



Distributed Systems

Winter Term 2025/26

Roland Wismüller
Universität Siegen
roland.wismueller@uni-siegen.de
Tel.: 0271/740-4050, Büro: H-B 8404

Stand: November 27, 2025



Distributed Systems

Winter Term 2025/26

3 Distributed Programming



Content

- ➔ RMI
 - ➔ Introduction
 - ➔ *Hello World* with RMI
 - ➔ RMI in more detail
 - ➔ classes and interfaces, stubs, name service, parameter passing, factories, callbacks, ...
- ➔ gRPC



Literature

- ➔ WWW documentation and tutorials from Oracle
 - ➔ <https://docs.oracle.com/en/java/javase/17/docs/api/java.rmi/java.rmi/package-summary.html>
 - ➔ <https://docs.oracle.com/javase/8/docs/technotes/guides/rmi>
- ➔ Hammerschall: Ch.. 5.2
- ➔ Farley, Crawford, Flanagan: Ch. 3
- ➔ Horstmann, Cornell: Ch. 5
- ➔ Orfali, Harkey: Ch. 13
- ➔ gRPC home page
 - ➔ <https://grpc.io>

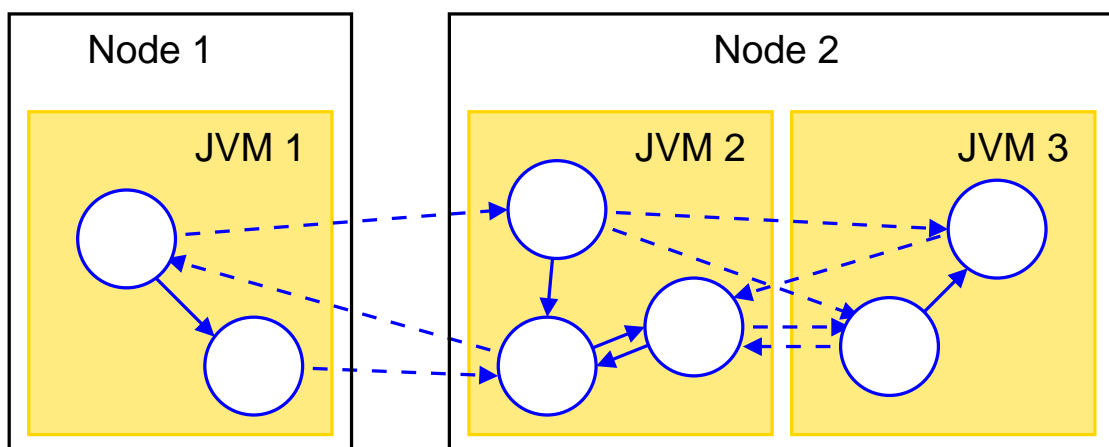
3.1.1 Introduction

- ➔ Java RMI supports the use of remote objects in Java
 - ➔ integral part of Java (package `java.rmi`)
- ➔ Essential elements:
 - ➔ remote object implementations
 - ➔ client interfaces (stubs) to remote objects
 - ➔ server skeletons for remote object implementations
 - ➔ name service to locate objects in the network
 - ➔ communication protocol
- ➔ All objects (i.e., client and server) must be programmed in Java
 - ➔ allows seamless integration into the language (including distributed garbage collection)
 - ➔ RMI/IIOP allows interoperability with CORBA

3.1.1 Introduction ...

(Animated slide)

Distributed Objects



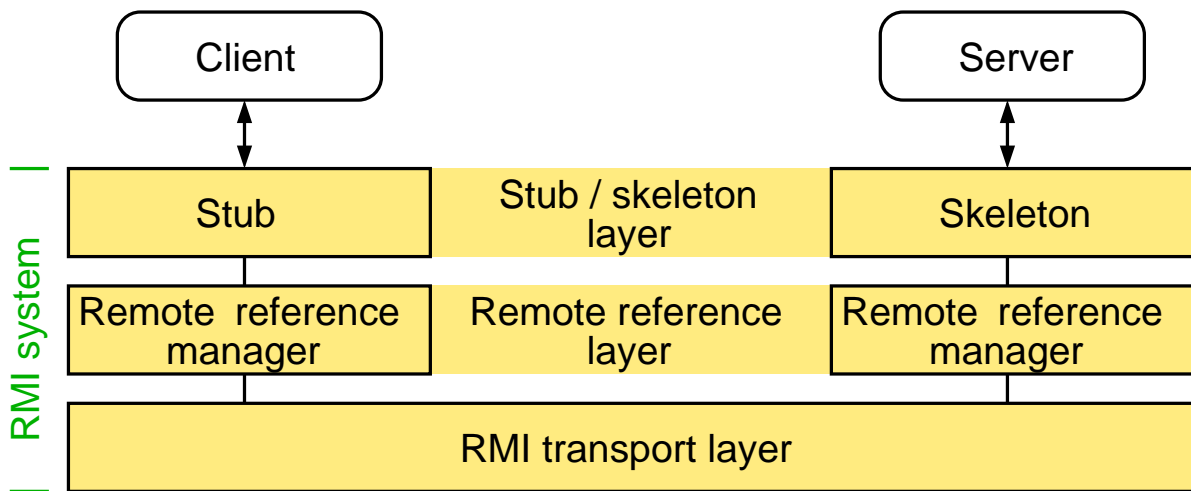
—→ local reference

- - -→ remote reference

- ➔ Remote references can be used just like local references
- ➔ Objects can occur in client and server roles



3.1.2 RMI Architecture



3.1.2 RMI Architecture ...



Stub/Skeleton Layer

- ➔ Stub and skeleton classes are automatically generated at runtime
 - ➔ from a Java interface

Remote Reference Layer

- ➔ Defines call semantics (unicast, multicast, object activation)
- ➔ Also: connection management, distributed garbage collection

Transport Layer

- ➔ Proprietary protocol: Java Remote Method Protocol (JRMP)
 - ➔ allows tunneling via HTTP (due to firewalls)
 - ➔ allows to use TLS via a socket factory
- ➔ Alternative: RMI-IIOP

Notes for slide 103:

- ➔ The skeleton class is generic and uses the Java reflection mechanism to call methods of server object. Reflection allows to query the method definitions of a class and to generically call methods at runtime.
- ➔ Stub classes are created at runtime using the class `java.lang.reflect.Proxy`.
- ➔ More information in Java reflection can be found, e.g., at <https://www.oracle.com/technical-resources/articles/java/javareflection.html>
- ➔ For more information on the Proxy class, see, e.g.: <https://docs.oracle.com/javase/8/docs/technotes/guides/reflection/proxy.html>

103-1

3.1 Java RMI ...



3.1.3 RMI Services

➔ Name service: RMI Registry

- ➔ registers remote references to RMI objects under freely selectable unique names
- ➔ a client can then get the corresponding reference for a name
 - ➔ registry sends a serialized proxy object (client stub) to the client.
- ➔ RMI can also be used with other naming services, e.g. via JNDI (Java Naming and Directory Interface)

Distributed Systems

Winter Term 2025/26

06.11.2025

Roland Wismüller
Universität Siegen
roland.wismueller@uni-siegen.de
Tel.: 0271/740-4050, Büro: H-B 8404

Stand: November 27, 2025

3.1.3 RMI Services ...

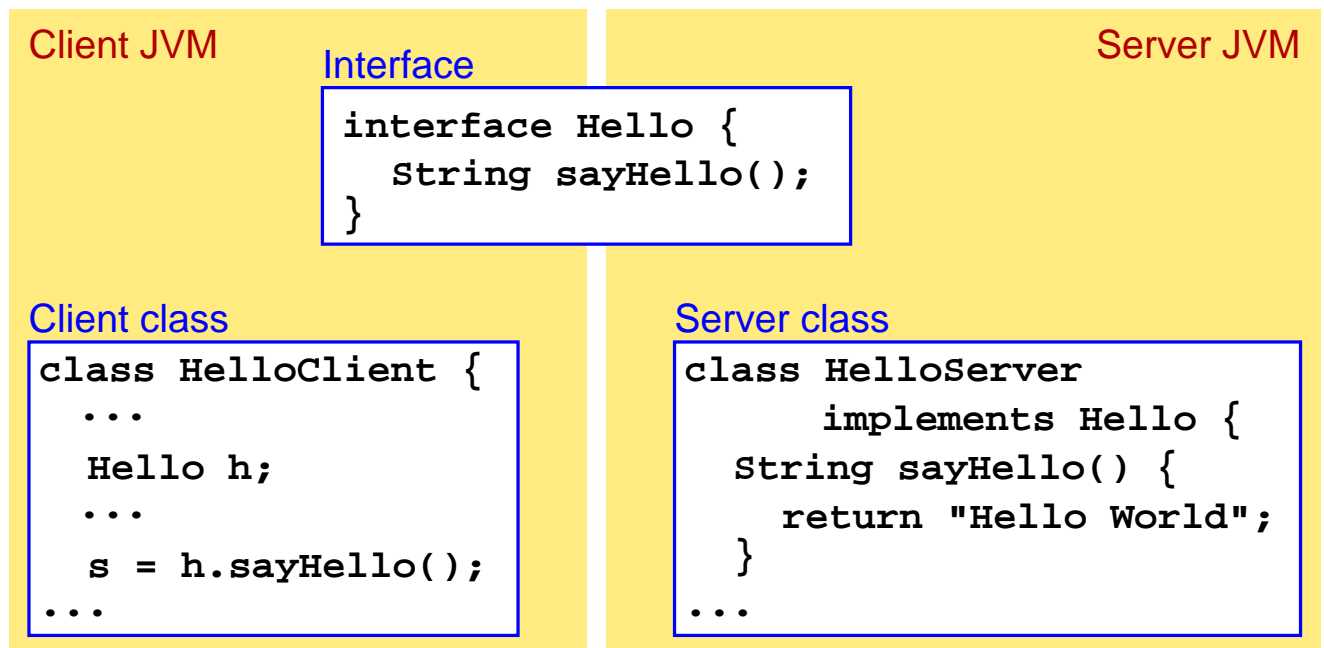


➔ Distributed Garbage Collection

- ➔ automatic garbage collection of Java also works for remote objects
- ➔ server-side JVM manages a list of remote references to objects
- ➔ references are “leased” for a certain time
- ➔ reference counter of the object is decremented, if
 - ➔ client deletes the reference (e.g., end of the lifetime of the reference variable), or
 - ➔ client does not renew the lease in time
 - ➔ reason: remote reference layer cannot explicitly “log off” an object, if the client crashes
 - ➔ default setting: 10 min.



3.1.4 Example: *Hello World* with Java RMI



3.1.4 Example: *Hello World* with Java RMI ...



Development Process:

1. Design the interface for the server object
2. Implement the server class
3. Develop the server application to include the server object
4. Develop the client application with calls to the server object
5. Compile and start the system



Designing the Interface for the Server Object

- ➔ Specified as normal Java interface
- ➔ Must extend `java.rmi.Remote`
 - ➔ no inheritance of operations, only marking as remote interface
- ➔ Each method must declare to raise the exception `java.rmi.RemoteException` (or a base class of it)
 - ➔ base class for all errors that may occur
 - ➔ in the client, during transmission, in the server
- ➔ No restrictions compared to local interfaces
 - ➔ but: semantic differences (parameter passing!)

Notes for slide 108:

- ➔ To test whether a reference is a remote reference (i.e., points to an object whose class implements `java.rmi.Remote`), use
`if (reference instanceof java.rmi.Remote)`



Hello-World Interface

```
import java.rmi.Remote;  
import java.rmi.RemoteException;  
  
public interface Hello extends Remote {  
    String sayHello() throws RemoteException;  
}
```

Marker interface,
contains no methods,
marks interface as
RMI interface

RemoteException
indicates error in the
remote object or
during communication



Implementing the Server Class

- ➔ A class that is to be usable remotely must:
 - ➔ implement a given remote interface
 - ➔ usually extend `java.rmi.server.UnicastRemoteObject`
 - ➔ defines call semantics: point-to-point
 - ➔ have a constructor that declares to throw a `RemoteException`
 - ➔ creation of object must be done in a try-catch block
- ➔ Methods usually do not need to specify `throws RemoteException`
 - ➔ because they don't throw the exception themselves

Notes for slide 110:

- ➔ The constructor must declare `throws RemoteException`, because the superclass constructor (from `UnicastRemoteObject`) may throw a `RemoteException`.
- ➔ The constructor of `UnicastRemoteObject` actually takes care of starting the server skeleton (in a new thread) or of registering the new object with an already existing skeleton.
- ➔ Since the skeleton will send and receive messages, it may throw an exception in case of a communication error.

110-1

3.1.4 Example: *Hello World* with Java RMI ...



Hello-World Server (1)

```
import java.rmi.*;
import java.rmi.server.UnicastRemoteObject;

public class HelloServer extends UnicastRemoteObject
    implements Hello {
    public HelloServer() throws RemoteException {
        super();
    }
    public String sayHello() {
        return "Hello World!";
    }
}
```

Remote method



Development of the Server Application to Include the Server Object

➔ Tasks:

- ➔ creating a server object
- ➔ registering the object with the name service
 - ➔ under a specified public name

➔ Typically not a new class, but `main` method of the server class



Hello-World Server (2)

```
public static void main(String args[]) {
    try {
        HelloServer obj = new HelloServer();
        Naming.rebind("rmi://localhost/Hello-Server", obj);
    }
    catch (Exception e) {
        System.out.println("Error: " + e.getMessage());
        e.printStackTrace();
    }
}
```

Create the server object

Register the server object under the name "Hello-Server" with the name server (RMI registry, local host, port 1099)



Development of the Client Application with Calls to the Server Object

- ➔ Client must first use the name service to get a reference to the server object from the name service
 - ➔ type cast to the correct type required
- ➔ Then: any method can be called
 - ➔ no syntactical differences to local calls
- ➔ Note: client can get remote references in other ways as well
 - ➔ e.g. as return value of a remote method



Hello-World Client

```
import java.rmi.*;

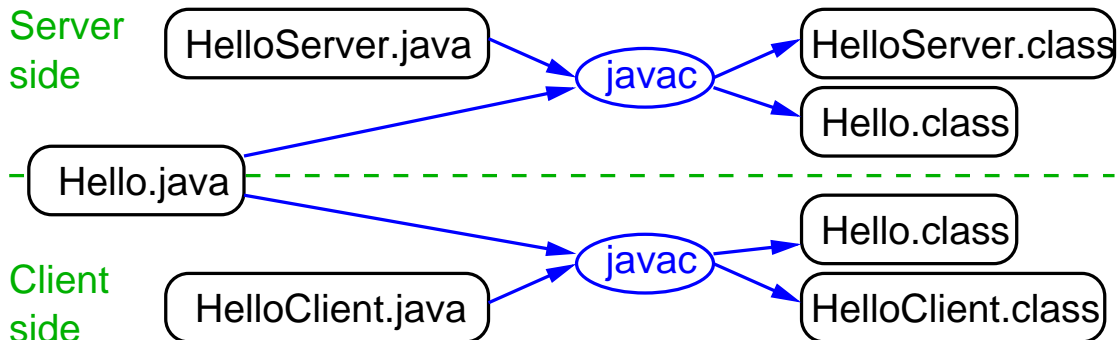
public class HelloClient {
    public static void main(String args[]) {
        try {
            Hello obj =
                (Hello)Naming.lookup("rmi://bspc02/Hello-Server");
            String message = obj.sayHello();
            System.out.println(message);
        }
        catch (Exception e) {
            ...
        }
    }
}
```

Get object reference from name server

Call the method on the remote object

Compiling and Starting the System

➔ Compiling:



➔ Starting the naming service: `rmiregistry [port]`

➔ for security reasons, objects can only be registered on the local host, i.e. RMI registry must run on server computer

➔ Starting the server: `java HelloServer`

➔ Starting the client: `java HelloClient`

Notes for slide 116:

The example assumes that the class `Hello.class` is found using the local classpath:

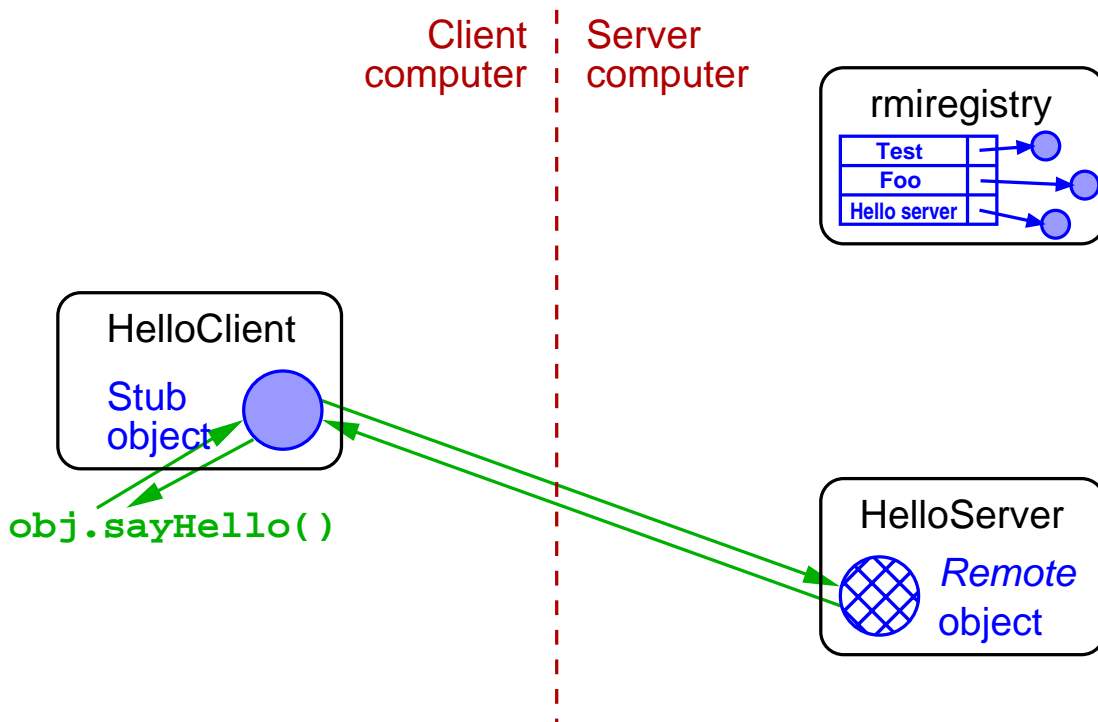
- ➔ when starting `rmiregistry`
- ➔ when starting `HelloServer`
- ➔ when compiling and starting `HelloClient`

3.1.4 Example: *Hello World* with Java RMI ...



(Animated slide)

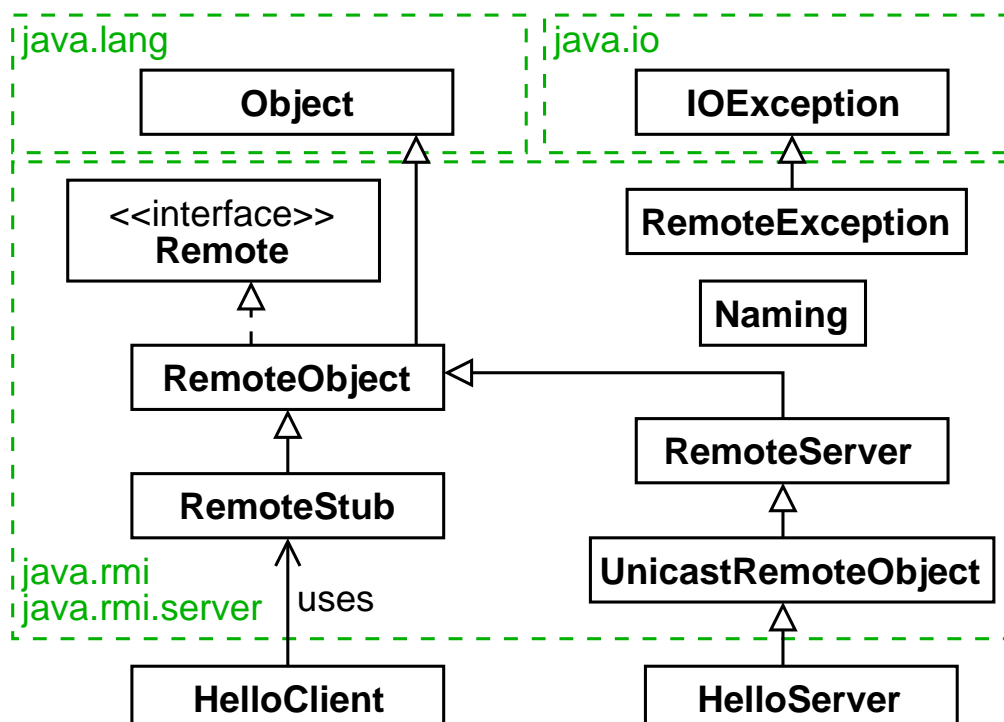
Execution of the Example



3.1 Java RMI ...



3.1.5 Classes and Interfaces





Interface `Remote`

- ➔ Every remote object must implement this interface
- ➔ Does not provide methods, serves only as a marker

Class `RemoteException`

- ➔ Superclass for all exceptions that can be triggered by the RMI system, for example, with
 - ➔ communication errors (server not reachable, ...)
 - ➔ (un-)marshalling errors
 - ➔ protocol errors
- ➔ Each remote method must specify `RemoteException` (or a base class of it) in the `throws` clause



Class `RemoteObject`

- ➔ Base class for all remote objects
- ➔ Redefines the methods `equals`, `hashCode`, and `toString`
- ➔ Static method `toStub()` returns a reference to the stub object

Class `RemoteStub`

- ➔ Base class for all client stubs

Class `RemoteServer`

- ➔ Base class for `UnicastRemoteObject`
- ➔ Method `getClientHost()`: host address of the client of the current RMI call
- ➔ `setLog()` and `getLog()`: logging of RMI calls

Class `UnicastRemoteObject`

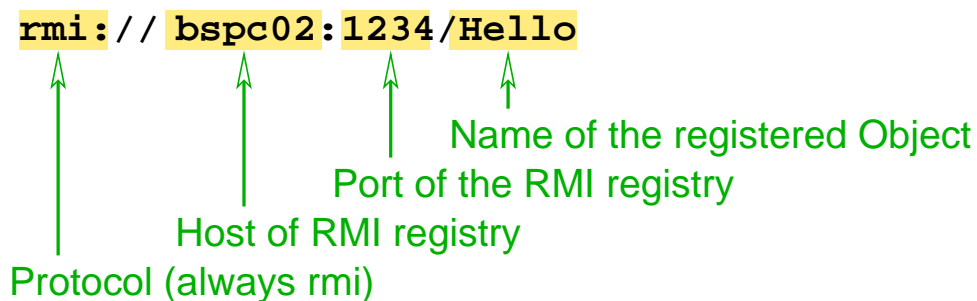
- ➔ Implements remote object with the following properties:
 - ➔ references to the object are only valid as long as server process (JVM) is still running
 - ➔ client call is routed to exactly one object (via TCP connection), no replication
- ➔ Constructor allows definition of port and socket factories
 - ➔ so that e.g. connections via TLS/SSL can be realized
- ➔ Static method `exportObject()` makes object available via RMI
- ➔ Static method `unexportObject()` cancels availability

Notes for slide 121:

- ➔ The constructor of `UnicastRemoteObject` actually creates the server skeleton for the server object (or registers the server object with an already existing skeleton), so that it can be contacted remotely.
The server skeleton is executed by a separate thread. This is the reason why the server process doesn't terminate, even when the `main()` routine returns.
- ➔ Instead of extending `UnicastRemoteObject`, it is also possible to export the server object(s) by calling `exportObject()`, which will then create the skeleton, as described above. Using this method is e.g. necessary, if the server class must (or should) extend some other class, as Java does not support multiple inheritance.
- ➔ The method `unexportObject()` deregisters the server object from the skeleton and destroys the skeleton in case of the last object.

Class Naming

- ➔ Allows easy access to RMI registry
- ➔ Important methods:
 - `bind()` / `rebind()`: registers object under given name
 - `lookup()`: get object reference to a name
- ➔ Names are given in URL format
 - also define the host and port of the RMI registry.
 - structure of the URL:



Notes for slide 122:

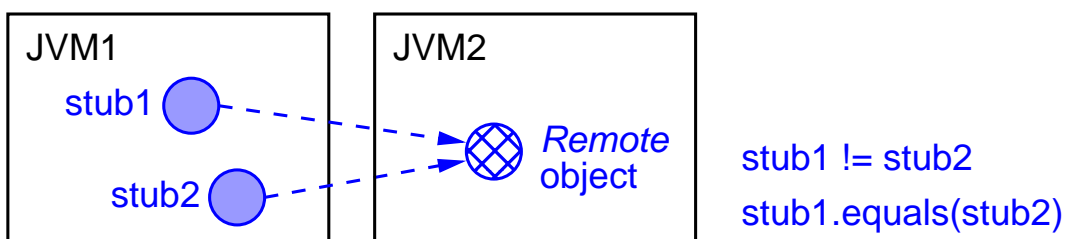
The method `rebind()` overwrites an existing entry with the same name, while `bind()` throws an exception in this case.

Another more flexible way to access the RMI registry is to use the `LocateRegistry` class and the `Registry` interface in the `java.rmi.registry` package.



3.1.6 Special Characteristics of Remote Classes

- ➔ Comparison of remote objects
 - Comparison with `==` refers only to the stub objects
 - Result is `false`, even if both stubs refer to the same remote object
 - comparison with `equals()` returns true if both stubs refer to the same remote object



3.1.6 Special Characteristics of Remote Classes ...



- ➔ Method `hashCode()`
 - used by container classes `HashMap`, `HashSet` and others
 - Hash code is calculated only from the object identifier of the remote object
 - same remote object \Rightarrow same hash code
 - but the content of the object is ignored
 - consistent with behavior of `equals()`
- ➔ Cloning objects
 - cloning of the remote object is not possible by calling `clone()` on the stub
 - cloning of stubs neither necessary nor meaningful



3.1.7 Parameter Passing

- ➔ Parameters passed to remote methods
 - either via *call-by-value*
 - or via *call-by-reference*
- ➔ The mechanism used depends on the type of the parameter
- ➔ Final decision may only be made at runtime!

- ➔ The return of the result follows the same rules as for parameter passing

3.1.7 Parameter Passing ...



Parameter Passing for Local Methods

- ➔ Java supports two kinds of types:
 - **value types**: simple data types
 - boolean, byte, char, short, int, long, float, double
 - are passed to local methods *by value*
 - that is, the method receives a copy of the value
 - **reference types**: classes (incl. String and arrays)
 - are passed to local methods *by reference*
 - that is, the method works on the original object and can also change object if required



Parameter Passing for Remote Methods

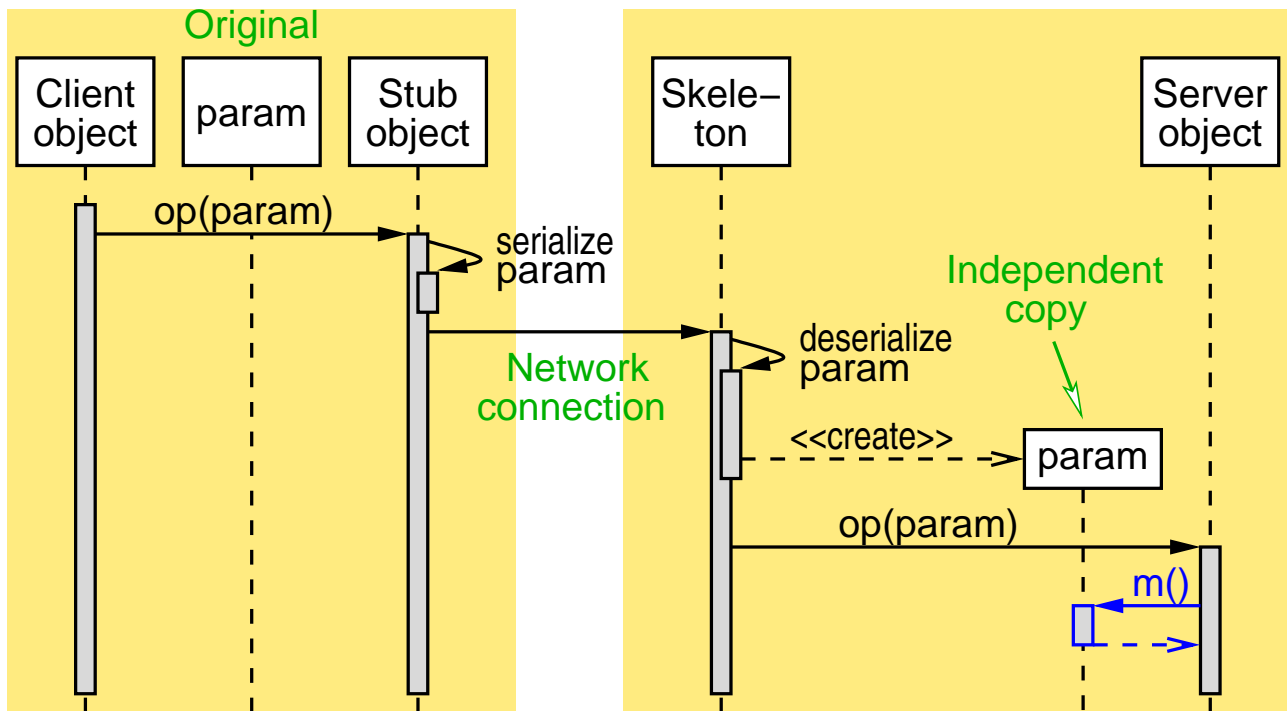
- ➔ Value types: are always passed *by value*
- ➔ Reference types: dependent on the concrete object
 - object can be serialized: *call-by-value*
 - object belongs to a class that implements the `Remote` interface: *call-by-reference*
 - neither: error (`java.rmi.MarshalException`)
 - both: ??! (this case is to be avoided!)
 - decision is made only at runtime



Call-by-Value (Serializable Objects)

- ➔ Class must implement interface `java.io.Serializable`
- ➔ Serializable objects can be transferred over a network
 - only the data is transferred, the code (`class` file) must be available at the receiver!
- ➔ Default serialization of Java:
 - all attributes of the object are serialized and transferred
 - recursive procedure!
 - prerequisite: all attributes and all base classes can be serialized
- ➔ Application specific serialization is possible:
 - implement the methods `writeObject` and `readObject`

Passing a Serializable Object

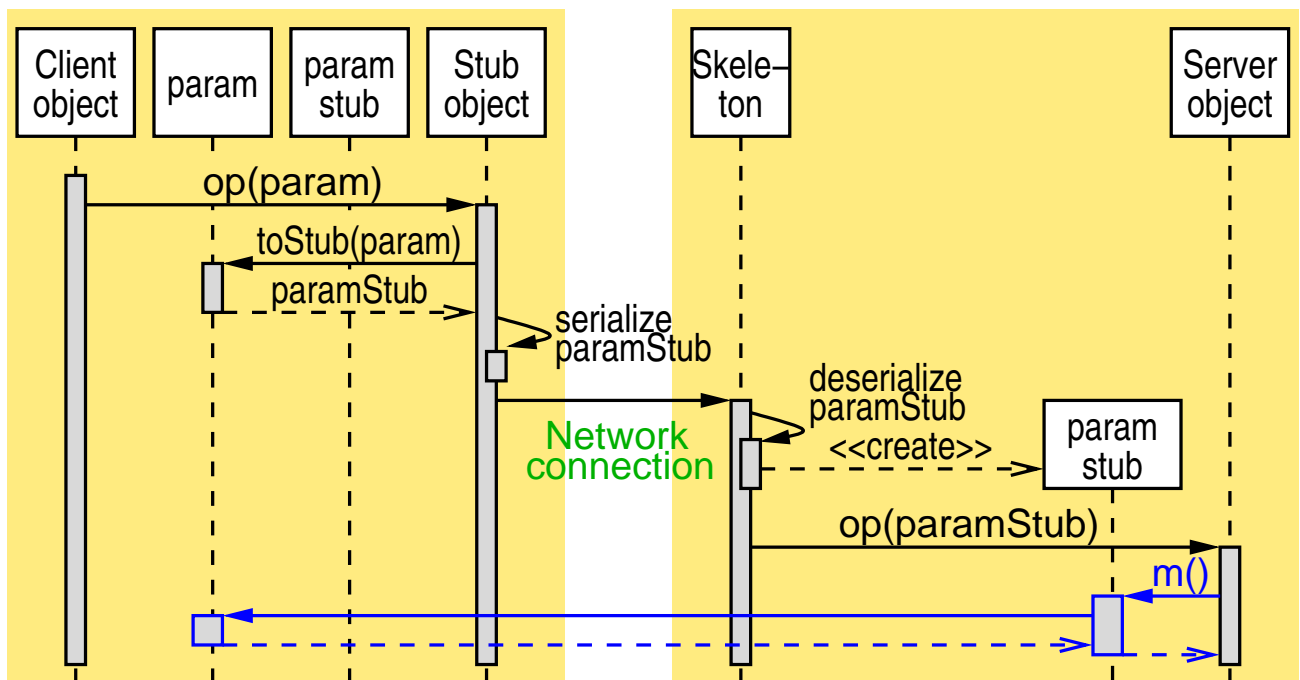


Call-by-Reference (Remote Objects)

- ➔ Class of the parameter object must implement an interface that extends `Remote`
 - ➔ parameter type must be this interface
 - ➔ class is typically derived from `UnicastRemoteObject`
- ➔ A serialized stub object is transferred
 - ➔ stub class is created dynamically
- ➔ If the server calls methods on the parameter object:
 - ➔ calls are routed to the original object using RMI



Passing a Remote Object



Examples

- ➔ See WWW:
 - ➔ Hello-World with call-by-value parameter
 - ➔ Hello-World with call-by-reference parameter



Arrays and Container Objects

- ➔ Arrays and container objects (from the Java Collection Framework, `java.util`) can be serialized
 - ➔ i.e., they will be reinstantiated at the receiver
- ➔ To the elements of the array / container the same rules apply as to simple parameters
 - ➔ for mixed content: elements are passed *by value* or *by reference* depending on their actual class

3.1 Java RMI ...



3.1.8 Remote Object References as Results

- ➔ Frequently: via RMI registry, the client receives a reference to a remote object, which provides references to other objects
 - ➔ the remote object may also create these objects on demand (*factory object / factory class*)
- ➔ Example: server for bank accounts
 - ➔ registration of all account objects with RMI registry not useful
 - ➔ instead: registration of a manager object that returns the reference to the account object for a given account number
 - ➔ if necessary, it can create a new object (from a database)
- ➔ Note: RMI does not allow remote object creation
 - ➔ client cannot create objects on a remote host

3.1.9 Client Callbacks

- ➔ Frequently: server wants to make calls in the client
 - ➔ e.g. progress bar, queries, ...
- ➔ For this: client object must be an RMI object
 - ➔ pass `this` reference to the server method
- ➔ In some cases, you cannot inherit from `UnicastRemoteObject`, e.g. for applets
 - ➔ then: export the object using
`UnicastRemoteObject.exportObject(obj, 0);`
- ➔ Example code: see WWW (*Hello-World with callback*)

Notes for slide 135:

- ➔ The second parameter of `exportObject()` is the port on which the server object listens. A 0 means to choose any free port.
- ➔ There is also a method `exportObject()` with only one argument, which is deprecated because it does not support dynamic stubs.



3.1.10 RMI and Threads

- ➔ RMI does not specify how many threads are provided on the server side for method calls
 - ➔ only one thread, one thread per call, ..
- ➔ This means that several server methods can be active at the same time
 - ➔ requires correct synchronization (synchronized)!
- ➔ Client-side locking of a remote object using a synchronized block is not possible
 - ➔ only local stub is locked
 - ➔ a lock must be implemented using methods of the remote object if necessary



Distributed Systems

Winter Term 2025/26

13.11.2025

Roland Wismüller
Universität Siegen
roland.wismueller@uni-siegen.de
Tel.: 0271/740-4050, Büro: H-B 8404

Stand: November 27, 2025

3.1.11 Deployment

- ➔ **Deployment:** distribution, transfer and installation of the components of a distributed application
 - ➔ specifically for RMI: which `class` file has to go where?
- ➔ Server, RMI registry and client need the `class` files for:
 - ➔ their own implementation
 - ➔ the remote server interface plus all classes / interfaces used therein (recursively)
- ➔ Client and server in addition need the classes of received serializable objects
- ➔ RMI allows remote loading of the required `class` files
 - ➔ security issue, especially since Java Security Manager is deprecated

Notes for slide 137:

- ➔ Remote class loading has a severe security risk, as it loads code from possibly unknown / untrusted locations.
- ➔ Java used to have a Security Manager, which allows to 'sandbox' the loaded code, i.e., it allows to deny certain actions of the code like accessing the file system.
- ➔ Oracle deprecated this Security Manager in Java 17 without any replacement (see <https://openjdk.org/jeps/411>).
- ➔ Therefore, remote class loading is no longer discussed in detail in this lecture.



3.2 gRPC (Google Remote Procedure Call)

- ➔ Open source framework for RPC-based services
- ➔ Support for many languages (C++, Go, Java, Python, Rust, ...)
- ➔ Service is described by a special interface definition language (*protocol buffer language*)
 - defines structure of messages
 - defines request and reply message for each procedure
- ➔ *Protocol buffers*: binary serialization format for messages
 - better efficiency and type safety than JSON
- ➔ Communication is based on HTTP/2
 - binary format, can multiplex several streams, allows server push messages

3.2 gRPC (Google Remote Procedure Call) ...



Example: Interface Definition

```
syntax = "proto3"; // Use protocol buffers revision 3

service Greeter {
    // Procedure with request and reply message
    rpc SayHello (HelloRequest) returns (HelloReply);
}

message HelloRequest { // Request message format
    string name = 1; // '1' is the field number
    int32 number = 2;
}

message HelloReply { // Reply message format
    string greeting = 1;
}
```

3.2 gRPC (Google Remote Procedure Call) ...



(Animated slide)

Example: Server

```
import io.grpc.Grpc;
import io.grpc.InsecureServerCredentials;
import io.grpc.Server;
import io.grpc.stub.StreamObserver;

public class HelloWorldServer {
    public static void main(String[] args) throws Exception {
        Server server = Grpc.newServerBuilderForPort(12345,
            InsecureServerCredentials.create())
            .addService(new GreeterImpl())
            .build()
            .start();
        server.awaitTermination();
    }
}
```

3.2 gRPC (Google Remote Procedure Call) ...



(Animated slide)

Example: Server ...

```
class GreeterImpl extends GreeterGrpc.GreeterImplBase {
    @Override
    public void sayHello(HelloRequest req,
        StreamObserver<HelloReply> resp) {
        String hello = "Hello " + req.getName()
            + ", " + req.getNumber();
        HelloReply reply = HelloReply.newBuilder()
            .setGreeting(hello)
            .build();
        resp.onNext(reply);
        resp.onCompleted();
    }
}
```

3.2 gRPC (Google Remote Procedure Call) ...



(Animated slide)

Example: Client

```
import io.grpc.Grpc;
import io.grpc.InsecureChannelCredentials;
import io.grpc.ManagedChannel;

public class HelloWorldClient {
    public static void main(String[] args) throws Exception {
        ManagedChannel channel =
            Grpc.newChannelBuilder("localhost:12345",
                InsecureChannelCredentials.create())
                .build();
```

3.2 gRPC (Google Remote Procedure Call) ...



(Animated slide)

Example: Client ...

```
GreeterGrpc.GreeterBlockingStub blockingStub =
    GreeterGrpc.newBlockingStub(channel);
HelloRequest req = HelloRequest.newBuilder()
    .setName("world")
    .setNumber(42)
    .build();
HelloReply resp = blockingStub.sayHello(req);
System.out.println(resp.getGreeting());
channel.shutdown();
}
```

Details on the Protocol Buffer Language

- ➔ The `protoc` compiler creates stubs, skeletons and interfaces for the implementation language
- ➔ Supported data types:
 - simple types: `int32`, `uint64`, `double`, `string`, `bool`, ...
 - enums
 - arrays, e.g. `repeated int32 val = 1;`
 - maps, e.g. `map<string, int32> size = 3;`
 - in addition, messages can be nested
- ➔ Fields can be optional, e.g. `optional int32 val = 1;`
 - if the field is not set, it gets a fixed default value
- ➔ RPCs may also receive and return *streams* of messages
- ➔ Explicit field numbers support forward and backward compatibility

Notes for slide 144:

The protocol buffer language supports different kinds of integer numbers:

int32/int64: variable-length encoded, signed, but inefficient for negative numbers

uint32/uint64: variable-length encoded, unsigned

sint32/sint64: variable-length encoded, signed, more efficient than `int32/int64`

fixed32/fixed64: fixed length encoding, unsigned

sfixed32/sfixed64: fixed length encoding, signed



Example: Replying a Stream of Messages

- ➔ Interface definition:
rpc SayHello (HelloRequest) returns (stream HelloReply);
- ➔ Server: sends several reply messages using onNext()
- ➔ Client: uses an asynchronous stub and a stream observer

```
GreeterGrpc.GreeterStub stub =  
    GreeterGrpc.newStub(channel);  
HelloRequest req = HelloRequest.newBuilder()  
    .setName("world").setNumber(42).build();  
stub.sayHello(req, new StreamObserver<HelloReply>() {  
    @Override public void onNext(HelloReply resp) {  
        System.out.println(resp.getGreeting());  
    }  
    ...  
});
```



Compatibility Issues in Client/Server Interfaces

- ➔ Client and server code evolves over time
 - message fields may be added or removed
 - the type of a message field may change
 - problem when client and server have different versions
- ➔ Safe changes (among others):
 - adding a field (old receiver code ignores it)
 - removing a field (old receiver code gets the default value)
 - reserved keyword prevents field number from being reused
- ➔ Unsafe changes (among others):
 - changing field number of existing field
 - reusing an old field number for a new field

3.3 Summary

➔ RMI

- ➔ allows transparent access to remote objects via proxy objects
 - ➔ proxy classes are generated automatically at runtime
- ➔ parameter passing semantics
 - ➔ *by value*, if parameter object can be serialized
 - ➔ *by reference*, if parameter object is an RMI object

➔ gRPC

- ➔ service oriented: procedures with request / reply messages
 - ➔ including support for streams
- ➔ stubs / skeletons generated by `protoc` compiler
- ➔ supports encryption and authentication