Bachelor thesis
Computer Science

Radboud University

Introduction to YARMIS, a dynamic alternative to RMI

Author:
Maurice Knoop
s4122461

First supervisor/assessor:
Dr. Sjaak Smetsers
s.smetsers@cs.ru.nl

Second assessor:
Dr. Pieter Koopman
pieter@cs.ru.nl

January 13, 2015

Abstract

In this thesis we will introduce a new system called YARMIS. YARMIS allows methods to be invoked on java objects that are stored on a Host. This system has been developed to allow for dynamic Host - Client connection where every device can easily switch between being a host or a client. We will compare this new system to the existing RMI system, that also allows for methods to be invoked on remote objects. The systems will be compared based on execution time, implementation and limitations. Finally we will discuss how aspects of YARMIS can be improved to have the advantages of RMI like faster execution time, without losing the flexibility that YARMIS offers.

Contents

Chapter 1

Introduction

This system needed to be able to provide functionality locally, but also let the same functionality be invoked on a different machine. This to let one application become a remote control to another instance of the same application, without cluttering the code with calls required for communication. So YARMIS was developed to provide this functionality. YARMIS makes it possible to switch between performing functionality locally or remotely. RMI is an existing system that provides similar functionality. To decide whether YARMIS has any added value over RMI is is required that these systems are compared.

Chapter 2

Preliminaries

Within this Thesis, methods and classes that are part of specific systems such as YARMIS are displayed as *ClassOfInstance.method(...)*. Methods and classes that are part of the JDK[1] will be displayed as *ClassOfInstance.method(...)*.

2.1 Reflection

Java provides access to meta data of classes. Fields and methods of a class can be obtained at runtime. Within Object Oriented Programming, this is known as type introspection. Fields and methods can be obtained as instances of *Field* and *Method* respectively. These classes provide access to properties such as access modifiers and *Annotations*.

Through *Field* and *Method* Java also allows for modifications to these instances at runtime. This is known as *reflection*. This allows for more dynamic interaction with classes and instances.For example, one can dynamically look up and invoke methods at runtime.

2.2 Distributed Object Applications

Distributed Object Applications (DOA) are applications that allow the invocation of methods on objects that are on a different machine than the calling objects. This is achieved through communication between the two machines. Invoking methods that are not on the same machine is also known as Remote Procedure Calls (RPC). For this thesis we will be looking only at DOA's that can be set up between two Java Virtual Machines.

Multiple systems exist for invoking methods on remote objects. Three well known systems are SOAP[2], CORBA[3] and RMI[4].

[1] excluding RMI
[2] Simple Object Access Protocol
[3] Common Object Request Broker Architecture
[4] Remote Method Invocation

SOAP is a standard developed by Microsoft. It allows for implementations across different programming languages. This is achieved by using XML to represent data and objects.

SOAP makes it possible to send the data contained in objects from one machine to another. It is also possible to use SOAP to The invocation of methods on remote objects could then be done by first obtaining a local version of such a object, invoke methods on that object, and then notify the server of any side effects of method calls. SOAP does not however specify how this should be done, only how the data should be passed. In effect this means that to use SOAP for method invocation on remote objects, we still need to define a new system to send those messages. Therefore SOAP itself is not of any interest.

CORBA[2] was a popular standard developed by the Object Management Group (OMG) in the nineties[6]. It allowed for communication between servers, regardless of the language they were implemented in. This was achieved by making use of definitions written following the syntax of Interface Definition Language(IDL) [1]. The usage of IDL requires a compiler that turns an IDL file into (amongst others) a so-called stub and skeleton. I have not been able to obtain such a compiler. As a result I am unable to use CORBA for this Thesis.

RMI[3] is developed by Oracle. It is written specifically for Java, and has been available since JDK 1.1[5]. The standard version of RMI can only work with Java. A variant of RMI, called RMI-IIOP, uses the standard set by CORBA to communicate. This allows it to communicate with other CORBA compliant systems.

2.3 Test application

We explain and test both RMI and YARMIS using a simple application. The functionality of this program is to let a Hosting machine append a string that is provided by the client machine to the string "Hello". The variable itself will be the string "world" that is passed from the client to the host. During the execution of the application, the client will tell the host to perform the append step, after which the client will obtain the result. Both versions of the application will contain the same methods. The application consists of 2 classes, each with 2 methods. The classes are called *Host* and *Client*. As these run on different machines, they both need to implement the *main* method. In this method any required preparation is performed.

The other predefined method of Host is called *execute*. It's implementation is fixed across all systems and will return "Hello " followed by the given argument. For simplicity, the concatenation is done using the + operator rather than a StringBuilder.

[5]http://docs.oracle.com/javase/7/docs/api/java/rmi/Remote.html

```
public class Host {


  public String execute(String argument)
  {
     return "Hello "+argument;
  }

  public static void main(String[] args)
  {
     Host host = new Host();
  }

}
```

Figure 2.1: The code for the Host as defined in the framework

The second method of Client is called *perform*. Its implementation is open to the different systems, but should contain a call to the *execute* method of Host, as well as a means of storing the value returned by that call, and printing it using the system's OutputStream, available under *System.out.println*. The basics of this operation are given in the framework for this example. This method is called by the main method of Client.

While *perform* is called by the main method of Client, the invocation of *execute* is done by the client through each of the various systems. The result of the invocation of *execute* should be stored in the String result.

The *perform* method of Client is made static as no instance is required to perform operations. This differs from the *execute* method of Host, as all systems require the remote object to be an instance.

```
public class Client {

    private static final String IP_HOST = "192.168.1.1";
    private static final String MESSAGE = "world";

  private static void perform()
  {
     String result = null;
     System.out.println(result);
  }

  public static void main(String[] args)
  {
     perform();
  }

}
```

Figure 2.2: The code for the Client as defined in the framework

2.4 Java RMI System

Host

Clients can retrieve objects that implement the interface *Remote* from the server and invoke methods on them. This interface doesn't require the implementation of any other methods. It is used simply to identify classes whose methods can be called remotely. For our example program this adds the requirement that Host needs to implement *Remote*. It is also required to implement the *Serializable* interface, to allow an instance of *Host* (and thus also an implementation of *Remote*), to be send over a connection.

```
import java.rmi.Remote;

public class Host implements Remote, Serializable { ... }
```

Figure 2.3: The new header for the Host

To allow access to classes that implement *Remote* so called *Registries* are used. In *Registries* objects are stored under a name (*String*). All clients who know that name can use the *Registry* to look up the object. To implement this we need to have two things: A name for the object we're going to register and a *Registry* to register it with. It is not important which name is assigned to a registered object, as long as that name is used consistently for that object.

To creation of a *Registry* is simple. We need to call *LocateRegistry.createRegistry(int port)*, where port refers to the port[6] that should be used. The returned *Registry* can then be used to register the object. This process is called *binding* and can be initiated by calling *bind(...)* on the *Registry*. Both *lookup(...)* and *bind(...)* can throw a *RemoteException*. The latter can also throw an *AlreadyBoundException*. These exceptions need to be caught.

The resulting code for Host then becomes:

```
public class Host implements Remote, Serializable {

  public static final String NAME = "name";

  public static void main(String[] args)
  {
     Host host = new Host();
     try {
        Registry registry = LocateRegistry.createRegistry();
        registry.bind(NAME, host);

     } catch (RemoteException | AlreadyBoundException e) {
        e.printStackTrace();
     }

  }

}
```

Figure 2.4: The code for the implementation of the Host's main method using RMI

When a registered object is no longer referenced (locally or remotely), then it is garbage collected. To prevent the registered object from being discarded as soon as *main* finishes, you need to keep a reference to it. This can be done by storing the registered object in a static field. This will keep the registered object available while the process the host is running in is still running. It is also possible to let the *main* method sleep for as long as is required to let a client obtain a reference to the object. This requires that the object is still referenced before sleeping. Here we'll prevent the registered object from being discarded by sleeping for a long time in the call to *main*.

Client

Two options exist for accessing objects that are stored on a server. The first option is by using *Registry*. This provides access to objects on a single

[6]The default port is 1099, accessible as *Registry.PORT*

server. The second option is to use a class called *Naming* containing mostly static methods. It can be used to connect to multiple servers. It is explained later on.

An existing registry can be obtained by calling *getRegistry(...)* on *LocateRegistry*. This will result in a Registry that was created on the host using *createRegistry(...)*. To connect to a different computer we need to provide an IP address. The server doesn't need to be on the local network. The port is assumed to be 1099 unless specified otherwise. Note that the *Registry* that the Client obtains is a reference to the *Registry* on the Host. The Client doesn't obtain the actual *Registry*. To allow for this, *Registry* needs to implement *Remote*.

By calling the method *lookup(String name)* on a Registry objects from a server can be retrieved. Calling this method might result in a *RemoteException* or a *NotBoundException*. The former is a generic exception that is thrown whenever something goes wrong while accessing a remote object. The latter is to indicate that there is no object registered on the Host under the provided name.

```
public class Client {

  private static final String IP_HOST = "192.168.1.1";
    private static final String MESSAGE = "world";

    private static Registry host;

  private static void perform()
  {
     Host hostObj;

     try {
        hostObj = (Host)host.lookup(Host.NAME);
     } catch (RemoteException | NotBoundException e) {
        e.printStackTrace();
        return;
     }

     String result = hostObj.execute(MESSAGE);
     System.out.println(result);
  }

  public static void main(String[] args)
  {
     try {
        host = LocateRegistry.getRegistry(IP_HOST);
     } catch (RemoteException e) {
        e.printStackTrace();
        return;
     }
     perform();
  }

}
```

Figure 2.5: The full code for the implementation of a Client using RMI

As mentioned previously, Naming exists for connecting to multiple servers. Naming holds a number of Registries, thus allowing multiple connections, to different servers. It defines mostly the same methods as a regular Registry and forwards the calls for *lookup(...)*, *bind(...)* and other methods to the correct internal Registry. To be able to forward the call to the correct Registry, it is required to provide an IP address every time any of the methods is called. This is done through creating a URL that is formed like *//[ip address]/[remote name]*[7], where *[remote name]* is the name under which the requested object is stored. In the line of the previous examples, a valid URL would be *//192.168.1.1/name*.

[7]http://docs.oracle.com/javase/7/docs/api/java/rmi/Naming.html

```
 private void usingLocateRegistry() throws
     RemoteException, NotBoundException
{
   Registry host = LocateRegistry.getRegistry(IP_HOST);
   Host hostObj = (Host)host.lookup(Host.NAME);
}

private void usingNaming() throws
    MalformedURLException, RemoteException, NotBoundException
{
   Host hostObj = (Host) Naming.lookup("//"+Client.IP_HOST+"/
       "+Host.NAME);
}
```

Figure 2.6: These two methods provide the same instance for hostObj. Note that usingNaming can also throw a *MalformedURLException*.

RMI also allows for class definitions to be send to a server, allowing the server to work with classes that weren't previously defined on the server. This is called Dynamic Code Loading. It requires a *SecurityManager* to be set and installed. This will manage the access classes have on the host. If Dynamic Code Loading isn't used, the *SecurityManager* isn't required, as the developer will know what access clients will have through the defined classes.

Chapter 3

Implementation

YARMIS[1] (Yet Another Remote Method Invocation System) is a way of allowing one Java Virtual Machine (JVM) to invoke methods on a different JVM. It was created by Moritz Neikes and Maurice Knoop.

The application that YARMIS was designed for required that certain aspects of the application would be allowed to be executed remotely when connected to one Host, but be blocked when connected to a different Host, all depending on the settings of that Host. Furthermore, it should be possible within the application to switch between being a Host and being a Client. This means that sometimes functionality should be executed remotely (on a Host), and sometimes needs to be executed locally (when being a Host to other Clients, or when its not communicating.). So the system is required to be transparent and flexible.

YARMIS abstracts all communication such that invoking a method on a remote object is as easy as on a local object. Its key selling point is its ability to switch between local and remote execution when required. This allows the program to function as a remote control to a different machine that's running the same application, but also to provide that functionality by itself. It is also suited for making a more intuitive approach to working with a server. Rather than having to convert and reconvert every API call to use, for instance, HTML POST, you can simply invoke a method. Extending the functionality is as simple as defining a method in an abstract class and then implementing it.

Consider the example of a music application that uses YARMIS. Suppose device d_A is running that music application. The application is not communicating. Then if *play(...)* is called on d_A, d_A should start playing music. Now suppose that d_A is being a Host. It will still handle everything locally, but will now allow others to connect. We introduce device d_B to the system, letting the application on d_B becomes a client to d_A. Then calling *play(...)* on d_B should make d_A start playing music. YARMIS is used to

[1] http://www.github.com/knoop/YARMIS

handle everything in this example, aside from actually playing music.

3.1 Structure

To allow for the required flexibility, YARMIS uses a modular approach. Each *Module* provides specific functionality. In the context of the music application you can define a *Module* with music controls like play and pause, and a *Module* that contains a song library.

To allow switching between executing remotely and locally, it is important that Host and Client have the same definition of which methods can be invoked on a *Module*. This can be achieved by using interfaces. However as some basic interaction is required with *Modules*, abstract classes extending *Module* are used. These abstract classes are named *Functionality Definition Classes* (FDCs). These FDCs define the functionality provided by the application.

Transparency

The transparency requirement is fulfilled using FDCs. This is done by creating separate implementations for local and remote execution. The local implementation is implemented like you would with any other interface or abstract class. So, if an FDC requires *play(...)*, then the local implementation will implement the method to start playing. The remote implementation is used to let the call be executed on the Host. When a method is invoked, a request is send to the Host to also invoke that method. The result will then be reported back to the Client. This means that when *play(...)* is called on a remote implementation, it will call its Host, and let the Host perform *play(...)* on its local implementation. The UML diagram for *Module*, the FDC and the local and remote implementations is shown in Figure 3.1.

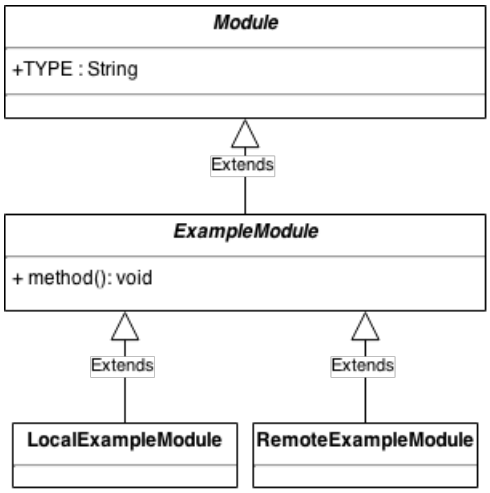


Figure 3.1: The UML diagram that shows how *Module* should be extended by an FDC(*ExampleModule*).

Accessing *Modules*

The usage of local and remote implementations by itself isn't fully transparent. The transparency can be increased by making it transparent how an instance of FDC (either local or remote) can be obtained. This is done through what is called *Core*. *Core* contains references to instances of *Modules* and makes these available.

Based on the fact that *Modules* provide functionality, it is not allowed to store more than one instance of a *Module* at the same time. Otherwise it is possible to have multiple *Modules* that are all providing the same functionality. Switching between local and remote execution is done by replacing the stored module.

It is not required that *Modules* are swapped often. A setup where one machine is always the Host, while others are always Clients is also possible. This can be achieved by never using the remote implementation on the Host.

The *Core* and *Module* structure is part of YARMIS. The actual definition of FDCs, as well as its local implementation defines the application itself and are up to the developer to be implemented[2].

3.2 Communication

The first problem about implementing remote method invocation is the communication between devices. How can you tell a device which method needs to be executed? What kind of protocol(s) should be used?

[2]The remote implementation is also part of the application, but the implementation is always the same, regardless of the method. See Figure 3.4 for that code.

Four things need to be known to be able to invoke a *Method* using reflection. The first is which *Class* should contain the *Method*. The second is the actual instance of that class to invoke the *Method* on. The third is the name of the *Method* to invoke. The fourth is an array of the type of all parameters. With this information the *Method* can be found. To invoke the method the parameters also need to be provided. These requirements follow directly from how methods should be obtained and invoked using reflection.

Because of the usage of *Modules*, the first two requirements are easily solvable. Only the *Module* type needs to be known to obtain the correct instance, of which the *Class* can then be obtained.

The third can not be derived as is the case with the first two. Therefore the method name has to be provided.

For the fourth requirement we can use the types of the parameters that have been passed.

Overall, the required components for the communication then become the name of the module, the name of the method and the instances of all arguments. The combination of these components is called a *Request*.

Connections

The connection between JVM's is made through *Connection*s. This is an abstract class that requires an *InputStream* and an *OutputStream*. *Connections* are capable of sending *Requests* through streams. This abstraction on the communication channel makes it possible to not only use (W)LAN/WiFi connections, but also Bluetooth or any other means of communication that uses in- and outputstreams.

All *Connections* are handled through the *Core*. Specifically through what we call the *ConnectivityPlugin*. It is available under *Core.connectivity*. It contains a *ConnectionManager* that keeps track of all active *Connections*. The *ConnectionManager* is the actual host, as it allows connections to be set up through a *ServerSocket*. The *ConnectivityPlugin* is not a *Module*. It can not have a remote implementation as it provides the *Connection* to a Host[3].

Messages

The requests sent over *Connection*s are packed in *Message*s. Messages are used to add some meta-data about the contained Request or other payload. Messages consist of the version code of the message, a type coding, the length of the payload, the actual payload and a Message Authentication Code (MAC). Their respective sizes can be fount in Table 3.1. The payload has a variable size. The actual size is indicated by the length of the payload

[3]Remote access to the *ConnectivityPlugin* can be achieved by defining a *Module* which interacts with the *ConnectivityPlugin*.

in bits. As there are 24 bits available to indicate the length of the payload, the payload size is limited to 16,777,215 bits, or 16MiB.

Version	Type	Payload length	Payload
4bits	4bits	24bits	variable

Table 3.1: The structure of a Message, with each length being indicated in bits.

Requests

As said earlier, *Requests* are used to contain all required information for a host to execute a method. They are sent by the client. They contain an identifier, the name of the intended Module, the name of the method to call, and all parameters for the given method along with the list of classes of those parameters. Requests are only created by *Remote* implementations. To reduce the odds that a *Request* is assigned the wrong values, we define *mkRequest(...)* in *Module* to generate the *Request* instead. The method that needs calling can be derived from the *StackTrace*. This is possible as the method that was called on *Remote* is also the method that should be called on *Local*. Furthermore, by implementing *mkRequest(...)* in *Module*, you can use the type stored to indicate the *Module*. The intended usage of sending a *Request* from a *Remote* implementation is shown in Figure 3.2.

```
public abstract class RemoteExampleModule
{
    protected void someMethod(String arg) {

        Core.connectivity.sendRequest(super.mkRequest(arg));

    }

    protected void someOtherMethod(String arg, int number) {

        Core.connectivity.sendRequest(super.mkRequest(arg,
            number));

    }

}
```

Figure 3.2: Using *mkRequest(...)* every method can be implemented roughly the same.

The offset of the method call previous to *mkRequest(...)* depends on the environment it is running on. Within the environment that offset will

never change. To make it work on every environment, the offset to find the correct *StackTraceElement* needs to be found. This is done with a one-time operation, that is performed within *mkRequest(...)*. If the index hasn't been set (equals -1) we search the obtained *StackTrace* for *mkRequest(...)*. The required index will be one greater than the index of *mkRequest(...)*. The implementation of *mkRequest(...)* is shown in 3.3

```
protected Request mkRequest(Serializable... args) {
  StackTraceElement[] stack = Thread.currentThread().
      getStackTrace();

  // Determine which index is of the previous call. This
      varies over different devices.
  if(STACKTRACE_MODULE_CALL_INDEX == -1){
    STACKTRACE_MODULE_CALL_INDEX = 0;
    while(!stack[STACKTRACE_MODULE_CALL_INDEX].
        getMethodName().equals("mkRequest")){
      ++STACKTRACE_MODULE_CALL_INDEX;
    }
    // We've found mkRequest, we want the one that sits
        underneath it.
    STACKTRACE_MODULE_CALL_INDEX++;
  }

  return new Request(type, stack[
      STACKTRACE_MODULE_CALL_INDEX].getMethodName(), args);
}

private static int STACKTRACE_MODULE_CALL_INDEX = -1;
```

Figure 3.3: The full implementation of *mkRequest(...)* and the required constant *STACKTRACE_MODULE_CALL_INDEX*.

Result

When a *Request* has been executed remotely, the result needs to be reported back. This also applies when the executed method is void, to indicate that the call has finished. An adaption of *Future* called *Result* is used to allow the Client to wait until the Host has performed the *Request*. This aspect works exactly like a *Future*. When *get(...)* is called on a *Result* instance, the current thread is locked and is released as soon as a response from the Host has been received. When the thread is released, the returned value is returned or the result is thrown as an *Exception* if the Host threw an *Exception*. With a *Result* it is not possible to cancel a request. It is however possible to use a callback rather than blocking the thread.

An example of the usage of *Result*s in combination with the remote implementation of a FDC is shown in Figure 3.4.

```
public class RemoteExampleModule extends ExampleModule
{

    public String getText() {
        try {
            return (String)Core.connectivity.sendRequest(
                super.mkRequest()).get();
        } catch(Exception e) { }

    }

    public void setText(String text) {
        try {
            Core.connectivity.sendRequest(super.mkRequest(
                text)).get();
        } catch(Exception e) { }

    }

}
```

Figure 3.4: Example remote implementation. This is the default way that methods should be implemented in a remote implementation.

To let *Results* work, it is important to know what response the *Result* is waiting for. This is achieved by using the identifier of a *Request*. When a *Request* is made, a *Result* object is returned. The *Result* will contain the identifier of the *Request*. When the host responds to a *Request*, it will provide the same identifier. This way the response can be linked to the *Request* and thus to *Result*. Then the *Result* can be released, and any threads waiting on the response can be notified.

3.3 Method invocation

When a *Request* is received by the Host, the Host needs to execute it. It uses reflection to obtain a *Method* using the information provided by a *Request*. This found *Method* can then be invoked using the arguments provided by the *Request*.

```
private Object performRequest(Request request) throws ...
{

    // The referenced Module, supplied by Core.
   Module module = (Module) Core.getModule(request.module);

   // Turn the given arguments into an array of classes.
   // This is required to find the correct method.
   Class<?>[] parameterTypes = new Class[request.arguments.
       length];
    int index = 0;
   for (Object argument : request.arguments)
      parameterTypes[index++] = argument.getClass();

    // The referenced Method, found through reflection
   Method method = module.getClass().getMethod(request.method,
       parameterTypes));

   return method.invoke(module, request.arguments);

}
```

Figure 3.5: The method in which a received *Request* is executed.

Access

The approach shown in Figure 3.5 is naive. It will allow for any method to be executed as long as it is defined within a *Module*. This can lead to unwanted results if the method was not meant for remote invocation. This can also be extended to situations where only some users should be allowed to invoke certain methods. To guard against unwanted invocations the concept of *rights* is introduced. They are strings that can be assigned to users(connections) using a *RightsProvider*. A method can demand rights by using the *@DemandRights* annotation. If the *@DemandRights* annotation is missing, then that method is not allowed to be invoked remotely[4]. The *rights* are not checked if the method is not invoked remotely but locally. An example of using *@DemandRights* is shown in Figure 3.6.

[4]This is to prevent access to methods that have not explicitly been made accessible

```
public class RemoteABCModule extends ABCModule
{
    // doesn't allow remote invocation
    public void A() {
    }

    // doesn't require any rights
    @DemandRights({})
    public void B() {
    }

    // only allows whoever has "admin" and "owner" rights.
    @DemandRights({"admin","owner"})
    public void C() {
    }


}
```

Figure 3.6: Example Remote implementation showing different

3.4 Test application in YARMIS

We will implement the test application using YARMIS. First off, we define a class that extends Module and that indicates which functionality the local and remote implementation should contain. We call this class *HostModule*. In *HostModule* we define the *execute* method. We are also required to define which rights are required to be allowed to execute this method. We will not require any rights, so we set it to an empty array. We are also required to supply a String indicating the type of this module. This is done by supplying that String to the super constructor.

This results in the following code:

```
public abstract class HostModule extends Module{

  public HostModule(){
          super(TYPE);
  }

  @DemandRights({}) // No specific rights required, so empty
       array.
  public abstract String execute(String argument);

    public static final String TYPE = "HostModule";
}
```

Figure 3.7: The FDC of *HostModule*.

Next we need the local and remote implementations of the *HostModule*. The remote implemenation can be created easily from the Module definition. It needs to create a *Request* and send that *Request* to the Host. Sending a *Request* is done through the *ConnectivityPlugin* of the *Core*, available through *Core.connectivity*. The creation of a *Request* is done in the *Module* class, by using the stacktrace. As a result the generated code for methods is always the same, except for the parameters passed to *super.mkRequest(...)*. The created code is as follows:

```
public class RemoteHostModule extends HostModule{

  public String execute(String argument)
  {
      Core.connectivity.sendRequest(super.mkRequest(argument)
          );
  }

}
```

Figure 3.8: The Remote Module definition for the HostModule.

The predefined *Host* requires few modifications. The first modification is that the class needs to extend *HostModule*. This allows the *execute* method to be called from any Client. The second modification is that the *Core* needs to be set up. This requires only two calls. The first is that needs to be indicated that this program is a Host. This will allow Clients to setup a connection to the Host. The second step is that needs to be indicated which instance of any important *Module*s it has. Here it is limited to the local implementation of the *HostModule*, in other words, the instance of *Host*. It is not necessary to indicate which type of *Module* is being set, as the type is already contained in *Module*. Note that the *RemoteHostModule* does not need to be defined on the Host in this case, as the Host will only use the local implementation(which is *Host*).

The full code for the Host when using YARMIS is as shown in Figure 3.9.

```
public class Host extends HostModule{

  public String execute(String argument)
  {
     return "Hello "+argument;
  }

  public static void Main(String[] args)
  {

     Host host = new Host();

     Core.connectivity.setHost(true);
      Core.setModule(host);

  }

}
```

Figure 3.9: The code for the Host using YARMIS

For the client we use the previously defined *RemoteHostModule*. Here the local implementation, in this example called *Host*, does not need to be known, as it will only use the remote implementation.

The Client needs to connect to the Host, and then call *execute* on its *Host* instance. Setting up a connection is again done through *Core.connectivity*. Specifically by calling *connectToHost(...)* providing an ip-address. This will create a LAN-based connection (*LanConnection*).

Next the *RemoteHostModule* needs to be registered as a *Module* for the *Core*. This is similar to how the local implementation of *HostModule* has been set in *Host*, with the distinction that we now need an instance of *RemoteHostModule*. The resulting code is shown in Figure 3.10.

```
public class Client {

  private static final String IP_HOST = "192.168.1.1";

    ...

  public static void main(String[] args)
  {
     Core.connectivity.connectToHost(IP_HOST);
     Core.setModule(new RemoteTestModule());
     perform();
  }

}
```

Figure 3.10: The code to setup a Client using YARMIS

Lastly the implementation for *perform* needs to be created. In that implementation we need to obtain a reference to the instance of *HostModule*, and call *execute(...)* on it. This is done by requesting the *Module* of the type *HostModule.TYPE*, and invoking *execute* on it, as seen in Figure 3.11.

```
public class Client {

  private static final String IP_HOST = "192.168.1.1";
    private static final String MESSAGE = "world";

    private static void perform()
    {
        HostModule hostObj = (HostModule)Core.getModule(
            HostModule.TYPE);
     String result = hostObj.execute(MESSAGE);
     System.out.println(result);
    }

  public static void main(String[] args)
  {
     Core.connectivity.connectToHost(IP_HOST);
     Core.setModule(new RemoteTestModule());
     perform();
  }

}
```

Figure 3.11: The full code for the implementation of a Client using YARMIS

Chapter 4

Research

As could be seen earlier, RMI and YARMIS share roughly the same structure. Both require that their Host does a few basic setup steps, and then it allows an object to be registered for remote access. The same applies to the Client only then it doesn't register an object but it obtains the registered object instead

Within this chapter we'll look at the measured values and we'll explain them by looking at the workings of both YARMIS and RMI. To illustrate these workings, we'll look at the required calls to perform the required operation[1]. This is visualized as a calltrace. This calltrace is displayed following these rules:

- If a method *A(...)* is above method *B(...)* then *A(...)* was called before *B(...)*;
- If *B(...)* is to the right of *A(...)* then a call to *A(...)* will lead to a call to *B(...)*;
- If *A(...)* and *B(...)* have the same indent, then *A(...)* and *B(...)* have been called within the same method;
- If a constructor is called we denote this as a call to *INIT(...)*.
- If a static assignment is done without, calling a specific method, we denote this as a call to *STATIC(...)*.

[1]The RMI sourcecode has been found using http://grepcode.com. The sourcecode for YARMIS is available through http://github.com/knoop/YARMIS

```
public abstract class C
{
    private void A() {
        this.B();
    }

    private void B() {
        // Do something
    }

    public static void main(String[] args){
        C c = new C();
        c.A();
    }

}
```

Figure 4.1: A simple example program to explain how the illustrations should be interpreted.

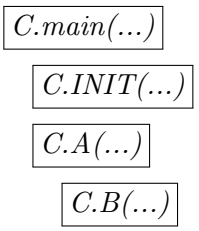


Figure 4.2: A visualization of the simple example program of Figure 4.1.

This visualization can get very cluttered if we show every method call. Hence we will include only calls that are essential, and leave out any detail that does not contribute to the understanding

4.1 Measurement setup

As RMI and YARMIS have a similar structure, it is possible to measure how long each system takes to complete an aspect of their respective structures. This is done for each part used within the test application. For measuring purposes, we let YARMIS be part of the application. This allows us to change parts of the YARMIS code, making it possible to place more measurement points within the structure of YARMIS itself. As RMI comes with JRE, we can not do the same for RMI.

Measuring

For the Host there are two aspects that are measured. The first is the amount of time it takes to setup the server to allow clients to connect. The second aspect is the amount of time it takes to register an object to allow remote invocation. We will refer to this aspect as *binding*. The former is mostly done only once, when the program is started. The latter can be done multiple times to allow access to multiple objects.

For clients we measure three aspects. The first is the amount of time it takes to setup a connection with the host. The second is the amount of time it takes to obtain a reference to an object that is available on the host. We will refer to this aspect as *lookup*. The third and last aspect is the amount of time it takes to invoke the *execute* method on the Host. This includes the time it takes to send the requests over a connection (WLAN or by using Localhost). We will refer to this aspect as execute.

The measurements to determine how much time each aspect takes are done by querying the current time in nanoseconds before and after that aspect is executed[2]. Each measurement is performed a thousand times. The JVM has a tendency to perform optimization when it comes to repetition. As setting up the Client and Host is not something that happens often in real world use, we need to prevent the JVM from performing these optimization steps. Therefore we explicitly do not do any warm-up runs, nor do we perform measurement in repetition. Instead we completely start and stop[3] the Host and Client for each measurement.

Environment

The measurements need to be performed in a controlled environment. This is achieved by using two Raspberry Pis[4]. The advantage of a Raspberry Pi over a regular desktop or server is that it is easier to let them run for long periods of time, while only performing this task. They will switch between hosting and being client.

4.2 Performance comparison

4.2.1 Host

As indicated before, we measure the time it takes to complete the setup for the Host, as well as the amount of time required to bind an object to the Host.

[2] A call to *System.nanoTime(...)* takes on average 41ns. This is neglible

[3] *System.exit(...)*

[4] Only one is used for measuring time. This is a model B, with 512mb of RAM

System	Setup	Binding
RMI	$2.5 * 10^1$ms	$2.8 * 10^0$ms
YARMIS	$3.8 * 10^2$ms	$7.6 * 10^{-1}$ms

Table 4.1: The average host setup time in milliseconds for both RMI and YARMIS

Setup

As shown in Table 4.1 it takes RMI an average of about 25 milliseconds to be set up. For YARMIS it takes about 380 milliseconds. This difference can be explained by looking at the setup processes in more detail.

RMI

During this setup process two important steps are performed. First a *UniCastServerRef* instance is created. This class inherits from the *RemoteRef* interface through a super class called *UniCastRef*. The *RemoteRef* interface requires that a given *Method* can be invoked on a *Remote* object. The method provided will be the method that was invoked on a *Remote* object. This method is obtained through the use of *Proxies*. This calltrace is also shown in Figure 4.3. A *Proxy* is an implementation of an interface that is generated at runtime. It only allows calls to the methods defined in the interface it implements. Every time a method is invoked, its corresponding *Method* instance is passed on to a *InvocationHandler*. How these *Proxies* are used in RMI is discussed in section 4.2.2.

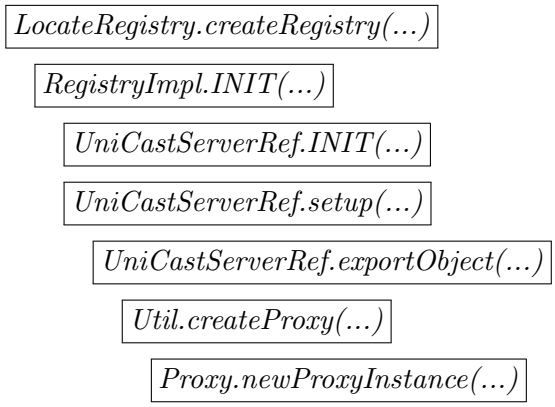


Figure 4.3: The calltrace of the setup process of the RMI Host.

YARMIS

The setup process of the *Core* is started when the *Core* is first referenced. This is due to the fact that *Core* consists solely of static attributes. The setup requires the creation of a *HashMap*[5] and a *ConnectivityPlugin*. The *ConnectivityPlugin* requires a *RequestReceiver*, a *RequestSender* and a *ConnectionManager*. The *RequestReceiver* creates a *Cached ThreadPool*. The *RequestSender* creates a new *Thread* and starts it. The *ConnectionManager* only creates a new *HashMap*. The calltrace of these calls is shown in Figure 4.4. The required times to execute each of the steps is show in Table 4.2.

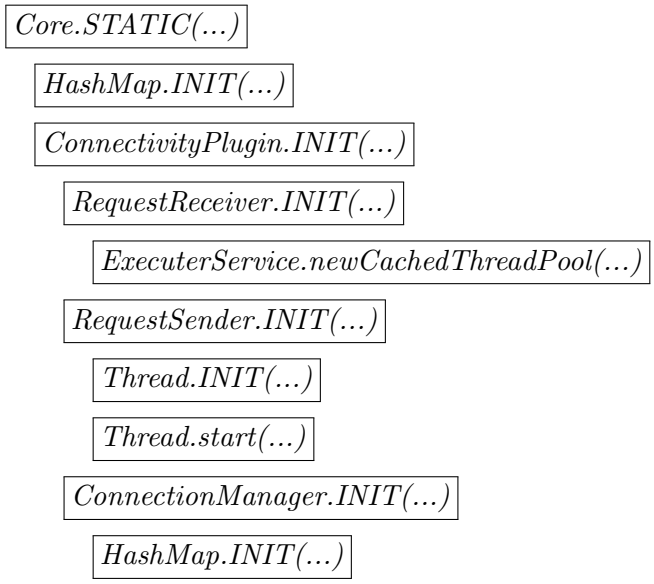


Figure 4.4: The calltrace of the setup process of the YARMIS *Core*.

Setup (overall)	ConnectivityPlugin	RequestReceiver	RequestSender	ConnectionManager
$1.6 * 10^2$ms	$8.9 * 10^1$ms	$4.3 * 10^1$ms	$1.0 * 10^0$ms	$1.6 * 10^1$ms

Table 4.2: The average time required to pass through the different stages of the YARMIS Core setup.

Here we see that the setup for *Core* is mostly dependent on the creation of *ConnectivityPlugin*. The most costly operation within the construction of *ConnectivityPlugin* is the creation of a *RequestReceiver*. The time required by *RequestReceiver* is spent on creating a *ThreadPool*.

When the *Core* is setup, it also needs to be set to hosting. This is done by telling its *ConnectivityPlugin* to fullfil the Host role. Within *fulfillRole(...)*

[5]For mapping a *String* to a *Module*

checks to see if it is not already hosting are performed. If it isn't hosting *startHosting(...)* is called. The time it takes for this method to finish is what we call $Hosting_{start}$. In *startHosting(...)* three steps are performed. First all still existing connections are removed. This we'll call *Clearing*. When all connections have been removed a new *Thread* is created and started. Within this *Thread* *ConnectionManager.host(...)* is called. The creation and starting of that *Thread* are the last two steps we measure being $Thread_{create}$ and $Thread_{start}$ respectively. The calltrace of these calls is also made visible in Figure 4.5. We will not measure *ConnectionManager.host(...)*. This is due to the fact that it is not part of the setup but rather what has been set up. Furthermore its execution has no effect on the main thread as its done asynchronous.

Figure 4.5: The calltrace of the setup process of the YARMIS host.

Within this calltrace we'll measure how long each of the steps takes to set the Host up. The results can be found in Table 4.3.

$Hosting_{start}$	Clearing	$Thread_{create}$	$Thread_{start}$
$8.5 * 10^0$ms	$1.6 * 10^0$ms	$5.8 * 10^0$ms	$7.6 * 10^{-1}$ms

Table 4.3: The average time required to pass through the different stages of *fulfillRole(...)*.

Binding

RMI

Binding a *Remote* in RMI is done by placing that *Remote* in a *HashTable*, using the name to bind it under as a key, and the *Remote* as the value. This is done within a synchronized block. Before a *Remote* is bound, it is checked that the caller is allowed to bind a *Remote*, by calling checking where the request to bind came from. The calltrace is as shown in Figure 4.6.

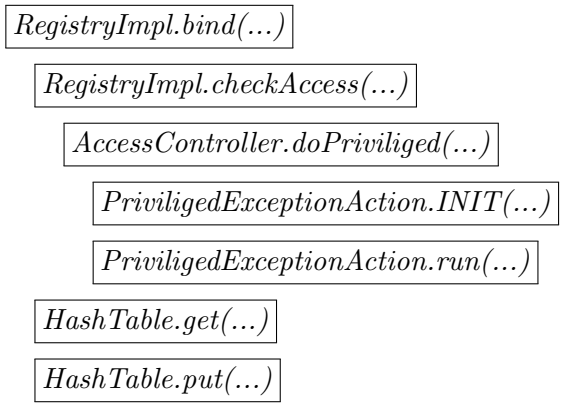


Figure 4.6: The calltrace for binding in YARMIS.

YARMIS

Binding a *Module* in YARMIS is done by placing the *Module* inside a *HashMap* that is inside *Core*. The type of the *Module* is used as the key, and the *Module* itself as the value. The *Module* that is replaced will be notified by calling *dismiss(...)*, and subsequently *onDismiss(...)* for extensions of *Module*, to perform required operations. The new *HashMap* is notified by calling *bind(...)* which calls *onBind(...)*. This is shown in Figure 4.7. It is susceptible to the implementation of *onDismiss(...)* and *onBind(...)*. If extensions perform heavy operations in these calls, then the overall time required to bind a *Module* will be higher.

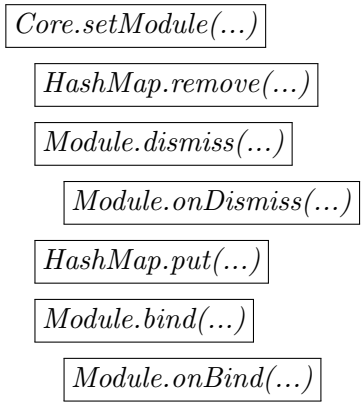


Figure 4.7: The calltrace for binding in YARMIS.

Based on the small implementation differences the time gain by YARMIS should be sought in the lack of synchronization and the use of a *HashMap*.

In principle the lack of synchronization isn't as disastrous as it might appear. It is true that YARMIS uses multiple threads to handle the different Clients a Host can be connected to. This multi-threaded system doesn't pose an immediate problem, *Core* can only be accessed locally. It is however possible to use multiple threads to access the *Core* locally, which will still require synchronization, making the YARMIS approach more similar to RMI.

Invocation

Both systems require a way for the Host to get from a received request for invocation to actual invocation. These steps happen in the background. As a result it is not possible to measure the amount of time RMI spends to turn a request into actual invocation. A comparison between the required method calls can still be made.

RMI

For RMI the process starts with a call to *TCPTransport.handleMessage(...)*. Here it is checked whether the incoming message is a RMI call. If it is, it is passed to *Transport* where a security check is performed using *AccessController*. The received call is then dispatched to *UnicastServerRef*[6]. The *UnicastServerRef* holds a mapping to identify *Methods* by their hash (stored as *long*). Specifically, it is a *WeakClassHashMap* that maps the *Class* of the referenced object to a mapping from the hash to the corresponding *Method*. The referenced object is identified using its id. When the *Method* is found, it is invoked on the reference object using the parameters provided by the incoming call. The result of this invocation (if any) is then sent back. This calltrace is shown in Figure 4.4.

The previously mentioned *WeakClassHashMap* is extended by *HashToMethod_MAPS*. It is used as a cache. If a *Class* is known, it will return the mapping as expected. Whenever a *Class* is requested that isn't yet mapped, it will create a new empty mapping from *Long* to *Method*. It then fills this mapping as follows. It will traverse each interface defined in that class. For each interface that extends *Remote*, it will calculate the hash of each *Method*, and store that hash along with the *Method* in the mapping. Furthermore, it will indicate to the *AccessController* that this method should be accessible. Once it is done iterating over all interfaces of the given *Class*, it will move on to the super class and repeat the process.

Once this process is done, every method that is defined by a *Remote* interface is made accessible and will be stored in the mapping, using its hash as the key. This makes it possible for RMI to find a *Method* based on a hash.

[6] As *UnicastServerRef* an implementation of *Dispatcher* and is the provided *Dispatcher*

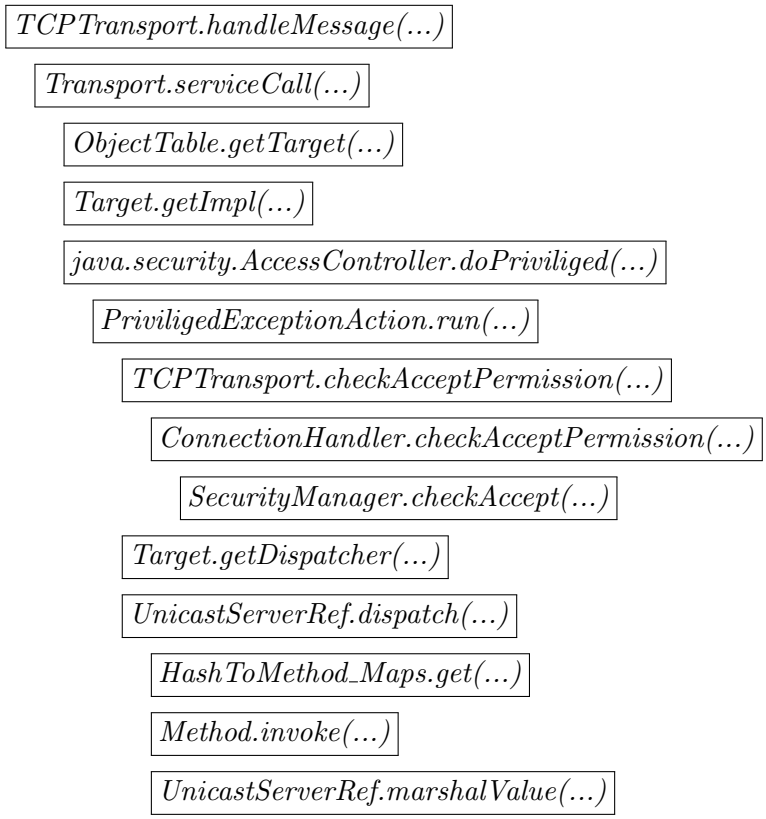


Figure 4.8: The calltrace of the invocation process of the RMI Host.

YARMIS

In YARMIS the process of invoking a method on the Host from a received message starts with a *ConnectionReader*. A *ConnectionReader* reads data from the *Connection* that it is assigned to. When a *Message* is received over that *Connection*, it unpacks that *Message* and inspects its payload. In this context the payload will be a *Request*, as only *Requests* can be executed. The other option, *Response*, is used to be notified of the outcome of a *Request* that was sent.

The received *Request* is sent to the *ConnectivityPlugin* to be processed by its *RequestHandler*. The *RequestHandler* will create a *RequestRunner* that will run the *Request*. This *RequestRunner* is then placed in a queue to be executed by a *ExecutorService*. This allows the *ConnectionReader* to be ready for new input. The calltrace is shown in Figure 4.9.

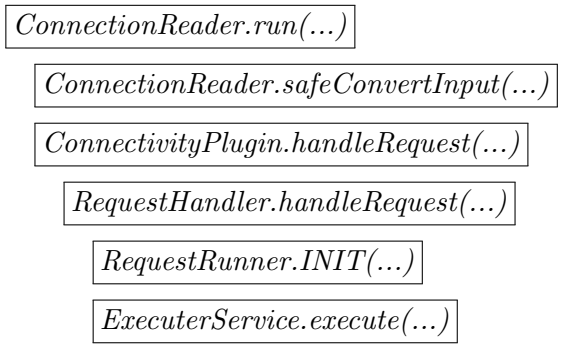


Figure 4.9: The calltrace of the invocation process of the YARMIS Host up to the placement in the executor queue.

After the *RequestRunner* is placed in the queue, it will be executed. First it makes a call to get the rights that are assigned to the *Connection* that made the *Request*. Then it finds the *Method* that is mentioned in the *Request*. To do so, it obtains an instance of the *Module* mentioned in the *Request*, and looks for the correct combination of the method name provided by the *Request* as well as the type of the provided arguments. When this *Method* is obtained, it compares the rights required[7] with the rights that are assigned to the *Connection*. If the *Connection* is assigned the required rights, the *Method* is invoked using the arguments stored in the *Request*. The return value is then packed and sent back to the connection. This calltrace is shown in Figure 4.10.

[7]These rights come from the *@DemandRights* annotation

Figure 4.10: The calltrace of the invocation process of the YARMIS Host when a request is taken from the queue and executed.

The time it requires to perform a request is shown in Table 4.4. The start is seen as when the Host first knows that it has received a *Request*. At this point it has already read the *InputStream*. The amount of time between placing a *RequestRunner* in the queue, and its *run(...)* method being called is indicated as *queue*. The amount of time required to use the *Request* to find the referenced *Method* and invoking that *Method* is indicated as textitperform. The time it takes for the result to be converted to a byte array is expressed as *converting*. The last step is self explanatory.

Total	queue	perform	converting	sending
$1.8 * 10^1$ms	$1.2 * 10^0$ms	$2.3 * 10^0$ms	$2.0 * 10^0$ms	$1.2 * 10^1$ms

Table 4.4: The average time required to pass through the different stages of performing a *Request*.

4.2.2 Client

As indicated earlier, we measure three aspects for the Client. These aspects are the setup, the lookup, and the execute processes.

System	Setup	Lookup	Execute
RMI_{local}	$5,9 * 10^2$ ms	$9,0 * 10^2$ ms	$1,4 * 10^1$ ms
YARMIS_{local}	$7,8 * 10^2$ ms	$3,6 * 10^{-1}$ ms	$9,4 * 10^2$ ms
RMI_{WLAN}	$5,9 * 10^2$ ms	$9,1 * 10^2$ ms	$2,1 * 10^1$ ms
YARMIS_{WLAN}	$3,8 * 10^2$ ms	$2,5 * 10^{-1}$ ms	$5,0 * 10^2$ ms

Figure 4.11: The average time it takes in milliseconds for processes of the client to finish for both RMI and YARMIS

As shown in Table 4.11 the differences between RMI and YARMIS are quite large. The performance differences are explained below using the workings of RMI and YARMIS. We'll first explain the difference between the two systems regarding the setup process. Due to a dependency between *lookup* and *execute* for RMI, we'll then look at the *execute* process before looking at *lookup*.

Setup

RMI

The setup time for a Client in RMI is confined to getting a reference to a *Registry*. In RMI the *Registry* with registered objects is stored on the Host. The *Registry* that the Client uses is only a reference to the *Registry* stored on the Host. So *Registry* is a *Remote* object that allows access to other *Remote* objects. The usual way for obtaining a *Remote* object is not possible here as it is the *Registry* that we are trying to obtain. So for this situation *LocateRegistry* must create a *Proxy* that forwards all calls to the *Registry* on the Host.

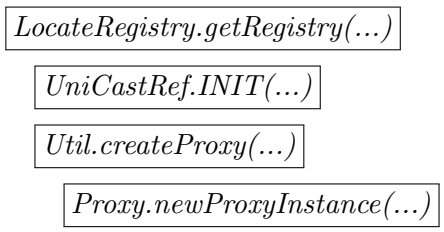


Figure 4.12: The calltrace of the setup process of the RMI Client.

YARMIS

For YARMIS we need to first initialize the *Core*, this is done automatically when *Core* is referenced for the first time. This setup has been explained in Section 4.2.1 and will not be explained here.

When the *Core* is setup, the Client needs to connect to a Host. This will create a new *LanConnection* using a created *Socket*. When this *Connection* is created, it is registered as an active connection within the *ConnectionManager*.

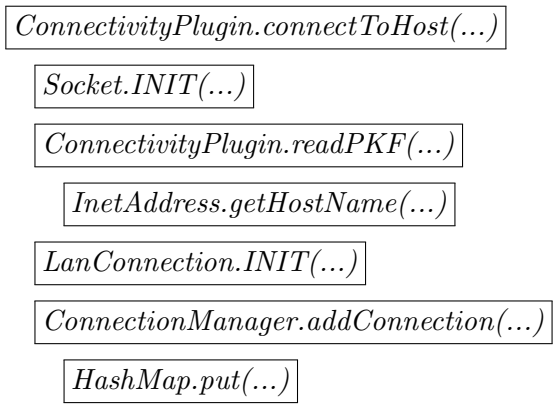


Figure 4.13: The calltrace of the setup process of the YARMIS Client.

We measure the amount of time it requires to perform each of these calls. The measured times per part can be found in Table 4.5.

Total	Socket	readPKF	LanConnection	addConnection
$2.8 * 10^2$ms	$1.9 * 10^2$ms	$5.3 * 10^1$ms	$3.2 * 10^1$ms	$4.7 * 10^{-1}$ms

Table 4.5: The average time required to pass through the different stages of the call to *connectToHost(...)*.

The setup time for a client is quite similar for RMI and YARMIS, being in favor of YARMIS by about 100ms. An important difference between the two systems during setup is that RMI requires much more communication to the Host. YARMIS doesn't require any communication by itself. So while the setup of *Core* takes a lot of time, it is compensated by requiring virtually no communication with the Host.

Execute

RMI

RMI uses *Proxy* to create remote references to objects. These proxies are always assigned a *RemoteObjectInvocationHandler* to handle any method calls. In handling these methods a distinction is made between methods whose methods are declared by *Object*, and those declared by other classes. The latter are always sent to the Host. The former are given a special

treatment. For instance, *equals(...)* is used to let the *Proxy* of an instance be equal to that instance.

Sending an invocation call

The methods intended for remote invocation are passed to a *UnicastRef*. This can either be a default instance of *UnicastRef* or a stub. Both provide the functionality required to invoke a given method on the Host. While *UnicastRef* can be used to handle calls to any class, a stub is defined by the developer for a specific class. In this example we will not define a stub and use the default instance of *UnicastRef* instead.

UnicastRef uses a *StreamRemoteCall* to allow invocation on a remote object. When a *StreamRemoteCall* is created, it tells the Host that it is about to invoke a method on a registered object. This is indicated by sending three messages. The first is an identifier of the registered object on which a method should be invoked. The second message sent is the operation type. The final message is the hash of the method that was invoked on the Client. This hash is calculated by *Util.computeMethodHash(...)*. For performance improvement, this hash is cached within *RemoteObjectInvocationHandler*.

Initially, no arguments are sent. With the data that has been sent, the Host can prepare the call by obtaining the referenced object, as well as the referenced method. This method is sent as its own hash. As demonstrated in Section 4.2.1, the method can be found on the Host using that hash. This allows for a limited amount of data to be required for the Host to be able to know which method should be used.

Arguments

As the next step the arguments need to be send to the Host. This is done using an *ObjectOutput* which is obtained through the *StreamRemoteCall*. This is used to marshal the arguments, and then to sent them to the Host. The arguments to a method call are passed during the call to *UnicastRef.invoke(...)* by calling *marshalValue(...)* for each argument.

Finishing

When all arguments are sent, the request for execution is finished by calling *executeCall(...)* on the *StreamRemoteCall*. This will wait for the return type to be sent back from the Host. This return type indicates whether the execution finished successfully, or with an exception. In case of the latter, the *Exception* that was thrown on the Host is unmarshalled and thrown on the Client too. In case of the former, the call to *executeCall(...)* finishes. If required, the object returned by the Host will be unmarshalled from the *StreamRemoteCall*'s *InputStream* and is returned, thus ending the invocation process.

This procedure is shown in Figure 4.14.

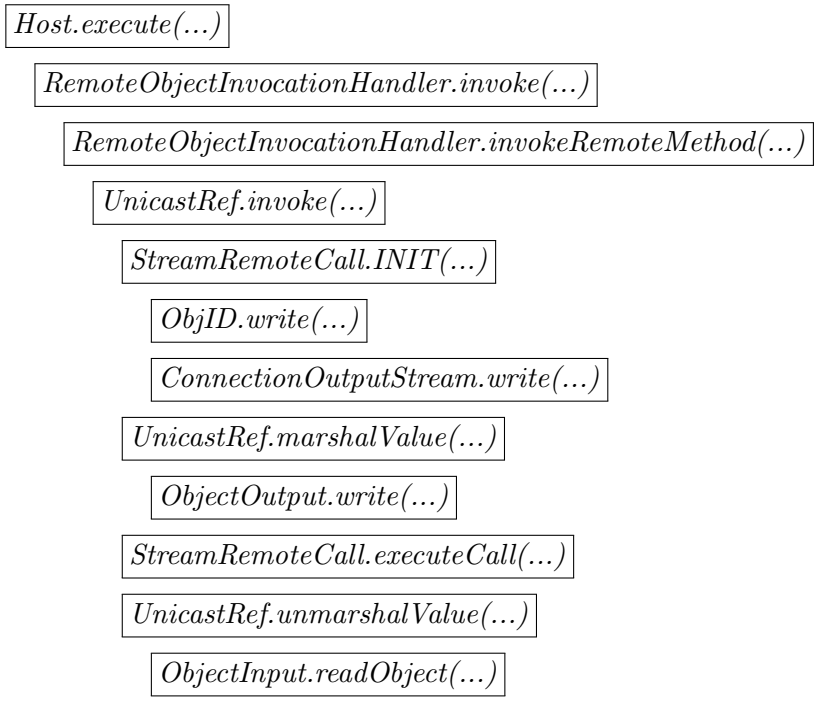


Figure 4.14: The calltrace of the remote execution process of the RMI client. The calltrace of calling *execute(...)* on a remote instance of *Host*.
.

YARMIS

The first step for remote execution in YARMIS is creating a *Request*. This *Request* is then added to a queue to be send over a *Connection*. When the *Request* is added to the queue, a *Result* is provided. This *Result* is then used to wait for the response of the Host. It blocks the current *Thread*, and will be released when a response has been received. The sending of a *Request* from the queue is done in a separate *Thread*.

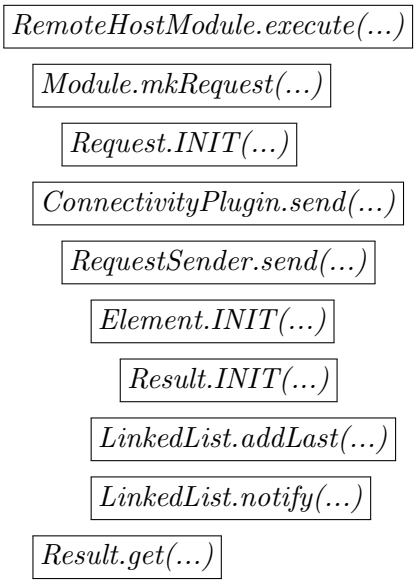


Figure 4.15: The calltrace of the remote execution process of the YARMIS Client. Not shown here is the asynchronous sending process. This is instead shown in Figure 4.16

The asynchronous sending of *Requests* is done through the *RequestSender*. The *RequestSender* will send each element of its queue to the required *Connection*. A part of sending *Requests* is placing it in a *Message*. When a *Request* is placed in a *Message*, a byte representation of the *Request* is created as well. This is shown in Figure 4.15.

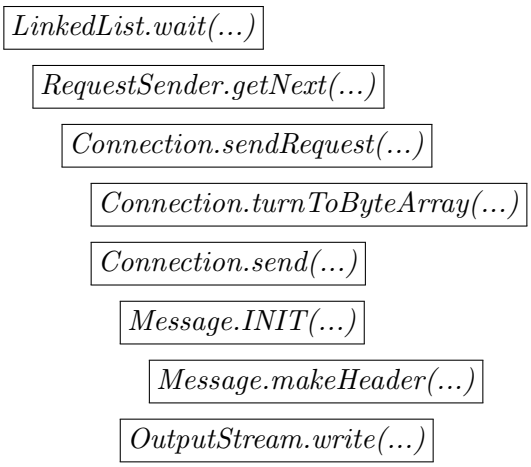


Figure 4.16: The calltrace of the remote execution process of the YARMIS client.

Lookup

RMI

In RMI objects need to be looked up through *Registries*. As indicated earlier, *Registry* is a remote object. The calltrace of the *lookup(...)* call is therefore nearly identical to that of *execute(...)*. The only difference being that the first called method is *lookup(...)*. The calltrace of *lookup(...)* can be found in Figure 4.17..

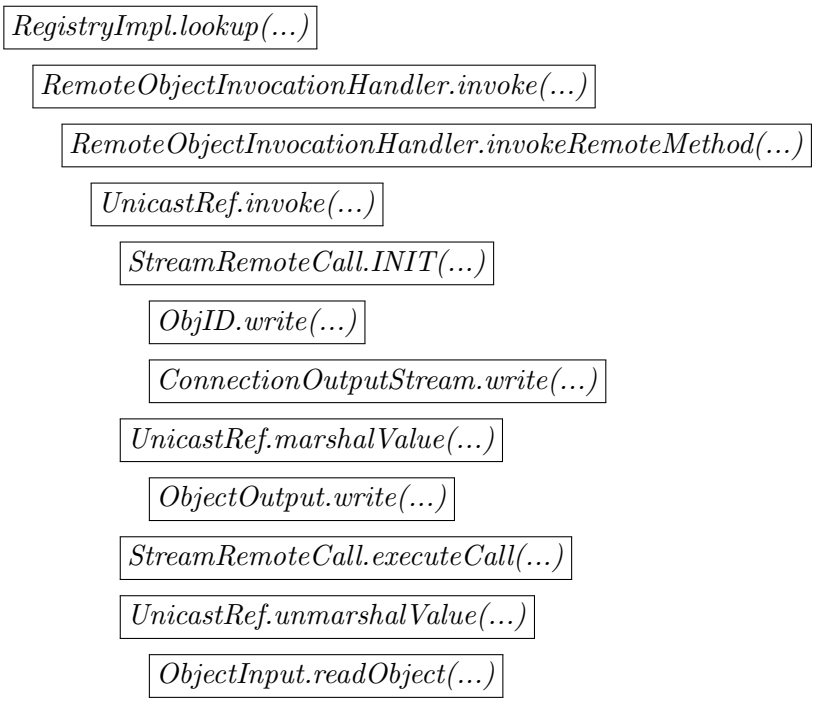


Figure 4.17: The calltrace of the lookup process of the RMI Client. Note that calltrace is, apart from the first call, identical to the one displayed in 4.14

YARMIS

The required amount of steps to look up a *Module* on YARMIS is much lower. YARMIS keeps local references for remote objects, therefore it does not require any communication between the Client and Host. It is nothing more than interacting with a *HashMap* to either store or obtain a *Module* instance, albeit through a synchronized block. The calltrace can be seen in 4.18. The fact that YARMIS is not only simpler, but also doesn't require communication makes it many times faster than RMI, as can be seen in Table 4.11.

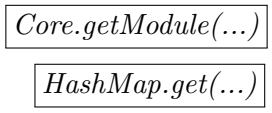


Figure 4.18: The calltrace of the look up process of the YARMIS client.

4.3 Limitations comparison

Method selection

YARMIS contains an important flaw. Whenever arguments are casted, be it explicitly or implicitly, then the wrong method will be called. This is due to the fact that YARMIS uses a combination of a method name, as well as the class of the given arguments to determine which method should be invoked. If the classes of the given arguments does not exactly match the method signature then the call on the Host will be to the wrong method.

Consider the following example. Suppose that we have defined the method *method(Object obj)* for remote invocation. We call this method, providing a *String* as an argument. A *Request* containing the method name ("method") and the parameter type will be send to the Host. The Host will then try to find a *Method* with the name "method" and as argument type *String*. Then either the method won't be found, or the wrong method is found. Either way this is an unwanted outcome.

To fix this problem, the Host needs to know the correct method signature. This can not be derived from the provided arguments alone, as we do not know to which class each argument was casted. This is due to the fact that casting is only used at compile time to determine which method should be invoked [4]. As a result we need to provide the correct signature, by indicating what the expected classes are of arguments. For this we expand *Request* and *mkRequest(...)* to require an array of classes. We then supply the expected classes at compile time. The resulting implementation for a remote instance is shown in Figure 4.19. When a *Request* is being performed at the Host, it no longer uses the classes of the arguments, but rather this supplied array of classes.

```
public void method(Object value) {
    super.mkRequest(new Class<?>[]{Object.class}, value);
}

public void method(String value) {
    super.mkRequest(new Class<?>[]{String.class}, value);
}
```

Figure 4.19: A very simple example program to show how the improvement works client side.

In RMI this problem doesn't exist. RMI identifies methods differently, removing the need to know the exact parameters. The solution RMI presents isn't applicable for YARMIS. RMI hinges on the fact that it receives a *Method* instance from a *Proxy*. On the Host this *Method* can be used to identify and invoke the correct *Method*[8]. YARMIS can't use *Proxies* due to the fact that the FDCs are abstract classes instead of interfaces. This makes it infeasible to obtain the actual *Method*. Instead only to the name of the method can be obtained, using the stack trace.

It is not entirely impossible to obtain the called *Method* without the use of a *Proxy*. Anonymous classes have a property called *EnclosingMethod*. This is the *Method* instance in which the anonymous class was created. By adding the call *Method m = (new Object()).getClass().getEnclosingMethod()* *m* will contain the *Method* that this code was executed in. This solution is however a hack, and shouldn't be used to solve the problem.

Garbage Collection

RMI uses *Distributed Garbage Collection* to perform garbage collection on objects registered in *Registries*. If remotely available objects are no longer referenced on any machine then they will be garbage collected.

This functionality does not exist in YARMIS. The reason lies in the intended usage of YARMIS. *Modules* are intended to provide the functionality of an application. You usually don't throw functionality away. This is why *Core* is designed to hold only one *Module* of each type. It can be used to hold multiple objects, but each of them will need to provide a different type when they are registered on the Host.

If the necessity arises to create and discard a lot of different *Modules* then this means that either YARMIS wasn't used as intended, or that a lot of switching between being a Host or Client is happening. The latter isn't a problem as long as the user does not hold on to instances of *Modules*, but instead calls *Core.getModule(...)* each time it needs a specific *Module*. *Core*

[8]This is a simplified run-down. See Section 4.2.2 and Section 4.2.1 for a more detailed explanation.

does not hold on to an old *Module* so as long as the application doesn't either, it will be garbage collected.

Access control

RMI and YARMIS can both use the standard *SecurityManager*. This can check whether the running thread is granted the required *Permissions* to perform a certain operation. Many basic *Permissions* have been predefined, but it is also possible to create custom *Permissions*. The granted *Permissions* depend on the active *Policy*.

Guarding methods from unintented invocation

RMI checks that a requested method is allowed to be invoked. This is done by checking that the referenced method is required by an interface that extends *Remote*. If this isn't the case then the call is forbidden. This situation can happen when a remotely accessible object implements other interfaces or extends other classes. If these classes were not intended for remote invocation, then the method shouldn't be allowed to be invoked.

YARMIS' rights system is used to provide the same security. Allowed calls however are not linked to their class but to whether they have the *@DemandRights* annotation. This would also allow methods that are not defined in an FDC to be invoked if they have the *@DemandRights* annotation.

User groups

Unlike with RMI, YARMIS' rights system also makes it possible to let access be specified for user groups[9] or individual users[10]. This can be useful if you want to have an *Administrator* who has the rights to change settings of the application, while other users do not have those rights. It can also be used to enforce that someone is a registered user to be able to do anything on the Host.

This system is not perfect due to the fact that its security checks are very superficial. Only the first method call is checked. This allows for a security breach in case that a non (or limited) secured method calls a high secured method. The security of the second method is then not checked, while the first method is allowed to be invoked by (nearly) everyone. It is essential that the rights required to execute specific methods are well chosen to prevent these kind of situations.

[9] Assign a right to multiple users
[10] Assign a right to a single user.

Chapter 5

Related Work

RMI has been compared before. Govindaraju et al [5] have looked at whether it is feasible to use RMI in a scientific environment. They have investigated whether SOAP RMI, which is RMI that serializes to SOAP rather than use the default serialization. [5]. They have shown that RMI outperforms SOAP RMI. This has been attributed to the fact that XML is an inefficient data format. It is also shown that RMI outperforms *nexusRMI*.

Juric et al [7] have compared RMI different methods to let RMI work over firewalls and proxies. They compared plain RMI with HTTP-to-port, HTTP-to-GCI and Java Web Services. They have shown that plain RMI for the most part outperforms the other techniques. They have measured the performance for sending different datatypes, which lets it be more indicative than using only one datatype. With these measurements, Java Web Services has been found to be the best alternative out of the investigated options.

Chapter 6

Conclusions

YARMIS provides a few advantages over RMI. It is quicker in allowing you to access remote objects. It also supports more security by letting rights be assigned to different users. However, the execution of remote methods happens significantly slower. Worse is the fact that YARMIS will not always find the correct method, and might even invoke the wrong method.

As it stands now, YARMIS is better of as a special version of RMI. The underlying functionality of RMI can be used, while YARMIS' *Modules* and *Core* can function as a replacement for *Registry* and *Remote*. Otherwise YARMIS has to be significantly improved, to let its unique features make it stand out, rather than having its flaws make it unusable. This can be achieved by incorporating *Proxy* and using to to create the remote instances for *Modules*. This would remove the necessity of user defined remote implementations. This can also remove the necessity to manually set an instance of a local or remote implementation of a *Module*. This can be achieved by letting the *Module* contain the local implementation and the *Proxy*. If the *Module* then forwards every call to the correct instance, switching between local and remote can be limited to telling the *Module* to flip a boolean.

YARMIS is already dynamic in the sense of remote versus local. If it came overcome its flaws then it will truly be a dynamic alternative to RMI.

Bibliography

[1] Common object request broker architecture (corba) specification, version 3.3. Technical report, OMG.

[2] Corba specification. Technical report, Object Management Group.

[3] Rmi specification. Technical report, Oracle.

[4] J Gosling, B Joy, G Steele, G Bracha, and A Buckley. The java language specification java se 7.0 edition. Technical report, Oracle.

[5] Madhusudhan Govindaraju, Aleksander Slominski, Venkatesh Choppella, Randall Bramley, and Dennis Gannon. Requirements for and evaluation of rmi protocols for scientific computing. In *Proceedings of the 2000 ACM/IEEE conference on Supercomputing*, page 61. IEEE Computer Society, 2000.

[6] Object Management Group. History of corba, 2014.

[7] Matjaz B Juric, Bostjan Kezmah, Marjan Hericko, Ivan Rozman, and Ivan Vezocnik. Java rmi, rmi tunneling and web services comparison and performance analysis. *ACM Sigplan Notices*, 39(5):58–65, 2004.

Appendix A

Measurements

A.1 RMI

Host

Measurement	average value (ns)
presetup	47.733
midsetup	25.046.286
postsetup	27.858.496

Client

Measurement	average value (ns)
presetup	0
postsetup	587.620.984
prelookup	587.681.003
postlookup	1.499.279.278
preexecute	1.499.333.183
postexecute	1.520.306.007

A.2 YARMIS

Host

For YARMIS the measurement is more detailed. The first Host measurement for YARMIS is a detailed look at the setup.

The second measurement is about how much time it takes to execute an incoming *Request*. The naming for measurements is less obvious. The measurement points are as follows:

- start/0: *ConnectionReader.run(...)*, when it is known that *received* is an instance of *Request*.
- requestpreperform: *RequestRunner.run(...)*, immediately.
- requesthandled: *RequestRunner.run(...)*, before the call to *connection.sendResponse*.
- preconvert: *Connection.sendResponse(...)*, before the call to *turnToByteArray(...)*.
- presendbyte: *Connection.sendResponse(...)*, after the call to *turnToByteArray(...)*, before the call to *send(...)*.
- responsesend: *Connection.sendResponse(...)*, after the call to *send(...)*.

Setup

Measurement	average value (ns)
presetup	50.880
precoresetup	76.495
preconnectivitypluginconstructor	67.115.610
prerequestreceiverconstructor	78.978.580
postrequestreceiverconstructor	122.404.317
prerequestsenderconstructor	139.274.580
prerequestsenderstart	139.536.222
postrequestsenderstart	140.296.468
postrequestsenderconstructor	140.358.453
preconnectionmanagerconstructor	156.208.045
postconnectionmanagerconstructor	156.392.918
postconnectivitypluginconstructor	156.448.542
postcoresetup	156.547.288
prerolesetup	156.574.420
postrolesetup	167.687.181
prestarthosting	168.204.744
preremoveconnections	168.309.699
postremoveconnections	169.958.593
precreatethread	170.010.992
postcreatethread	175.779.987
prestartthread	175.833.109
poststartthread	176.592.466
poststarthosting	176.662.164
midsetup	176.696.918
postsetup	177.306.328

performing *Request*

Measurement	average value (ns)
requestpreperform	1.249.139
requesthandled	3.583.698
preconvert	3.616.730
presendbyte	5.613.969
responsesend	18.307.721

Client

The Client has also been measured more extensively for YARMIS. In the first measurement the setup is measured. The second measurement is a detailed look at what happens when a remote method is invoked.

Setup

Measurement	average value (ns)
presetup	107.563
precoresetup	170.178
postcoresetup	162.250.999
presetmodule	162.330.467
postsetmodule	164.575.267
preconnecttohost	164.660.869
presocketconstructor	165.035.447
postsocketconstructor	353.116.311
prelanconnectionconstructor	406.377.113
postlanconnectionconstructor	439.056.376
preaddconnection	439.161.715
postaddconnection	439.635.154
postconnecttohost	439.687.867
postsetup	439.178.699

Making *Request*

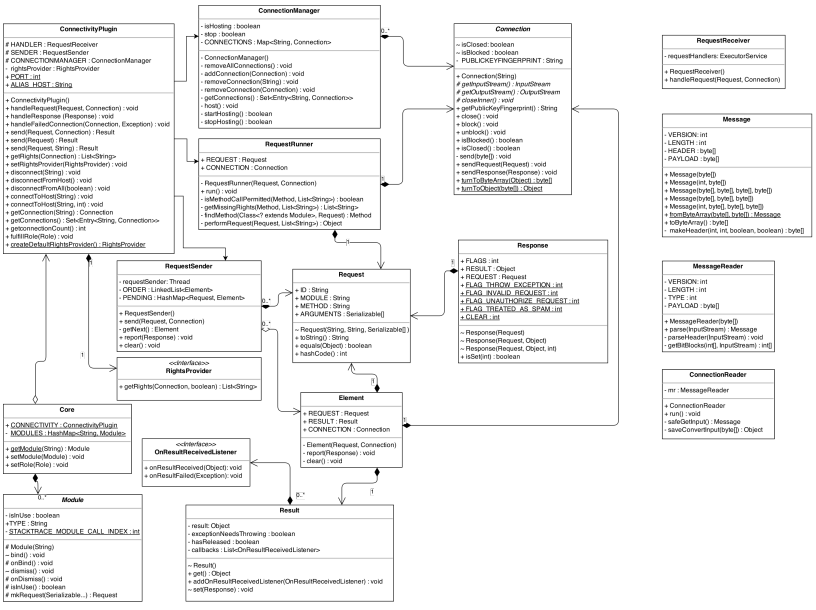
Measurement	average value (ns)
postsetup	379.623.735
prelookup	379.723.042
postlookup	379.958.029
preexecute	380.032.854
premkrequest	380.148.265
postmkrequest	394.548.714
presendrequest	394.663.485
postsendrequest	417.780.199
preunpackresult	417.882.035
prereceived	418.026.926
postreceived	877.292.590
postunpackresult	877.400.304
postexecute	877.439.045

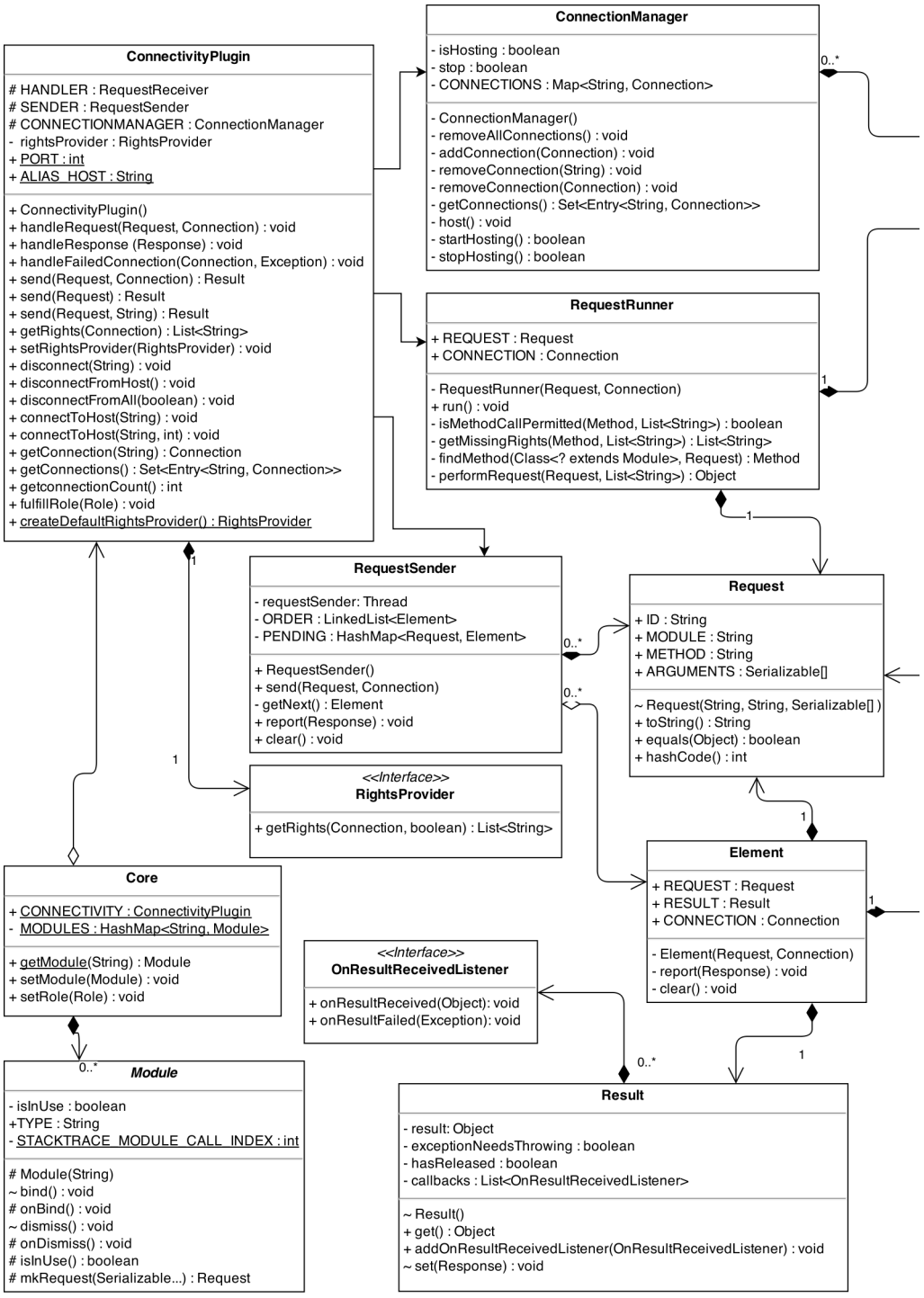
Appendix B

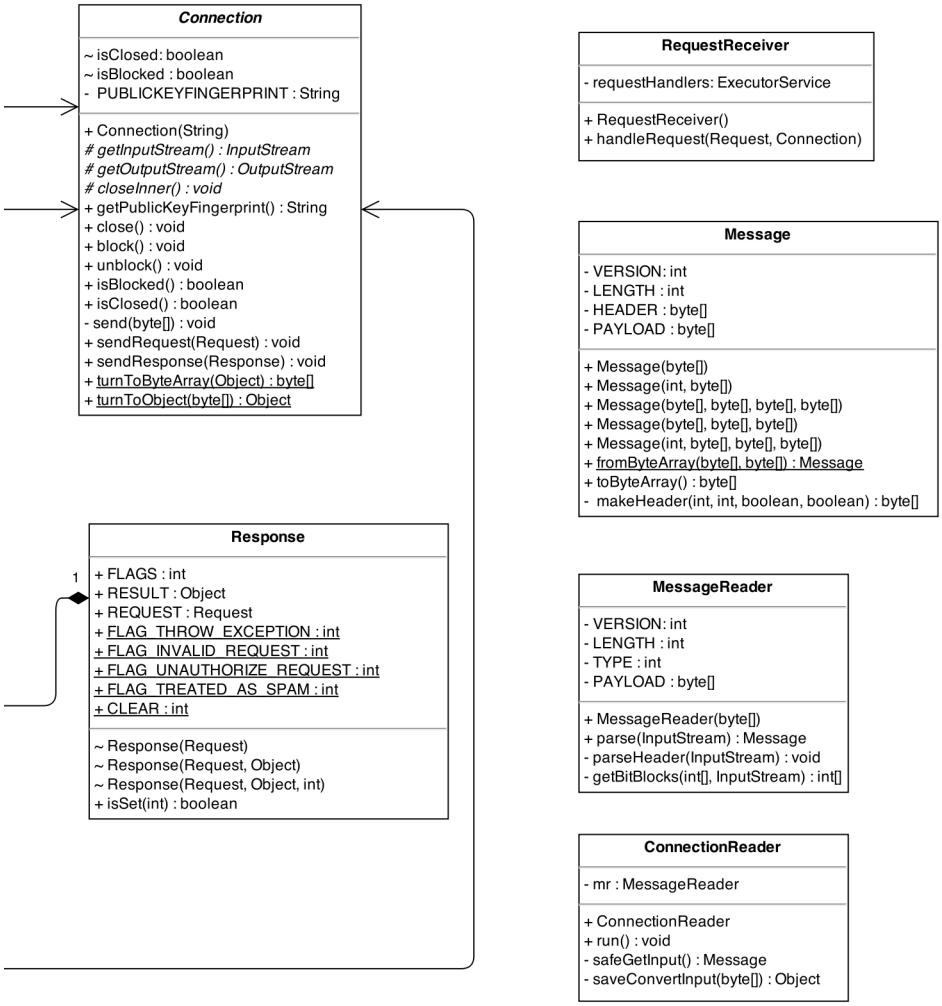
UML

The full UML Diagram of YARMIS is shown below. for readability, larger versions have been included after the full view.

Full view




ConnectionManager
- isHosting : boolean
- stop : boolean
- CONNECTIONS : Map<String, Connection>
- ConnectionManager()
- removeAllConnections() : void
- addConnection(Connection) : void
- removeConnection(String) : void
- removeConnection(Connection) : void
- getConnections() : Set<Entry<String, Connection>>
- host() : void
- startHosting() : boolean
- stopHosting() : boolean
ConnectivityPlugin
HANDLER : RequestReceiver
SENDER : RequestSender
CONNECTIONMANAGER : ConnectionManager
- rightsProvider : RightsProvider
+ PORT : int
+ ALIAS_HOST : String
+ ConnectivityPlugin()
+ handleRequest(Request, Connection) : void
+ handleResponse (Response) : void
+ handleFailedConnection(Connection, Exception) : void
+ send(Request, Connection) : Result
+ send(Request) : Result
+ send(Request, String) : Result
+ getRights(Connection) : List<String>
+ setRightsProvider(RightsProvider) : void
+ disconnect(String) : void
+ disconnectFromHost() : void
+ disconnectFromAll(boolean) : void
+ connectToHost(String) : void
+ connectToHost(String, int) : void
+ getConnection(String) : Connection
+ getConnections() : Set<Entry<String, Connection>>
+ getconnectionCount() : int
+ fulfillRole(Role) : void
+ createDefaultRightsProvider() : RightsProvider
RequestRunner
+ REQUEST : Request
+ CONNECTION : Connection
- RequestRunner(Request, Connection)
+ run() : void
- isMethodCallPermitted(Method, List<String>) : boolean
- getMissingRights(Method, List<String>) : List<String>
- findMethod(Class<? extends Module>, Request) : Method
- performRequest(Request, List<String>) : Object
RequestSender
- requestSender: Thread
- ORDER : LinkedList<Element>
- PENDING : HashMap<Request, Element>
+ RequestSender()
+ send(Request, Connection)
- getNext() : Element
+ report(Response) : void
+ clear() : void
Request
+ ID : String
+ MODULE : String
+ METHOD : String
+ ARGUMENTS : Serializable[]
~ Request(String, String, Serializable[])
+ toString() : String
+ equals(Object) : boolean
+ hashCode() : int
<<Interface>>
RightsProvider
+ getRights(Connection, boolean) : List<String>
Core
+ CONNECTIVITY : ConnectivityPlugin
- MODULES : HashMap<String, Module>
+ getModule(String) : Module
+ setModule(Module) : void
+ setRole(Role) : void
Element
+ REQUEST : Request
+ RESULT : Result
+ CONNECTION : Connection
- Element(Request, Connection)
- report(Response) : void
- clear() : void
<<Interface>>
OnResultReceivedListener
+ onResultReceived(Object): void
+ onResultFailed(Exception): void
Module
- isInUse : boolean
+TYPE : String
- STACKTRACE_MODULE_CALL_INDEX : int
Module(String)
~ bind() : void
# onBind() : void
~ dismiss() : void
# onDismiss() : void
# isInUse() : boolean
# mkRequest(Serializable...) : Request
Result
- result: Object
- exceptionNeedsThrowing : boolean
- hasReleased : boolean
- callbacks : List<OnResultReceivedListener>
~ Result()
+ get() : Object
+ addOnResultReceivedListener(OnResultReceivedListener) : void
~ set(Response) : void
0..*
1


Connection
~ isClosed: boolean
~ isBlocked : boolean
- PUBLICKEYFINGERPRINT : String
+ Connection(String)
# getInputStream() : InputStream
# getOutputStream() : OutputStream
# closeInner() : void
+ getPublicKeyFingerprint() : String
+ close() : void
+ block() : void
+ unblock() : void
+ isBlocked() : boolean
+ isClosed() : boolean
- send(byte[]) : void
+ sendRequest(Request) : void
+ sendResponse(Response) : void
+ turnToByteArray(Object) : byte[]
+ turnToObject(byte[]) : Object
RequestReceiver
- requestHandlers: ExecutorService
+ RequestReceiver()
+ handleRequest(Request, Connection)
Message
- VERSION: int
- LENGTH : int
- HEADER : byte[]
- PAYLOAD : byte[]
+ Message(byte[])
+ Message(int, byte[])
+ Message(byte[], byte[], byte[], byte[])
+ Message(byte[], byte[], byte[])
+ Message(int, byte[], byte[], byte[])
+ fromByteArray(byte[], byte[]) : Message
+ toByteArray() : byte[]
- makeHeader(int, int, boolean, boolean) : byte[]
Response
1
+ FLAGS : int
+ RESULT : Object
+ REQUEST : Request
+ FLAG_THROW_EXCEPTION : int
+ FLAG_INVALID_REQUEST : int
+ FLAG_UNAUTHORIZE_REQUEST : int
+ FLAG_TREATED_AS_SPAM : int
+ CLEAR : int
~ Response(Request)
~ Response(Request, Object)
~ Response(Request, Object, int)
+ isSet(int) : boolean
MessageReader
- VERSION: int
- LENGTH : int
- TYPE : int
- PAYLOAD : byte[]
+ MessageReader(byte[])
+ parse(InputStream) : Message
- parseHeader(InputStream) : void
- getBitBlocks(int[], InputStream) : int[]
ConnectionReader
- mr : MessageReader
+ ConnectionReader
+ run() : void
- safeGetInput() : Message
- saveConvertInput(byte[]) : Object