Java, summer semester 2020

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

- `java.rmi` module
Remote call in general

client

vo.method();

stub
method() { ... }

remote object

method() { ... }

RMI

transport layer (TCP/IP)
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;
}
```
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) {
            ...
        }
    }
}
```
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
• 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
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`
1. registrace remote objektu

2. lookup

3. reference na remote objekt

4. požadavek na stub

5. stub

RMI klient

RMI registry

RMI server

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

http server

host
Distributed Object Model

- **no differences** from the plain Java Object Model
  - 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 Model
  - 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
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 – \textit{rmiregistry}
  - a reference in the registry is either
    - till its explicit removal, or
    - till \textit{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 \textit{rmiregistry}?
  - (the chicken-egg problem)
  - the reference to \textit{rmiregistry} is created from the address and port of the computer, where \textit{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)
  - `java.namig` module
    - `javax.naming` package
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
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
  - javax.rmi package (java.corba module)
  - extend PortableRemoteObject
    - no UnicastRemoteObject
  - use rmic with the parameter -iiop
  - use the CORBA naming
    - javax.naming.... (JNDI)
    - instead of rmiregistry, use orbd
Other “RMIs”
gRPC

- https://grpc.io/
- multiplatform
  - Java, Python, C#, C++, …
- interfaces ~ protocol buffers

```java
service Greeter {
  rpc SayHello (HelloRequest) returns (HelloReply) {}  
}
message HelloRequest {  
  string name = 1;  
}
message HelloReply {  
  string message = 1;  
}
```

- protocol – HTTP + WebSockets
Další

• ...

Java, summer semester 2020
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";
};
Java Management Extensions
JMX
Overview

- part of JDK since version 5
  - previously an external set of jar archives
- MBean = Managed Java Bean
  - beans intended for managing something (device, application, anything)
  - provides an interface like std. beans
    - properties (get and set methods)
    - regular methods
    - notifications via events
  - several types
    - standard
    - dynamic
    - open
    - model
- (not only) universal client – JConsole
Architecture

Distributed Services Level

Agent Level

Instrumentation Level

Connectors and Protocol Adaptors

MBean Server

Resource 1 (MBean)

Resource 2 (MBean)

Java virtual machine (host1)

Current JMX specification

Separate JSRs

Future phases of the JMX specification

image source: JMX Specification, version 1.4
Types of MBeans

- **Standard**
  - the simplest type
  - its interface = all methods

- **Dynamic**
  - must implement a particular interface
  - more flexible
  - can be changed at runtime

- **Open**
  - dynamic
  - but can use only basic types
    - no need for a special descriptor

- **Model**
  - dynamic
  - fully configurable at run-time
Standard MBean

• defined explicitly by its interface and implementation (class)
  − the interface must have the same name as the class plus extension MBean
  − all methods in the MBean interface are provided
    • methods of the class but not in the interface are not visible via JMX
  − rules for naming properties and methods are the same as for regular beans
  − the interface is at run-time obtained via reflection
package example.mbeans;

public interface MyClassMBean {
    public int getState();
    public void setState(int s);
    public void reset();
}

package example.mbeans;

public class MyClass
    implements MyClassMBean {
    private int state = 0;
    private String hidden = null;
    public int getState() {
        return(state);
    }
    public void setState(int s) {
        state = s;
    }
    public String getHidden() {
        return(hidden);
    }
    public void setHidden(String h) {
        hidden = h;
    }
    public void reset() {
        state = 0;
        hidden = null;
    }
}
package example.mbeans;

import java.lang.management.*;
import javax.management.*;

public class Main {

    public static void main(String[] args) throws Exception {

        MBeanServer mbs = ManagementFactory.getPlatformMBeanServer();

        ObjectName name = new ObjectName("example.mbeans:type=MyClass");

        MyClass mbean = new MyClass();
        mbs.registerMBean(mbean, name);

        System.out.println("Waiting forever...");
        Thread.sleep(Long.MAX_VALUE);
    }
}
Dynamic MBean

- intended for a changing interface
- implements the `DynamicMBean` interface
  - the bean's interface is obtained at run-time via calling methods of this interface

```java
interface DynamicMBean {
    MBeanInfo getMBeanInfo();
    Object getAttribute(String attribute);
    AttributeList getAttributes(String[] attributes);
    void setAttribute(Attribute attribute);
    AttributeList setAttributes(AttributeList attributes);
    Object invoke(String actionName, Object[] params, String[] signature);
}
```
Dynamic MBean

- MBeanInfo
  - describes the MBean interface
  - for each call, a result of getMBeanInfo can be different
    - then, universal JMX clients cannot be (usually) used
Identification

- the class `ObjectName`
  - represent the name of a mbean or a pattern for searching
  - composed of a domain and properties
  - domain
    - string
    - must not contain colon and //
  - properties
    - name-value pairs
      - type – type of mbean
      - name
      - ...

JMX notification

• MBean can generate events
  – e.g. after change of its state
  – similar to regular beans
• the Notification class
  – represents an event
  – extends java.util.EventObject
  – can be used directly
    • but typically via its children (again as with regular beans)
• the NotificationListener interface
  – registering for event listening
• the NotificationBroadcaster interface
  – MBeans generating events must implement this interface
  – it is better to implement NotificationEmitter
    • extends NotificationBroadcaster
JMX notifikace

- the NotificationFilter interface
  - filtering notifications
  - a listener registers it
- types of event
  - it is not the class
  - a property of the event (String)
  - hierarchical
    - JMX.<something> reserved for JMX
- properties of the event (of the class Notification)
  - type
  - sequence number
  - timestamp (when the event was generated)
  - message
  - user data
JMX notification

- NotificationEmitter
  - `void addNotificationListener(NotificationListener listener, NotificationFilter filter, Object handback)`
  - `handback` - a utility object
  - the emitter does not use it
  - it is passed during event delivery
- `void removeNotificationListener(NotificationListener listener)`
- `void removeNotificationListener(NotificationListener listener, NotificationFilter filter, Object handback)`
- `MBeanNotificationInfo[] getNotificationInfo()`
JMX notification

- NotificationListener
  - void handleNotification(Notification notification, Object handback)

- NotificationFilter
  - boolean isNotificationEnabled(Notification notification)

- support for notifying field changes
  - AttributeChangeNotification
  - AttributeChangeNotificationFilter

- the NotificationBroadcasterSupport class
  - a prepared implementation of NotificationBroadcaster
public class Hello extends NotificationBroadcasterSupport implements HelloMBean {
    ....
    public synchronized void setCacheSize(int size) {
        int oldSize = this.cacheSize;
        this.cacheSize = size;
        Notification n = new AttributeChangeNotification(this,
                sequenceNumber++, System.currentTimeMillis(), "CacheSize changed", "CacheSize", "int", oldSize, this.cacheSize);
        sendNotification(n);
    }

    public MBeanNotificationInfo[] getNotificationInfo() {
        String[] types = new String[] {
            AttributeChangeNotification.ATTRIBUTE_CHANGE
        };
        String name = AttributeChangeNotification.class.getName();
        String description = "An attribute of this MBean has changed";
        MBeanNotificationInfo info = new MBeanNotificationInfo(types, name, description);
        return new MBeanNotificationInfo[] {info};
    }
}
Open MBean

• dynamic MBean
• uses only a limited set of data types
  – basic data types
    • primitive types (wrapper types)
    • String
    • BigDecimal, BigInteger
    • Date
    • javax.management.openbean.CompositeData
    • javax.management.openbean.CompositeTabular
    • arrays of these types
• can be used with universal clients
  – no need to recompile clients after the interface change
Open MBean

- `javax.management.openbean.CompositeData`
  - interface
  - represents composed types
  - “structures”
  - similar to a hash table
- `javax.management.openbean.CompositeTabular`
  - interface
  - represents arrays

- OpenMBeanInfo
  - extends MBeanInfo
  - plus other “Open” descriptors
    - OpenMBeanOperationInfo,...
Model MBean

- dynamic
- generic and fully configurable at run-time
  - no static interface, but elements are dynamically added
Model MBean example

MBeanServer mbs = ...

HashMap map = new HashMap();

Method getMethod = HashMap.class.getMethod("get", new Class[]{Object.class});
ModelMBeanOperationInfo getInfo =
    new ModelMBeanOperationInfo("Get value for key", getMethod);
ModelMBeanInfo mmbi =
    new ModelMBeanInfoSupport(HashMap.class.getName(),
    "Map of keys and values",
    null,   // no attributes
    null,   // no constructors
    new ModelMBeanOperationInfo[]{getInfo},
    null);   // no notifications

ModelMBean mmb = new RequiredModelMBean(mmbi);
mmb.setManagedResource(map, "ObjectReference");

ObjectName mapName = new ObjectName(":type=Map,name=whatever");
mbs.registerMBean(mmb, mapName);

mbs.invoke(mapName, "get", new Object[]{"key"}, new String[]{Object.class.getName()});
MXBean

- a new type of MBean
  - since JDK 6 (partially also in 5)
- a standard MBean
- plus rules for Open MBean
  - i.e. uses only a limited set of data types
- MXBean is a class implementing a `<something>`MXBean interface
  - the class can have any name
- instead of the extension MXBean the annotation @MXBean can be used
  - also @MXBean(false) can be used to set that the given interface is not a JMX interface even it has the MXBean extension
Architecture (recap.)

- JMX-compliant Management Application
- Web Browser
- Proprietary Management Application
- Distributed Services Level
- Agent Level
- Instrumentation Level
- Connectors and Protocol Adaptors
- JMX Manager
- MBean Server
- Resource 1 (MBean)
- Resource 2 (MBean)
- Java virtual machine (host1)

Current JMX specification
Separate JSRs
Future phases of the JMX specification

Additional Management Protocol APIs
- SNMP Manager API
- CIM/WBEM API
- TMN Manager API

Image source: JMX Specification, version 1.4
JMX Remote

- remote access to JMX
- via *connectors*
  - composed of
    - connector client
    - connector server
- connectors can be created over (almost) anything
- the specification defines 2 particular connectors
  - RMI
  - generic
    - JMX Messaging Protocol (JMXMP)
      - directly over TCP
    - its implementation is optional
A connection creation

connect “service:jmx:jmxmp://host1:9876”

1. connection request

2. create server end

3. connection response

4. create client end

image source JMX Specification, version 1.4
JMX Remote

- creating a MBean, registration,... are as previously
- plus creating the connector server

```java
MBeanServer mbs = MBeanServerFactory.createMBeanServer();
...

JMXServiceURL url = new JMXServiceURL("service:jmx:rmi:///jndi/rmi://localhost:9999/server");

JMXConnectorServer cs = JMXConnectorServerFactory.newJMXConnectorServer(url, null, mbs);

cs.start();
...

cs.stop();
```
JMX Remote

- JMXServiceURL
  - url of the connector server
  - depends on the type of a connector
  - common structure
    - service:jmx:<protocol>:
  - for own connectors it is not necessary to follow the structure
    - but it is recommended

- the JMX specification defines
  - message buffering
  - rules for parallel usage
  - how to deal with communication errors
  - dynamic class loading
  - security
  - ...
JMX Remote – RMI connector

- mandatory
  - every JMX implementation must contain it
- uses regular RMI
- usage of JRMP or IIOP can be specified
- using the RMI connector
  - service:jmx:rmi://host:port
    - the connector server creates a RMI server and returns a URL in a form service:jmx:rmi://host:port/stub/XXXX
      - XXXX is the serialized RMI server
  - service:jmx:iiop://host:port
    - the connector server creates a CORBA object and returns a URL in a form
      service:jmx:iiop://host:port/ior/IOR:XXXX
      - XXXX is std. ior
    - creates a server and registers it in the naming service
      - iiop can be written instead of rmi
JMX Remote – Generic connector

- optional
  - JMX implementations need not to contain it
- configurable
  - goal – a simple specification of transport protocols and wrapper objects for communication
- defines communication using messaging
  - a connection initialization
  - messages
  - ...
- JMXMP connector
  - a configuration of the generic connector for JMXMP
JMX Remote – client

- creating a connection to the server

  ```java
  JMXServiceURL url = new JMXServiceURL("service:jmx:rmi:///jndi/rmi://localhost:9999/server");
  JMXConnector jmxc = JMXConnectorFactory.connect(url, null);

  MBeanServerConnection mbsc = jmxc.getMBeanServerConnection();
  ```

- usage

  ```java
  mbsc.queryMBeans(ObjectName name, QueryExp query)
  mbsc.getAttribute(ObjectName name, String attrName)
  mbsc.setAttribute(ObjectName, Attribute attr)
  ```
JMX Remote – client

• creating a proxy object for direct access
  – it is necessary to know the interface
    • works for standard mbeans

T JMX.newMBeanProxy(MBeanServerConnection connection, ObjectName objectName, Class<T> interfaceClass)

T JMX.newMBeanProxy(MBeanServerConnection connection, ObjectName objectName, Class<T> interfaceClass, boolean notificationBroadcaster)