Compiler support for an RMI implementation using NexusJava

Fabian Breg Dennis Gannon

December 16, 1997

1 Introduction

Java [7] is a portable, object oriented programming language. Its portability is obtained by compiling Java source code into **bytecode** which can be directly executed by a Java Virtual Machine (JVM) running on any host.

The main operation in the Java language is the invocation of a **method** on an object. These methods perform some action after which control is returned to the invoking method. Typically, only methods of objects residing in the same JVM can be invoked from other objects. Communication with objects located in a different JVM is a tedious programming task and therefore error-prone.

Because, however, there is a growing demand for distributed applications, a number of distributed object models have been developed for the Java programming language that simplify distributed programming in an object oriented environment. Examples of such distributed object models are Voyager [9], Infospheres [4] and Java RMI [10]. While the former two aim at a flexible and highly dynamic framework, the last one aims at providing a syntax for remote method invocation identical to the syntax for method invocation on local objects.

Another goal when designing distributed object technology is to provide inter-operability between different languages. The CORBA project [8] is designed to enable objects written in any language to invoke methods on remote objects written in any other language.

In this report we take a look at our implementation of Java RMI on top of the Nexus communication library. This project is part of a larger project, which aims at inter-operability between various distributed object models. Our goal is to have (a subset of) our RMI implementation talk to remote objects written in HPC++ [1], which are built on top of the Nexus communication library. Since Nexus and NexusJava are already fully inter-operable and both portable, these libraries provide an excellent basis for communication in our project.

The outline of this report is as follows: Section 2 gives an overview of both Java RMI and NexusJava. Section 3 describes the implementation of the standard RMI classes as well as the classes that support our implementation. Section 4 describes the generation of stubs and skeletons. Our approach to object serialization is described in Section 5. Section 6 concludes this report.

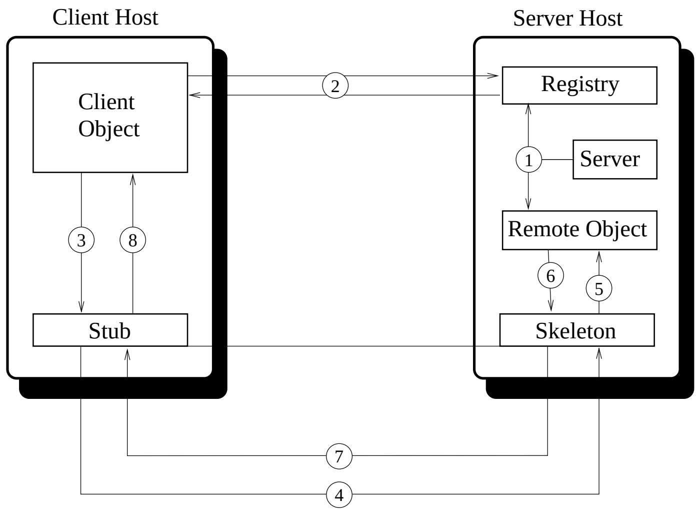

Figure 1: RMI overview

2 Preliminaries

This section first gives an overview of Java RMI, describing the most important classes and methods involved with an example. For a more rigorous description of the RMI interface we refer to [10]. After that we will describe the NexusJava library for communication, describing the most important classes and methods with a simple example.

2.1 Java RMI

To provide communication between objects in different JVM's, the Java API contains classes that implement the socket mechanism. Although the Java API provides a simple interface for programming sockets, applications still have to implement a protocol for encoding and decoding messages, which is a cumbersome and error-prone task.

Java RMI is designed to simplify the communication between two objects on separate machines by allowing an object to invoke the methods of an object on a remote machine in the same way as methods on local objects are invoked. To invoke methods of a **remote object**, a **remote reference** to that object has to be obtained. Since the object resides in a different name-space, a **registry** is used to manage remote references; RMI servers can register remote objects at the registry after which clients can obtain a reference to these remote objects.

An overview of RMI is shown in Figure 1. First, a **server** creates a remote object and registers it at the registry, which is represented by arrows (1) in the figure. The client can obtain references to objects stored in the registry as shown by arrows (2). When the client invokes a method on a remote object, the method is actually invoked on a **stub** object, located on the same JVM, instead (3). This stub object makes a message containing the name of the method together with its parameters, a process called **marshalling**, and sends this message to the associated **skeleton** object residing on the server host (4). The skeleton object extracts the

```
public interface Weather extends java.rmi.Remote
{
  public String getForecast() throws java.rmi.RemoteException;
}
```

```
public class WeatherImpl extends java.rmi.server.UnicastRemoteObject implements Weather
{
  public String getForecast() throws java.rmi.RemoteException
  {
    return "It will be a sunny day !";
  }
}
```

```
public class Server
{
  public static void main(String[] Args)
  {
    try {
      System.setSecurityManager(new java.rmi.RMISecurityManager);
      java.rmi.registry.LocateRegistry.createRegistry(2099);
      WeatherImpl w = new WeatherImpl();
      java.rmi.Naming.bind("rmi://rainier.extreme.indiana.edu:2099/WeatherService", w);
    } catch(java.rmi.Exception e) { }
  }
}
```

```
public class Client
{
  public static void main(String[] Args)
  {
    try {
      Weather w =
         (Weather)java.rmi.Naming.lookup("rmi://rainier.extreme.indiana.edu:2099/WeatherService");
      System.out.println(w.getForecast());
    } catch(java.rmi.RemoteException e) { }
  }
}
```

Figure 2: Example RMI application

method name and parameters from the message, a process called **unmarshalling**, and invokes the appropriate method on the remote object with which it is associated (5). The remote object executes the method and passes the return value back to the skeleton (6). The skeleton in its turn marshals the return value in a message and sends this message to the stub object (7). The stub unmarshals the return value from the message and returns this value to the client program (8).

An example of an RMI based application is given in Figures 2. The methods of a remote object that can be invoked remotely must be specified in an interface that extends the `Remote` interface, which itself is an empty interface. Every method in the interface must be declared to throw the `RemoteException` in order to account for errors during the remote method invocation; a number of subclasses of RemoteException exist which represent the various errors that can occur.

A remote object should be declared to implement at least one remote interface and to extend `RemoteServer` class or one of its subclasses. Typically, the `UnicastRemoteObject` class is extended, by remote object classes. After creating a remote object it has to be exported; if the remote object is declared to extend `UnicastRemoteObject` the object is automatically exported during creation, otherwise it must be explicitly exported using the static method `exportObject()` of the `UnicastRemoteObject` class, providing the remote object as parameter. Once created and exported, a remote object must be registered using either the `bind()` or `rebind()` method of the `Naming` class, providing the remote object and an appropriate name for it.

Creating, exporting and registering remote objects is done by a server object. The server can either use an existing registry or create a registry itself using the `createRegistry()` method of the `LocateRegistry` class, providing the port number to which the registry should listen. To ensure security, the server must install an RMI specific security manager using the `setSecurityManager()` method providing an instance of the `RMISecurityManager` class.

The remote interface of a remote object is used as the type of the remote reference at the client side. A remote reference can be obtained by invoking the `lookup` method of the Naming class, providing the name of the desired remote object. When invoking a method through the remote interface, the `RemoteException` must be caught to handle errors during the remote method invocation. The kind of exception thrown to the client is dependent on the type of exception that occurred during the remote method invocation. Runtime Exceptions are encapsulated in a `ServerRuntimeException`, errors are encapsulated in a `ServerError`, remote exceptions are encapsulated in a `ServerException` and exceptions thrown outside the remote method body are encapsulated in an `UnexpectedException`.

After writing and compiling the remote interface and the remote object implementation, the RMI stub/skeleton compiler, **rmic**, can be used to generate a stub and skeleton pair for the remote object. This way, the stub and skeleton objects, and thus the entire communication, are completely shielded from the programmer.

2.2 NexusJava

Nexus [5] is a communication library, providing dynamic resource management, multithreading and multiple methods for communication, allowing it to operate in a heterogeneous environment. NexusJava [6] is an implementation of a subset of Nexus in Java.

The Nexus interface is organized around six basic abstractions, which are illustrated in Figure 3. A **node** represents a physical processing resource. On a node, multiple **contexts** can be running, which can be considered to be equal to a JVM. In each context, multiple **threads** can be present. Communication is performed over a communication link which is created by binding a **startpoint** to an **endpoint**. To invoke methods on user objects associated with an endpoint, a **remote service request** (RSR) can be issued on the startpoint connected to that endpoint. When an RSR is issued on a startpoint, a message containing a **handler identifier** and a data buffer is sent over the communication link to the endpoint, after which the method specified by the handler is invoked on the user object providing the buffer as data.

Figures 4 and 5 show how to implement a client and server in NexusJava. The server is declared to implement the `AttachApprovalInterface` which allows it to handle attach requests with the `attach_approval()` method. Implementing the `HandlerInterface` allows an object to invoke the `invoke_handler()` method to dispatch incoming messages to the appropriate objects.

Both client and server begin by creating a new object of class `Nexus` and initialize it by invoking `init()` on it. The server than allows clients to attach by invoking `allow_attach()` on the `nexus` object with the port number to listen to, itself as the object to handle incoming attach requests and a user object. The client attaches to the server by invoking `attach()` supplying the url of the server. When an attach request arrives from a client, the `attach_approval()` method

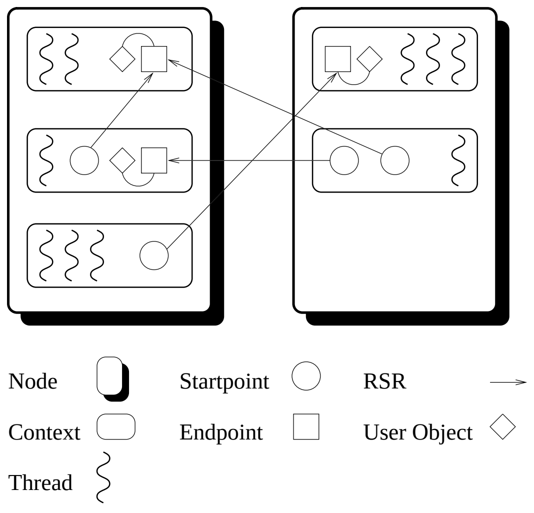

Figure 3: Nexus abstractions

is invoked with the user object and the url of the server. This method creates an `Endpoint`, by invoking `endpoint_init()` on the `nexus` object, with the server as dispatcher specified in the `enpointattr_init()`, and associates the user object with it by invoking `set_user_object()` on the endpoint, supplying the user object. Next a `Startpoint` is bound to the endpoint by invoking `startpoint_bind()` after which the result is passed back to the client as an `AttachReturn` object. The client can obtain a startpoint to the server from this `AttachReturn` object.

The client also creates an endpoint and associated startpoint. The client handles all incoming messages itself, so no object is associated with the endpoint. To send an RSR to the server, the client first creates a `PutBuffer` by invoking `buffer_init` on the `nexus` object, providing the size of the buffer. The buffer can now be filled using the various `put_`-methods from the `PutBuffer` class. In this case, the buffer only contains the startpoint to the client to which the server can reply.

The actual RSR is initiated by invoking `send_rsr()`, supplying the data buffer, the server startpoint, the handler identifier and a flag whether the data buffer should be destroyed afterwards. The message is handled at the server by the `invoke_handler()` method which is invoked automatically when a message arrives. The endpoint to which the request was sent, the `GetBuffer` and the `handler_id` are passed to this method. This method than invokes the appropriate method on the user object associated with the endpoint. This method can extract the data from the `GetBuffer` with the various `get_`-methods of the `GetBuffer` class. Eventually, the method returns a message to the client startpoint.

3 RMI classes

Implementing the RMI protocol involves implementing the classes from the RMI interface as described in [10]. Most of these classes involve exceptions, which can be copied directly into our

```
class Weather
{
  private final int REPLY = 0;
  private Nexus nexus;
  private String forecast = "It will be a nice day!";

  Weather(Nexus nexus)
  {
    this.nexus = nexus;
  }

  public void getForecast(GetBuffer inbuf)
  {
    try {
      Startpoint sp = new Startpoint(nexus, inbuf);
      int bufsize = nexus.sizeof_char(forecast.length()) + nexus.sizeof_int(1);
      PutBuffer outbuf = nexus.buffer_init(bufsize, 0);
      outbuf.put_int(forecast.length());
      outbuf.put_char(forecast.toCharArray(), 0, forecast.length());
      nexus.send_rsr(outbuf, sp, REPLY, false);
    } catch(Exception e) { }
  }
}

public class Server implements AttachApprovalInterface, HandlerInterface
{
  private final int GET_FORECAST = 0;
  private Nexus nexus;
  private boolean done = false;
  private Weather w;

  public AttachReturn attach_approval(Object userObject, String url)
  {
    Endpoint ep = nexus.endpoint_init(nexus.endpointattr_init(this));
    ep.set_user_object(userObject);
    Startpoint sp = nexus.startpoint_bind(ep);
    return new AttachReturn(0, sp);
  }

  public void invoke_handler(Endpoint ep, GetBuffer inbuf, int handlerId)
  {
    switch(handlerId) {
    case GET_FORECAST:
      ((Weather)ep.get_user_object()).getForecast(inbuf);
      break;
    }
  }

  public void run(String[] Args)
  {
    nexus = new Nexus();
    Args = nexus.init(Args, "nx", null);
    int port = nexus.allow_attach(2099, this, new Weather(nexus));
  }

  public static void main(String[] Args)
  {
    (new Server()).run(Args);
  }
}
```

Figure 4: Example Nexus server

```
import nexus.*;

public class Client implements HandlerInterface
{
  private final int REPLY = 0;
  private final int GET_FORECAST = 0;

  private Nexus nexus;
  private String forecast;
  private boolean reply_recd = false;

  public synchronized void handleReply(GetBuffer inbuf)
  {
    try {
      char[] fc = new char[inbuf.get_int()];
      inbuf.get_char(fc, 0, fc.length);
      forecast = new String(fc);
      reply_recd = true;
      notifyAll();
    } catch(Exception e) { }
  }

  public void invoke_handler(Endpoint ep, GetBuffer inbuf, int handlerId)
  {
    switch(handlerId) {
    case REPLY:
      handleReply(inbuf);
      break;
    }
  }

  private synchronized void waitForReply()
  {
    try {
      while(!reply_recd) {
        wait();
      }
    } catch(Exception e) { }
  }

  public void run(String[] Args)
  {
    try
    {
      nexus = new Nexus();
      Args = nexus.init(Args, "nx", null);
      Endpoint ep = nexus.endpoint_init(nexus.endpointattr_init(this));
      Startpoint mysp = nexus.startpoint_bind(ep);
      Startpoint sp =
                nexus.attach(new java.net.URL("", "rainier.extreme.indiana.edu", 2099, "")).sp;
      PutBuffer outbuf = nexus.buffer_init(mysp.sizeof(), 0);
      mysp.put(outbuf);
      nexus.send_rsr(outbuf, sp, GET_FORECAST, false);
      waitForReply();
      System.out.println(forecast);
    } catch(Exception e) { }
  }

  public static void main(String[] Args)
  {
    (new Client()).run(Args);
  }
}
```

Figure 5: Example Nexus client

```
public class Communication
{
  public static Nexus getNexus()
  public static Startpoint getStartpoint(URL url) throws nexusrmi.UnknownHostException
  public static Startpoint getStartpoint(HandlerInterface h, Object o)
}
```

Figure 6: The communication class

```
public interface Nexusable
{
  public void resetNexusIndex();
  public void nexusWrite(PutBuffer NexusBuffer, Vector SerTab) throws BufferOverrunException;
  public Object nexusRead(GetBuffer NexusBuffer, Vector SerTab) throws BufferOverrunException,
                                                                    LostPrecisionException;
  public int nexusSizeof(Vector SerTab);
}
```

Figure 7: The Nexusable interface

implementation. Exception handling in our initial prototype, however, differs somewhat from the way in which exceptions are handled in the original RMI version.

The rest of the classes in our implementation can be divided into classes that implement the client-server communication and classes that implement the registry. The distributed garbage collector has not been implemented in our initial prototype yet. Furthermore, we added some classes that support our implementation by providing some common low level routines.

Only a subset of the original RMI classes are implemented in out initial prototype. Some classes are provided to keep the interface consistent, but have no functionality yet. In this section we describe the classes that we have implemented.

3.1 Support classes

3.1.1 The `Communication` class

The methods that the `Communication` class implements are shown in Figure 6. In addition a static initializer is provided to create and initialize a `Nexus` object. The `Nexus` object can be obtained by invoking `getNexus()`.

The other two routines both create and return `Startpoint` objects. The first attaches to the given url and returns a `Startpoint` for this connection. The second creates a local `Endpoint` and returns a `Startpoint` connected to it.

3.1.2 The `Nexusable` interface

The `Nexusable` interface is shown in Figure 7. The interface and its methods are automatically added to any object that is declared `Serializable` by the **nexusrmis** compiler. The routines implement object serialization and are described in Section 5.

3.1.3 The `Nexusize` class

The `Nexusize` class contains methods used in object serialization. It contains read, write and sizeof methods for array objects as well Strings. Since NexusJava does not provide support for boolean primitive types, methods are supplied to convert these to byte values.

The `nexusSizeofObject()` method returns the size of any reference, either array or generic object. For generic object, this method invokes the `nexusSizeof()` method from the object. This method assumes complete type information to be sent over the wire. Similarly, `nexusWriteObject()` and `nexusReadObject()` write and read references, including complete type information. These methods are invoked in case the type of the actual object to be transferred cannot be inferred at compile time. If complete type information is available at compile time, the compiler serializes objects directly.

Strings are the only Java system classes that can be handled in our NexusRMI. Methods to serialize Strings are also included in this class. Strings can also be read and written with the `nexusReadObject()` and `nexusWriteObject()` methods, allowing a String to be passed to a formal parameter of a superclass.

3.2 Server classes

Most of the classes for the RMI server are provided in our implementation, although only a few are actually implemented. Most of the functionality provided in the other classes is, however, seldomly used, so leaving these out poses no restrictions on most applications. This section only describes those classes that are implemented.

3.2.1 The `RMISecurityManager` class

An implementation of the `RMISecurityManager` class is provided which currently poses no restrictions at all.

3.2.2 The `Remote` interface

This interface is copied and not modified from the standard RMI implementation.

3.2.3 The `RemoteStub` class

The `RemoteStub` class is the superclass of all stub objects, including the `RegistryStub`. This class provides a table to keep track of currently executing remote method invocations. Multiple remote invocations can occur when two threads invoke a remote method concurrently. The table stores the current status of the invocation and the result of the invocation if available. The status can either be BUSY, which means a remote invocation is in progress and no reply is available yet, or READY, which means a reply is available but not yet handled by the stub, or FREE which means the current entry is available for new remote method invocation. The result can either be an object to be returned by the remote method or an exception thrown in the remote method.

To invoke a method on a remote object, the stub must first obtain a free entry from the table by invoking `getFreeEntry()`. This method blocks until a free entry is available.

When a free entry is obtained, a handler that receives the reply must be created, provided with the index of the obtained table entry. The handler object is described later in this section.

To enable the remote object to reply to this handler a startpoint has to be obtained for it using the `getStartpoint()` method of the `Communication` class, providing the handler object. No user object needs to be supplied, since it is statically known what object handles the incoming replies.

The next step is to create and fill a buffer and send it with a handler identifier to the remote object. The `waitForReply()` method is than invoked with the index of the table entry. This method waits until the status of the remote invocation indicated by index becomes READY, indicating the arrival of a reply message. If a reply has arrived, the stub checks whether it is a result or an exception, after which the table entry is cleaned up and the exception is thrown or the result is returned.

The handler object, used to handle replies, is declared to implement the `HandlerInterface` and thus provides a `invoke_handler()` method. When a reply arrives for this handler, either the reply handler corresponding to the remote invocation or the EXCEPTION_HANDLER is executed. The former sets the result object using `setResult()`, supplying the index of the current method invocation and the result object. The latter creates the appropriate exception from the received string representation by using routines from the reflection classes. The exception is returned to the stub using `setException()`, again providing the index and the exception object. Both the `setResult()` and `setException()` method signal the waiting thread.

3.2.4 The `UnicastRemoteObject` class

The `UnicastRemoteObject` class maintains a table of stub information for every stub created on this JVM. This information contains the startpoint to the corresponding skeleton and the fully qualified name of the stub object. As described in Section 5, this information is sent when transferring a remote reference, instead of a complete stub object.

The `exportObject()` method creates a stub and skeleton for the given remote object. The constructor of the `UnicastRemoteObject` class invokes this method to automatically export remote objects that are an instance of a subclass of the `UnicastRemoteObject` class. The `createStub()` method creates a stub for the given remote object name and startpoint and enters the relevant information in the stub table if not already present. The `createSkeleton()` method creates a skeleton for the given remote object. The invocation of `getStartpoint()` creates an endpoint for which the skeleton will handle incoming messages. The actual remote object will be associated with the endpoint as userobject.

Both the `createStub()` and `createSkeleton()` use the reflection classes to create an instance of the correct Stub and Skeleton class, since the actual type of stub and skeleton is not know to the `UnicastRemoteObject` class at its compile time. The `getStartpoint()` returns a startpoint to a previously created stub for the given remote object, using the stub table.

3.3 Registry classes

3.3.1 The `Registry` interface

The `Registry` interface, shown in Figure 8 contains the remote methods of the registry. It is implemented by the `RegistryStub` class, which serves as a stub for the registry. Note that both RMI servers and RMI clients act as a client to the remote registry.

```
public interface Registry extends Remote
{
  public static final int REGISTRY_PORT = 1099;

  public Remote lookup(String name) throws RemoteException, NotBoundException, AccessException;
  public void bind(String name, Remote obj) throws RemoteException, AlreadyBoundException,
                                                                        AccessException;
  public void unbind(String name) throws RemoteException, NotBoundException, AccessException;
  public void rebind(String name, Remote obj) throws RemoteException, AccessException;
  public String[] list() throws RemoteException, AccessException;
}
```

Figure 8: The `Registry` interface

3.3.2 The `RMIRegistry` class

The registry itself is implemented in the class `RMIRegistry`. Constructors are provided that allow the registry to run as a stand alone Java process or as a daemon thread created by a Java process. After initialization, the registry allows other processes, both RMI servers and clients, to attach as described in Section 2.2.

The registry maintains a table of remote objects, containing the name of the object as specified in the file-name field of the url supplied by the clients, a `Startpoint` to the skeleton of the remote object and the fully qualified name of the class of the remote object. The latter is needed to create the stub object dynamically at the client, since it has no notion of the actual class name. Recall that the type of a remote reference is the name of the remote interface, which is not required to be related to the name of the remote object implementing it.

The main part of the registry is the dispatching routine `invoke_handler`, which is passed a handler identifier, indicating which operation is to be performed, a databuffer and an endpoint. This method performs the desired routine and returns the result, if there is one, to the client. If an operation could not be performed by the registry, an exception message, containing the fully qualified name of the exception to be thrown at the client side, is returned to the client.

The `bind` method enters the object name, startpoint and class name supplied in the buffer into the table. If an object with the given object name already exists in the table, an `AlreadyBound` exception is returned to the client. The `rebind` method does the same, but does not return an exception if an object with the given object name is already present in the table. Instead it replaces this entry with the new one. The `unbind` method removes the entry with the given object name, to the client returning a `NotBoundException` if no object with the given object name is present in the table. The `lookup` method returns the entry with the given object name to the client, returning a `NotBoundException` if no object with the given object name exists. The `list` method returns a list of all object names currently stored in the table.

3.3.3 The `RegistryStub` class

The `RegistryStub` class is declared to implement the `Registry` and `RegistryHandler` interfaces. This class acts as a stub for the client to be used when invoking methods on the remote registry and thus extends the `RemoteStub` class. When an instance of this class is created, a startpoint for the registry is obtained by invoking `getStartpoint()` from the `Communication` class supplying the url of the registry.

All registry methods send a startpoint for their handler as first item. The `lookup()` method sends an additional string, containing the url for the registry to look up. A startpoint for the skeleton of the required object and the fully qualified name of the corresponding stub object is returned by the registry. These objects are passed as parameters to the `createStub()` method of the `UnicastRemoteObject` class, which creates a stub for the remote object. The `bind()` method sends the startpoint of a skeleton for a remote object, together with the fully qualified name of the remote object and the url under which the object has to be registered. The reply to this method is an empty message. The `rebind()` method sends and receives the same information as the `bind()` method. The `unbind()` method sends the url of the remote object to be unbound and also receives an empty message. Finally, the `list()` method sends no information besides the startpoint to which the reply has to be sent. It receives an array of strings containing the names of all remote objects currently bound in the registry.

If an exception is received from the registry, the actual exception is created from the name of the exception contained in the reply using the methods from the reflection classes. These exception objects can than be thrown to the client.

All methods use the routines from the `RemoteStub` class to communicate and synchronize with their corresponding handlers as described in Section 3.2.3.

3.3.4 The `LocateRegistry` class

The `LocateRegistry` class is used to obtain an object of the `RegistryImpl` class, which acts as a stub for communicating with the registry. The `getRegistry()` methods all return a stub for an existing registry. The `createRegistry()` method first creates and starts a new registry running as daemon thread and than returns a `RegistryImpl` object for it.

3.3.5 The `Naming` class

The `Naming` class provides methods that RMI clients and servers can use to register and query remote objects in the registry. The actual work is done by the `RegistyStub` class, to which a reference is obtained by the methods from the `LocateRegistry` class. The supplied url for the registry must contain "`rmi`" as protocol, which is replaced within the `Naming` class since it is not an official protocol.

4 Stub and Skeleton generation

As in the original RMI implementation, we provide a stub compiler, which generates stubs and skeletons for remote objects. Our first approach was to use the source code for a remote object and create a stub and skeleton using only the information offered by the source code . We found, however, that the source code for the remote object only did not supply us with enough information. We will elaborate on this further in Section 5 when we explain our approach to object serialization, which is where the main difficulties arise. Because of these problems, we decided to construct our stubs and skeletons using the bytecode representation of the remote object.

In this section we will describe how stubs and skeletons are implemented in our RMI implementation. We will first describe the implementation of the stub and skeleton and than briefly describe the implementation of our stub/skeleton compiler. Although stub and skeleton also contain code for marshalling and unmarshalling, we will not describe it in this section, but rather defer it to the next section.

4.1 Stubs

The name of a stub object is derived from the name of the remote object by appending "`Stub`" to it. All stubs are declared to be a subclass of the `RemoteStub` class. The stub implements the remote interface belonging to the remote object which means that every method that can be invoked on the remote object can also be invoked on the stub object. Associated with each method from this interface is a unique handler identifier.

Associated with each stub is a handler object, which receives the reply message from the skeleton. The `invoke_handler()` method mainly consist of a switch statement with a label for every possible handler identifier and an additional one for the exception handler.

The implementation of each of the methods from the remote interface in the stub object consists of sending a message containing the following items (to the left of the colon is the value written and on the right side is its type):

- `(sp:Startpoint)`, the startpoint to the handler object to which the reply is to be sent.
- `(ao:int)`, an array offset, which for now is always zero. This information will become useful when communicating with HPC++ objects.
- `(handlerid:int)`, the handler identifier corresponding to the method to be invoked on the remote object.
- The value of every parameter given to the remote method, according to the format described in Section 5.

The reply consists of the handler identifier of the method that was invoked, followed by the object returned by the method. If the remote method threw an exception, the EXCEPTION_HANDLER_ID is received followed by the following items:

- `(ex:byte)`, a number, identifying the kind of exception that has occurred and has to be thrown to the client. A `0` denotes that the method body threw a regular exception, which is thrown to the client directly. A `1` indicates an `Error` was thrown in the method body, which is encapsulated in a `ServerError` at the client side. A `2` indicates a `RemoteException` has been thrown in the method body during a remote method invocation, which is encapsulated in a `ServerException` at the client side. A `3` indicates a `RuntimeException` has been thrown in the method body, which is encapsulated in a `ServerRuntimeException` at the client side. A `5` denotes any exception occurring in the stub or skeleton, which will be encapsulated in a `UnknownException` at the client side.
- `(package:String)`, the package name of the exception thrown at the server side.
- `(exname:String)`, the name of the exception thrown at the server side.

The reflection classes are used to create the exception to be thrown at the client from the information supplied in the reply. Sending a message by the stub and handling the reply message is described in Section 3.2.3, where we described the `RemoteStub` class.

4.2 Skeletons

The name of a skeleton is derived by appending "Skel" to the name of the remote object. A skeleton is declared to implement the HandlerInterface which allows it to handle incoming messages. The invoke_handler() method starts a new thread that actually handles the incoming message. Creating a new thread of control prevents a deadlock in situations where two methods of the remote object need to be handled concurrently, for instance if execution of one method waits for a signal from another method.

The handler thread first extracts the startpoint, array offset and handler identifier from the incoming message. Based on the handler identifier the appropriate handler routine is executed. This routine extracts the parameters for the remote method from the message and invokes the method on the object associated with the endpoint for the incoming message, which is the target remote object. The result is than put in a reply message, which is returned to the client using the startpoint from the incoming message.

If an exception occurred when handling an incoming message the exception handler of the handler object is executed. This handler determines the exception to be returned to the client from the thrown exception and the status variable. The latter keeps track whether a BufferOverrunException was thrown during unmarshalling (status = 0) or marshalling (status = 1). The three strings as described in the previous section are than returned to the client.

4.3 Compiler implementation

We now briefly describe the implementation of **javas**, which is our stub/skeleton compiler. **Javas** creates source code for the stub and skeleton object of a remote object by analyzing the bytecode representation of the remote object class.

The read_classes() function reads the bytecode for the remote object and for every object directly or indirectly referenced from the remote object. It generates a data structure of class definitions. Iterators to traverse all fields, methods and interfaces of a class are supplied as well as some query routines.

Most of the code for stubs and skeletons is generated by writing fixed code into the stub and skeleton file. Where specific code is needed for each method of the remote object the traversal routine remote_method_traverse() is used, which calls a given function providing every method in turn as parameter. To iterate over the parameters of a method the function handle_parameters() is used, which calls handle_type() for every parameter in turn, supplying the given action specifier. A similar function handle_return_type is provided, which calls handle_type() for the return type. The handle_type() function dumps a specific piece of code, depending on the type and action supplied.

5 Object Serialization

Object serialization deals with sending the internal state of objects as a message to another JVM and with reconstructing that object from the received message. According to the Java RMI specification, The internal state of an object consists of the values of all fields of the object, except for transient and static fields.

The first problem when implementing object serialization is that `private` and `protected` fields are part of the internal state. Since other objects cannot access these fields, public methods have to be added to serializable objects which actually perform the actual serialization.

The second problem is that more information is needed in the compiler than is present in the source code. To be able to generate efficient code, we therefore used the bytecode representation of the classes.

The rest of this section describes the format in which different kinds of data are send over the network, ignoring the information added by Nexus and lower layers.

5.1 Serialization of primitive types

The `GetBuffer` and `PutBuffer` class from the NexusJava library provide methods to read and write values of primitive types from and to a Buffer, except for boolean types. For variables of a primitive type we just read and write the current value, where we convert boolean values to byte values (`true` = `1`, `false` = `0`). For primitive types, complete type information is always available at compile time, so no type information is included when sending primitive types.

5.2 Serialization of references

The generic format for object references is one of the following (again, the value written is on the left side of the colon, while the right side indicates its type):

1.
   - `(0:byte)`, that is a byte of value zero, for null references.
2.
   - `(1:byte)`, denoting this is not a null reference, followed by
   - `(0:byte)`, denoting that this is the first reference to this object in the current method invocation, followed by
   - The internal state of the object.
3.
   - `(1:byte)`, denoting this is not a null reference, followed by
   - `(1:byte)`, denoting that a reference to this object has been sent previously, followed by
   - `(index:int)`, an index in a table where a reference to this object already exists. We will see how this works shortly.

On the sending side, on every remote method invocation, a table is constructed containing the previously encountered objects to be sent over the wire. Whenever an object has to be packed into the buffer, a check is made to see whether the object is already in this table. If it is, this means that the object has previously been serialized. There are two reasons why we do not serialize such objects again. First of all, two references to one object should yield one object with two references to it at the receiving side. The second reason is that objects contained within other objects are serialized recursively. In the case of circular data structures, this would cause object serialization to loop forever. Thus, instead of writing the internal state again, we send the index of the object in the table. Since buffers in NexusJava have a first-in-first-out property for for their contents, the exact same table can be reconstructed at the receiving side, by inserting an object in the table, whenever they are reconstructed from the received internal

```
// Writing an object
if(o == null)
  buf.put_byte((byte)0);
else {
  buf.put_byte((byte)1);
  if(SerTab.contains(o)) {
    buf.put_byte((byte)1);
    buf.put_int(SerTab.indexOf(o));
  }
  else {
    buf.put_byte((byte)0);
    SerTab.addElement(o);
    // Write internal state

}
```

```
// Reading an object
if(buf.get_byte() == 0)
  return null;
else {

  if(buf.get_byte() == 1) {

    return SerTab.elementAt(buf.get_int());
  }
  else {
    Object res = new ...;
    SerTab.addElement(res);
    // Read internal state and add to table
    return res;
}
```

Figure 9: Reference serialization

state. Whenever an index is received, the reference can be set to the table entry at the index received. Figure 9 shows how this protocol is implemented on the sending and receiving sides. Note, that when reading a regular object from a stream, the reference has to be stored after allocating the object, but before reading the internal state, since the internal state may have a reference to itself, which must not be serialized again.

The internal state of an object depends on how much information the compiler has when generating object serialization code. If the compiler encounters a class name, it cannot tell whether an instance of this class or an instance of a subclass will be assigned to it, unless it is a final class. This holds for fields as well as parameters to remote methods. Thus, complete type information is needed when sending references to objects, in order to reconstruct an object of the correct type at the receiving side. Also note, that arrays are assignable to references of type Object.

To send the internal state of any reference, the `nexusSizeofObject()` and `nexusWriteObject()` methods are used. The former calculates the total buffersize needed for the whole object, the latter actually writes the object. The `nexusReadObject()` method is used to reconstruct the object from an incoming buffer. The internal state of object references is one of the following:

1.
   - `(0:byte)`, denotes a remote reference, followed by
   - `(stubname:String)`, the classname of the stub associated with the remote reference, followed by
   - `(sp:Startpoint)`, the startpoint to the skeleton associated with the remote reference.
2.
   - `(1:byte)`, denotes a String reference, followed by
   - `(s:String)`, the contents of the String (in the format dictated by the String methods from the `Nexusize` class.
3.
   - `(2:byte)`, denote a regular object, followed by
   - `(cn:  String)`, the actual classname of the object to be transferred., followed by
   - The values of each field recursively.
4.
   - `(3|4|5|6|7|8|9|10:byte)`, denotes an array of booleans, bytes, shorts, integers, longs, floats, doubles, or characters respectively, followed by

   - The primitive array in the format described shortly.

5. - `(11:byte)`, denotes an array of references, followed by
   - `(ct:String)`, the type of the components of the array, followed by
   - `(l:int)`, the length of the array.
   - Each element of the array recursively.

NexusJava provides special methods to write arrays of primitive types. We use these methods to write primitive arrays in the following format:

- `(l:int)`, the number of elements of the array, followed by
- The contents of the array according to the NexusJava format.

If a class is declared final we do not send type information for it, since these classes cannot be subclassed. In this case, the compiler knows that an instance of that object will be given as actual argument and can thus generate type information. The same holds if a variable is declared to be of type array. Note, however, that types contained in arrays or final classes may still have to be handled dynamically.

5.3 Compiler implementation

The original **javar** compiler [3] was developed to experiment with restructuring transformations on Java Programs [2]. Although it does not provide a full front end (little semantic analysis is included), it is capable of parsing a Java source file and building a syntax tree representation from it. Furthermore, it provides functions to traverse over the statements and expressions in the syntax tree. A great number of functions are provided to create additional subtrees, which can than be added to the tree or replace parts of the original tree.

We now briefly indicate how we extended the **javar** compiler to add serialization methods to classes. After having read a Java source file, we traverse all classes in the syntax tree. If a class is found, we load the corresponding bytecode representation of this class and all classes directly or indirectly referenced from it using `readclasses()`.

The methods and fields we add are first constructed using the various `make_`- functions from the syntax tree, which are than added to the original tree. The methods are constructed by iterating over all fields of the class, using the bytecode representation, in a similar way as it is done in **javas**.

Every field is passed to `handle_type_expr()` which produces the appropriate code for the given field as described earlier in this section.

6 Conclusions

We have presented a design of the Java RMI API on top of NexusJava. We have shown that it is possible to implement most of the RMI semantics without any special runtime support besides what is present in the Java Core API. We have also shown that although a lot can be done by considering just Java source code, cleaner and more efficient code can be generated when using bytecode.

Acknowledgments

We would like to thank Aart J.C. Bik who is the author of the **javar** compiler, which is we used as a basis for our **nexusrmis** compiler, and who is also the author of the various routines that read bytecode representations, used in both **nexusrmis** and **nexusrmic**.

A Nexus RMI manual

This appendix describes how a Java RMI program can be made to run on top of the NexusJava communication library

The first step is to determine whether remote interfaces involved in the application do not send objects belonging to classes of which no source code is available. The next step is to determine whether the application uses only those RMI classes and methods that are fully supported in our implementation. If these two restrictions are met we can start adapting the source code.

The only thing that needs to be changed in the source code are all references to classes and interfaces from the original `java.rmi` package. These references now have to use the corresponding classes in the `nexusrmi` package instead. This typically involves replacing every occurrence of `java.rmi` with `nexusrmi`.

The next step is to compile the complete source code using the standard Java compiler **javac**.

Now it is time to add serialization methods to those objects that occur as parameter or return value in a remote method invocation. Although only these classes need the serialization methods no harm is done when the serialization methods are also added to other classes. The serialization methods can be added by calling `nexusrmis` on the source files of the objects. This command will make a call to **javar** to add the methods and will than recompile the changed files.

The last step involves the construction of a stub/skeleton pair for every remote object in the application. This can be done by calling **nexusrmic** on the bytecode representation of every remote object. This command will make a call to **javas**, which will generate source code for the stubs and skeletons, and than compile these stubs and skeletons.

The external registry, if needed, can be started by running the `RMIRegistry` class from the `nexusrmi.registry` package.

Example 1 *Consider the following example, where class **TestObj** is sent as parameter and class **ServerImpl** is the implementation of a remote object.*

```
rainier> pwd
~/java
rainier> ls
Client.java          ServerImpl.java      Server.java          TestObj.java
```

We now compile this application to run with our RMI implementation (the code has already been adapted to use our implementation.

```
rainier> javac *.java
rainier> nexusrmis TestObj.java
    javar -ox TestObj.java
    javac TestObj.java
rainier> nexusrmic ServerImpl
    javas rmitest.ServerImpl
    javac *Stub.java *Skel.java
```

*Client and server are started on **school** and **rainier** respectively:*

```
rainier> java rmitest.ServerImpl
school> java rmitest.Client
```

References

[1] P. Beckman, D. Gannon, and E. Johnson. Portable Parallel Programming in HPC++. 1996.

[2] A.J.C. Bik and D.B. Gannon. Automatically Exploiting Implicit Parallelism in Java. *Concurrency, Practice and Experience*, 9(6), 1997.

[3] A.J.C. Bik, J.E. Villacis, and D.B. Gannon. *JAVAR manual.* Computer Science Department, Indiana University, 1997. This manual and the complete source of `javar` is made available at `http://www.extreme.indiana.edu/hpjava/`.

[4] K.M. Chandy. Caltech Infospheres Project Overview: Information Infrastructures for Task Forces. Technical Report Computer Science 256-80, California Institute of Technology, nov 1996.

[5] I. Foster, J. Geisler, C. Kesselman, and S. Tuecke. Managing multiple communication methods in high-performance networked computing systems. *Journal of parallel and Distributed Computing*, page to appear, 1997.

[6] I. Foster, G.K. Thiruvathukal, and S. Tuecke. Technologies for ubiquitous supercomputing: a Java interface to the Nexus communication system. *Concurrency: Practice and Experience*, 9(6):465–475, jun 1997.

[7] J. Gosling, B. Joy, and G. Steele. *The Java Language Specification.* The Java Series. Addison-Wesley Developers Press, 1996.

[8] Object Management Group. The Common Object Request Broker: Architecture and Specification, jul 1995.

[9] Objectspace. Objectspace Voyager Core Package Technical Overview, 1997.

[10] Sun Microsystems. *Java Remote Method Invocation Specification*, feb 1997.