JAVA

RMI
Overview

- **Remote Method Invocation**
- usage of remote object
  - objects in a different VM (on the same computer or over the network)
- as there would be local objects (almost)
  - calls just take longer time
Remote call in general

client

\texttt{vo.method();}

\texttt{method() \{ ... \}}

\texttt{stub}

\texttt{method() \{ ... \}}

\texttt{skeleton}

\texttt{method() \{ ... \}}

RMI

transport layer (TCP/IP)

remote object

\texttt{method() \{ ... \}}
Example: interface

1. the interface for a remote object
   - must extend java.rmi.Remote
   - java.rmi.RemoteException declared by each methods

```java
import java.rmi.Remote;
import java.rmi.RemoteException;

public interface Hello extends Remote {
    String sayHello() throws RemoteException;
}
```
Example: implementation

2. implementation of the interface

```java
public class HelloImpl extends UnicastRemoteObject implements Hello {

    public HelloImpl() throws RemoteException {
    
    
    public String sayHello() throws RemoteException{
        return "Hello, world!";
    
    }

    }
```
Example: creating the object

3. create the object
4. register the object

```java
public class HelloImpl implements Hello
        extends UnicastRemoteObject {
    ...

    public static void main(String args[]) {
        try {
            HelloImpl obj = new HelloImpl();
            Naming.rebind("Hello", obj);
        } catch (Exception e) {
            ...
        }
    }
```
Example: client

```java
public class HelloClient {

    public static void main(String[] args) {
        try {
            Hello robj = (Hello) Naming.lookup("Hello");
            String mesg = robj.sayHello();
            System.out.println(mesg);
        } catch (Exception e) {
            ....
        }
    }

    5. obtaining a reference to the remote object
    6. using the object
```
Example: compilation and run

7. compilation
   - as usually

8. launching
   a) rmiregistry
   b) java -Djava.rmi.server.codebase=file:/***/ HelloImpl
      • codebase ~ a path to the class files
   c) java HelloClient
Example: object implementation

- different way to implement an object
  - if UnicastRemoteObject cannot be extended

```java
public class HelloImpl implements Hello {
    ...
    public static void main(String args[]) {
        try {
            HelloImpl obj = new HelloImpl();
            Hello robj = (Hello)
                UnicastRemoteObject.exportObject(obj, 0);
            Naming.rebind("Hello", robj);
        } catch (Exception e) {
            ...
        }
    }
}
```
Stubs & skeletons

- generated automatically
- JDK 1.4
  - automatically skeletons only
  - stubs generated “by-hand”
  - rmic compiler
    - executed after javac to Remote objects implementations
  - codebase must be set for the server
    - -Djava.rmi.server.codebase=......
    - codebase point to the stubs
    - a client automatically downloads them from codebase
    - codebase is typically file:, ftp://, http://
    - must end with /
  - it is necessary to set the security policy
    - -Djava.security.policy=....
  - the security manager must be set
    - System.setSecurityManager(new SecurityManager());
Stubs & skeletons

- JDK 1.4 (cont.)
  - `rmiregistry` must not have set CLASSPATH, in which are classes to be downloaded
- JDK 1.5
  - if stubs are available => they are not generated
  - for always generated stubs
    - set the property `java.rmi.server.ignoreStubClasses` to `true`
- JDK 1.1
  - nothing is generated
  - `rmic` generates both stubs and skeletons
Stubs & skeletons & codebase

- WARNING
  - since JDK 7 Update 21 change of behavior
  - the property `java.rmi.server.useCodebaseOnly` set to `true` by default
    - previously it was `false`
  - if it is set to true, automatic loading of classes is allowed only from locally set codebase
    - i.e. codebase has to be set also for the rmiregistry or the rmiregistry has to have the `useCodebaseOnly` set to `false`
Code downloading

1. registrace remote objektu
2. lookup
3. reference na remote objekt
4. požadavek na stub
5. stub

java.rmi.server.codebase = http://host/path/

host

RMI klient

RMI registry

RMI server

http server
Distributed Object Model

- **no differences** from the plain Java Object Modelu
  - references to remote objects can be passed method parameters
  - remote objects can be cast to a remote interface
  - it is possible to use `instanceof` for remote interface tests

- **differences** from the plain Java Object Modelu
  - clients always work with a remote object via remote interface
    - i.e. no direct access to object fields
  - non-remote parameters passed by-value
  - several methods from `java.lang.Object` are overridden
    - `hashCode`, `equals`
  - methods throw `RemoteException`
Class hierarchy

<< interface >>
Remote

RemoteObject
+ hashCode(): int
+ equals(): boolean
+ toString(): String

RemoteStub
+ setRef(): void

RemoteServer
+ getClientHost(): String
+ getLog(): PrintStream
+ setLog(): void

IOException

IOException

UnicastRemoteObject
+ exportObject(): Remote
+ clone(): Object

RemoteException
Threads

- no guarantee how calls on a remote object are associated with threads
- calls on the same remote object can be executed concurrently
Naming

- obtaining an initial reference to remote object
- simple directory service
  - references to objects associated with strings
- implementation – `rmiregistry`
  - a reference in the registry is either
    - till its explicit removal, or
    - till `rmiregistry` terminating
  - a reference to an object can be in the registry even the object has been already terminated
- accessible also via RMI
- how to obtain a reference to `rmiregistry`?
  - (the chicken-egg problem)
  - the reference to `rmiregistry` is created from the address and port of the computer, where `rmiregistry` runs
Naming

- several rmiregistries can be used at the same moment
  - on different computers
  - on different ports
- to rmiregistry, only processes running on the same computer can register objects
  - reading from the registry from everywhere

  - work-around
    - create a remote object running on the same computer as the registry; the object will register objects running elsewhere
the program **rmiregistry**
- one parameter – port
  - default 1099
- typical usage
  - unix
    - `rmiregistry &`
  - Win
    - `start rmiregistry`
Naming: access

- `java.rmi.Naming`
- only static methods
  - `bind`, `rebind`, `unbind`
  - `lookup`
  - `list`
- first parameters is String – defines the name of an object and possibly the registry
  - `//host:port/jmeno`
    - `host` and `port` are optional
    - default – `localhost` and `1099`
Naming: access

- the package `java.rmi.registry`
  - the class LocateRegistry
    - obtaining a reference to the registry
    - creating the registry
  - the interface Registry
    - the same methods as the `Naming` class
      - first parameter specifies only the name of an object

- JNDI – Java Naming and Directory Interface
  - unified access to different directory services
  - support also trading (yellow pages)
Own sockets

- own sockets can be used for RMI
- create own socket factory
  - client socket factory
    - implements RMIClientSocketFactory and Serializable
  - server socket factory
    - implements RMIServerSocketFactory
- factories are specified during a remote object creation
- typical usage – encryption
  - javax.rmi.ssl
    - SSLRMIServerSocketFactory
    - SSLRMIClientSocketFactory
Activation

• an object activated when it is necessary
• rmid
  − activation daemon
  − a "database" of activation records

• objects
  − extends the class java.rmi.activation.Activatable
    • or export an object using the class
      − plus – registering the activation record to rmid
  • during registration of the activation record, an explicit specification of permissions is necessary
    − AllPermissions is not enough
Activation

```java
public interface MyRemoteInterface extends Remote {
    ...
}

public class MyRemoteImpl extends Activatable
    implements MyRemoteInterface {
    public MyRemoteImpl(ActivationID id, MarshaledObject m) throws RemoteException {
        super(id, 0);
    }
    ....
}

Or

public class MyRemoteImpl implements MyRemoteInterface {
    public MyRemoteImpl(ActivationID id, MarshaledObject m) throws RemoteException {
        Activatable.exportObject(this, id, 0);
    }
    ....
}
```
Activation

• registration
  – create registration record
    • public ActivationDesc(ActivationGroupID groupID, String className, String location, MarshalledObject data)
  – register it
    • static Remote Activatable.register(ActivationDesc desc)
      – returns a stub
  – register the stub in rmiregistry
    • as usually
Distributed garbage collector

- garbage collector in distributed environment
- reference counting
- "leases"
- an object can be collected if there is no reference or lease has expired
- VMID – an identifier of VM
  - unique
  - lease contains it
RMI-IIOP

• transport protocol – JRMP
  – Java Remote Message Protocol
• IIOP can be used
  – CORBA interoperability
    • CORBA client – RMI server

• usage
  – the javax.rmi package
  – extend PortableRemoteObject
    • no UnicastRemoteObject
  – use rmic with the parameter -iiop
  – use the CORBA naming
    • javax.naming.... (JNDI)
    • instead of rmiregistry, use orbd
Security
(Access control)
Overview

- originally in Java – a “sandbox” model
- later, other services added
  - next – managing access to resources
Security Manager

- java.lang.SecurityManager
  - before a resource is accessed, it checks, whether there are necessary permissions
  - not set by default
    - for “regular” applications
      - for JNLP executed application, it is set by default
- permissions – java.security.Permission
  - during class loading, the classloader assigns permissions to classes
- java.security.Policy
  - a set of permissions
  - only one in VM
  - typically, it is set via a text file
Security Manager

- setting SM
  - either in code
    - System.setSecurityManager(sm)
  - or from command-line
    - -Djava.security.manager
      - sets default sm
    - -Djava.security.manager=org.foo.SM

- default SM
  - implemented via java.security.AccessController
  - tests all “elements” in the call stack

- own one can be implemented
Policy

• setting a Policy
  -Djava.security.policy=file.policy

• formát

```
grant [SignedBy "signer_names"] [, CodeBase "URL"]
    [, Principal [principal_class_name] "principal_name"]
    [, Principal [principal_class_name] "principal_name"] ...
    { permission permission_class_name [ "target_name" ]
      [, "action"] [, SignedBy "signer_names"];
      permission ... }
```

– příklad

```
grant codeBase "file:/home/sysadmin/" {
      permission java.io.FilePermission "/tmp/abc", "read";
};
```