UNIVERSITÄT SIEGEN

Fakultät IV
Betriebssysteme und verteilte Systeme
Prof. Dr. rer. nat. Roland Wismüller

Excercise Sheet 4
Solution

Lecture Distributed Systems
Summer Term 2020

Exercise 1: Programming: Java-RMI - Follow-up example

You will find the solution to this problem in in the archive `l04eFiles.zip`[1] on the lecture's web page.

Exercise 2: Programming: Generic proxy objects

You will find the solution to this problem in in the archive `l04eFiles.zip`[2] on the lecture's web page.

Exercise 3: Generic dispatcher with Java Reflection

a) The request message must in any case contain the following information:

- **(i)** an identifier for the object on which the method is to be called,
- **(ii)** a specification of the method to be called,
- **(iii)** the arguments for the method call.

In the simplest case, the object ID can be a sequence number. The server skeleton must then, for example, maintain an array that contains references to all remote objects of the server.

The method to be called could be specified by its name. Due to the possibility of overloading, however, this may not be sufficient, but the parameter types must also be specified. The Java class `Class` contains a method `getMethod()`, which is used to find the corresponding `Method` object for a given name and the parameter types.[3] Since `Method` itself cannot be serialized, the message cannot directly contain the `Method` object.

The arguments for the method call are most easily transferred in Java as serialized objects. The arguments for RMI must always be either serializable or remote objects. In the latter case, the associated stub object is serialized during the call.

The Java data type for a query message could therefore look as follows:

```
public class RequestMessage implements Serializable
{
   int        objectID;
   String     methodName;
   Class<?>[] argumentTypes;
   Object[]   arguments;
   ...
}
```

b) With the above query message type, the code for a generic dispatcher might look something like this:

[1] `http://www.bs.informatik.uni-siegen.de/web/wismueller/vl/vs/l04eFiles.zip`
[2] `http://www.bs.informatik.uni-siegen.de/web/wismueller/vl/vs/l04eFiles.zip`
[3] However, the parameter types must match the specified types exactly. In principle, the method could also be determined automatically from the types of the passed arguments. However, these can be subclasses of the parameter types, so that the search for the suitable method is not trivial. The Java class `Class` does not offer a method for this either.

```
public void dispatch(RequestMessage request)
{
   try {
      // Determine reference to target object from objectID
      Object target = objectTable[request.objectID];
      // Determine the appropriate Method object via the Class object
      Method method = target.getClass().getMethod(request.methodName,
                                                  request.argumentTypes);
      // Call method
      Object result = method.invoke(target, request.arguments);
      ... pack result into reply message package and send back
   }
   catch (InvocationTargetException e) {
      // Called method threw an exception
      ... pack e.getCause() into reply message package and send back
   }
   catch (Exception e) {
      // Exception when calling the method (e.g. IllegalArgumentException)
      ... pack e into reply message package and send back
   }
}
```

Exercise 4: Parameter passing

If *call-by-reference* is used, a pointer to `i` is passed to `incr`. This will increment `i` twice, so the end result in this case is 2.

With *call-by-value* the value is passed directly and no reference to `i`. Therefore `i` remains unchanged for the caller, i.e. 0.

Call-by-copy/result means: Copying the parameter values when calling the procedure, copying back and overwriting the call parameters when the procedure is finished. In this case, `i` is passed as a value twice (i.e. 0 each) and incremented so that both values are now 1. If they are copied back, the second copy overwrites the first and the final value is now also 1.

Exercise 5: Transparency of Java RMI

The solutions of the programming task can be found in the archive `l04eFiles.zip`[4] on the lecture's web page. The problem is the parameter passing in Java RMI, which has a *call by value* semantics for serializable objects (and `ArrayList` is serializable). I.e., the server sorts the list correctly, but the result has no influence on the client.

To fix the problem, you might get the idea to make the list a remote object located at the client, so that the server can modify it via RMI. However, since sorting an `ArrayList` results in an extremely large number of accesses, this procedure would be extremely inefficient. It is therefore more efficient to realize a parameter passing with *call by value and result* semantics by having the `sort()` method of the remote interface `SortServer` return the sorted list as the return value and `sort()` method of the wrapper class copy this return value back to the original list.

[4] `http://www.bs.informatik.uni-siegen.de/web/wismueller/vl/vs/l04eFiles.zip`