurrentTime extends HttpServlet {
    /** process the HTTP Get request */
    public void doGet(HttpServletRequest request, HttpServletResponse
    response) throws ServletException, IOEXception {
        response.setContentType('text/html');
        PrintWriter out = response.getWriter();
        out.println("<p>The current time is " + new java.util.Date() );
        out.close(); // close stream
    }
}
```

Invocation

http://localhost:8080/.../servlet/CurrentTime

This HTML form and the associated servlet process student registrations.

HTML Form

Servlet

Generated by Targeteam

```
<html>
<head>
<title>Student Registration Form</title>
</head>
<body>
<p>Student Registration Form
<form action="http://localhost:8080/examples/servlet/GetParameters"
method="GET">
    Last Name <input type="text" name="lastName" size="20">
    First Name <input type="text" name="firstName" size="20">
    <p>Gender:
    <input type="radio" name="gender" value="M" checked>Male
    <input type="radio" name="gender" value="F">Female</p>
    <p>Major <select name="major" size="1">
    <option value="CS">Computer Science
    <option value="MA">Mathematics
    </select>
    Minor <select name="minor" size="2" multiple>
    <option>Computer Science
    <option>Mathematics
    <option>Economics
```

Servlet

```
import javax.servlet.*;
import javax.servlet.http.*
import java.io.*;

public class GetParameters extends HttpServlet {
    /** process the HTTP Get request */
    public void doGet(HttpServletRequest request, HttpServletResponse
    response) throws ServletException, IOException {
       response.setContentType('text/html');
       //obtain parameters from the client
       String lastName = request.getParameter("lastName");
       String firstName = request.getParameter("firstName");
       String gender = request.getParameter("gender");
       String major = request.getParameter("major");
       String[] minors = request.getParameterValues("minor");
       String tennis = request.getParameter("tennis");
       String soccer = request.getParameter("soccer");
       String golf = request.getParameter("golf");
       String remarks = request.getParameter("remarks");
       out.println("Last Name: <b>" + lastName + "</b> First Name: <b>"
          + firstName + "</b><br>");
       out.println("Gender: <b>" + gender + "</b><br>");
```

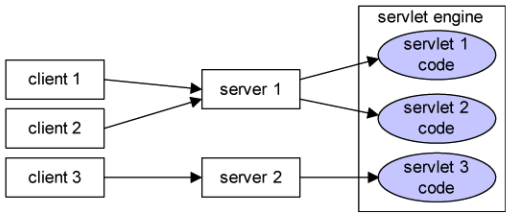
Servlet

```
       String lastName = request.getParameter("lastName");
       String firstName = request.getParameter("firstName");
       String gender = request.getParameter("gender");
       String major = request.getParameter("major");
       String[] minors = request.getParameterValues("minor");
       String tennis = request.getParameter("tennis");
       String soccer = request.getParameter("soccer");
       String golf = request.getParameter("golf");
       String remarks = request.getParameter("remarks");
       out.println("Last Name: <b>" + lastName + "</b> First Name: <b>"
          + firstName + "</b><br>");
       out.println("Gender: <b>" + gender + "</b><br>");
       out.println("Major: <b>" + major + "</b> Minor: <b>");
       if (minors != null)
          for (int i = 0; i < minors.length; i++)
             out.println(minors[i] + " ");
       out.println("</b><br> Tennis: <b>" + tennis + "</b> Soccer: <b>"
          + soccer + "</b> Golf: <b>" + golf + "</b><br>");
       out.println("Remarks: <b>" + remarks + "</b>");
       out.close(); // close stream
    }
```

Servlets

Servlets (Java Servlets) are programs invoked by a client and executed on the server host:

used to extend the functionality of the server.



Servlet Properties

Servlet Lifecycle

HttpServlet Interface

Structure of a Servlet

Issues

The following section discusses several important basic issues of distributed applications.

Data representation in heterogeneous environments.

Discussion of an execution model for distributed applications.

What is the appropriate error handling?

What are the characteristics of distributed transactions?

What are the basic aspects of group communication (e.g. algorithms used by ISIS) ?

How are messages propagated and delivered within a process group in order to maintain a consistent state?

External data representation

Time

Distributed execution model

Failure handling in distributed applications

Distributed transactions

Group communication

Distributed Consensus

Authentication service Kerberos

External data representation

Heterogeneous environment means different data representations

⇒ requirement to enable data transformation.

independence from hardware characteristics while exchanging messages means: use of external data representation.

Marshalling and unmarshalling

Centralized transformation

Decentralized transformation

Common external data representation

XML as common data representation

External data representation

Heterogeneous environment means different data representations

⇒ requirement to enable data transformation.

independence from hardware characteristics while exchanging messages means: use of external data representation.

Marshalling and unmarshalling

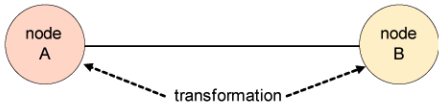
Centralized transformation

Decentralized transformation

Common external data representation

XML as common data representation

Generated by Targeteam



All nodes execute data transformations.

Variants

A transforms data which are then sent to B; B transforms data which are then sent to A.

A transforms data by B; B transforms data by A.

A and B transform data in a network-wide standard format; the respective recipients retransform the received data into the local format.

If new system components are dynamically added to the distributed system, the new system components simply have to "learn" about the network-wide unique standard representation.

No special hardware is required.

Example: XDR as part of ONC by Sun.

Two aspects of a common external data representation are of importance:

a machine-independent format for data representation, and

a language for description of complex data structures.

Examples: XDR ("eXternal Data Representation") by Sun and ASN.1 (Abstract Syntax Notation). Other formats are

Corba's common data representation: structured and primitive types can be passed as arguments and results.

Java's object serialization: flattening of single objects or tree of objects.

Representation of numbers
External representation of strings
External representation of arrays
Transfer of pointers

For the representation of numbers in main memory, one of the following methods are generally used.

"little endian" representation: the lower part of a number is stored in the lower memory area

"big endian" representation: the higher part of a number is stored in the lower memory area, e.g. the Sun-Sparc architecture

Example representation of the number 1347

Memory Address	1000	1001	1002	1003
Big Endian	00000000	00000000	00000101	01000011
Little Endian	01000011	00000101	00000000	00000000

Convention: for network transfer, numbers which encompass several bytes are structured according to a well-defined representation, such as "big endian".

Common external data representation

Two aspects of a common external data representation are of importance:

a machine-independent format for data representation, and

a language for description of complex data structures.

Examples: XDR ("eXternal Data Representation") by Sun and ASN.1 (Abstract Syntax Notation). Other formats are

Corba's common data representation: structured and primitive types can be passed as arguments and results.

Java's object serialization: flattening of single objects or tree of objects.

Representation of numbers
External representation of strings
External representation of arrays
Transfer of pointers

Generated by Targeteam

no shared address space for client and server: pointer transfer is problematic.

1. prohibit pointers in a remote procedure call.
2. dereference pointers in a remote procedure call.

   serialize the data structure the pointer is pointing to ("marshal"), and transfer the whole data structure.

   - no use of null pointers; instead, we use boolean variables.
   - in heterogeneous environments no use of function pointers.

     However, in a homogenous Java environment function pointers can be dereferenced and the function transferred to the server site.
3. pointer transfer.

External data representation

Heterogeneous environment means different data representations

⇒ requirement to enable data transformation.

independence from hardware characteristics while exchanging messages means: use of external data representation.

Marshalling and unmarshalling

Centralized transformation

Decentralized transformation

Common external data representation

XML as common data representation

Request for the invocation of the Java method

```
echoString("cat")
```

SOAP body of request that sends a string.

```
<soap:Body>
   <n:echoString
   xmlns:n='http://tempuri.org/mapping.server.Primitive'>
   <value xsi:type='xsd:string'>cat</value>
   </n:echoString>
</soap:Body>
```


Request for Java method invocation

```
echoInts([1, 2, 3])
```

SOAP body of request that sends an array.

```
<soap:Body>
   <n:echoInts
   xmlns:n='http://tempuri.org/mapping.server.Array'>
   <ints href='#id0'>
   </n:echoInts>
   <id0 id='id0'
      soapenc:root='0'
      xsi:type='soapenc:Array'
      soapenc:ArrayType='xsd:int[3]'>
      <i xsi:type='xsd:int'>1</i>
      <i xsi:type='xsd:int'>2</i>
      <i xsi:type='xsd:int'>3</i>
   </id0>
</soap:Body>
```

XML as common data representation

Complex data types can be mapped to XML for transmission across the network.

primitive datatypes → XSD equivalent

boolean, byte, unsignedShort (used for char), int, long, float, string, ...

Example: primitive datatypes

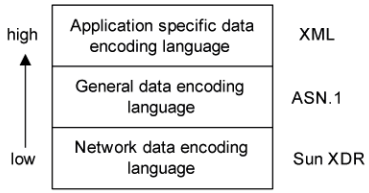
SOAP provides built-in support for encoding arrays.

Example: array datatype

complex data types are mapped to XML schema types;

SOAP platforms provide API for creating custom mapping.

e.g. writeSchema to specify an XML schema definition



Level of Abstraction