Inter-Application Communication

RPC

RPC (Remote Procedure Call) is an API that allows programs to call methods in other networks, mainly used in C/C++ codes. This method was the first of its kind to allow access to remote elements. To use RPC, multiple files must be specified:

date.x

This file specifies the number of the method and the number of the program which must be unique across the network.

```
// date.x - Specification of remote date and time service bindate() which returns the binary time and date (no args). This file is the input to rpcgen
program DATEPROG { // remote program name (not used)
  version DATEVERS { // declaration of program version number
    long BINDATE(void) = 1; // procedure number = 1
  } = 1; // definition of program version = 1
} = 0x3012225; // remote program number (must be unique)
```

dateproc.c

This is the file that contains the method that the client will need.

```
// dateproc.c - remote procedures; called by server stub
#include <stdio.h>
#include <stdlib.h>
#include <rpc/rpc.h>
#include "date.h"
/* return the binary date and time
##############################################################
In Linux: long * bindate_1_svc(void* arg1, struct svc_req *arg2) {
##############################################################
In Dec Unix (MacOS): long * bindate_1() {
##############################################################
*/
```

```
13
14 long * bindate_1_svc(void* arg1, struct svc_req *arg2) {
15   static long timeval; // must be static
16   timeval = time((long *) 0);
17   return (&timeval);
18 }
```

rdate.c

This is the client that will request the method from the RPC server

```
1  // rdate.c - client program for remote date service
2  #include <stdio.h>
3  #include <rpc/rpc.h>
4  #include <stdlib.h>
5  #include "date.h"
6  int main(int argc, char *argv[]) {
7    CLIENT *cl;
8    char *server;
9    long *lres;
10
11   if (argc != 2) {
12     fprintf(stderr, "usage: %s hostname\n", argv[0]);
13     exit(1);
14   }
15
16   server = argv[1];
17   // create client handle
18   if ((cl = clnt_create(server, DATEPROG, DATEVERS, "udp")) == NULL) {
19     // couldn't establish connection with server
20     printf("can't establish connection with host %s\n", server);
21     exit(2);
22   }
23
24   // first call the remote procedure bindate()
25   if (( lres = bindate_1(NULL, cl)) == NULL){
26     printf(" remote procedure bindate() failure\n");
27     exit(3);
28   }
29
30   printf("time on host %s = %ld\n", server, *lres);
31   clnt_destroy(cl); /* done with handle */
32   return 0;
33 }
```

To compile the code run the following

```
rpcgen date.x #Generate RPC files to be used by server
gcc -c date_clnt.c # RPC client
gcc -c date_svc.c  # RPC server
gcc -c dateproc.c  # method sender
gcc -c rdate.c     # method receiver
gcc -o client date_clnt.o rdate.o # link RPC client with method receiver (our client)
gcc -o server date_svc.o dateproc.o # link RPC server with method server (our server)
```

To test the code, the RPC registry needs to be running, for Archlinux:

```
systemctl start rpcbind
```

RMI

Following RPC, came RMI (Remote Method Invocation), this is an API that provides a mechanism to create distributed application in Java. The RMI allows an object to invoke methods on an object running in another JVM.

To implement the RMI, an interface needs to be created that will be implemented by both the server and the client:

RMIInterface

```
import java.rmi.Remote;
import java.rmi.RemoteException;

public interface RMIInterface extends Remote {

    public String helloTo(String name) throws RemoteException;

}
```

The server needs to extend the UniCastRemoteObject to be able to called remotely

Server

```
import java.rmi.Naming;
import java.rmi.RemoteException;
import java.rmi.server.UnicastRemoteObject;

public class ServerOperation extends UnicastRemoteObject implements RMIInterface{

    private static final long serialVersionUID = 1L;

```

```
    protected ServerOperation() throws RemoteException
        super();
    }

    @Override
    public String helloTo(String name) throws RemoteException{
        System.err.println(name + " is trying to contact!");
        return "Server says hello to " + name;
    }

    public static void main(String[] args){

        try {
            Naming.rebind("//localhost/MyServer", new ServerOperation());
            System.err.println("Server ready");
        } catch (Exception e) {
            System.err.println("Server exception: " + e.toString());
            e.printStackTrace();
        }
    }
}
```

Client

```
import java.net.MalformedURLException;
import java.rmi.Naming;
import java.rmi.NotBoundException;
import java.rmi.RemoteException;
import javax.swing.JOptionPane;

public class ClientOperation {

    private static RMIInterface look_up;

    public static void main(String[] args)
            throws MalformedURLException, RemoteException, NotBoundException {

        look_up = (RMIInterface) Naming.lookup("//localhost/MyServer");
        String txt = JOptionPane.showInputDialog("What is your name?");

        String response = look_up.helloTo(txt);
        JOptionPane.showMessageDialog(null, response);

    }

}
```

The example above sends the string collected from the client to server by calling the **helloTo** method. To run the example, make sure that the java files are compiled to the .class files, and navigate to that directory from the terminal and start the RMI registry as so:

```
rmiregistry
```

Then run the server class then the client class.

RabbitMQ

Message queues serve as a buffers for requests between applications. RabbitMQ is a messaging queue written in Erlang, a language created for scalable, distributed, fault-tolerant systems and soft-realtime systems. There are three categroies of realtime applications:

- Hard: missing a deadline is a total system failure
- Firm: infrequent deadline misses are tolerable, but may degrade the system's quality of service. The usefulness of a result is zero after its deadline]
- Soft: the usefulness of a result degrades after its deadline, thereby degrading the system's quality of service

RabbitMQ comes packaged with a plugin manager, for instance to enable the web interface

```
#: rabbitmq-plugins enable rabbitmq_management
```

The interface is now running at (http://localhost:15672), the default username and password are both *guest*.

Sender-Receiver

A basic setup is as follows:

Send

```
package RabbitMQ;

import com.rabbitmq.client.Channel;
import com.rabbitmq.client.Connection;
import com.rabbitmq.client.ConnectionFactory;

public class Send {

  private final static String QUEUE_NAME = "hello";

  public static void main(String[] argv) throws Exception {
    ConnectionFactory factory = new ConnectionFactory();
    factory.setHost("localhost");
    Connection connection = factory.newConnection();
    Channel channel = connection.createChannel();


    channel.queueDeclare(QUEUE_NAME, false, false, false, null);
```

```
18      String message = "Hello World!";
19      channel.basicPublish("", QUEUE_NAME, null, message.getBytes("UTF-8"));
20      System.out.println(" [x] Sent '" + message + "'");
21
22      channel.close();
23      connection.close();
24    }
25  }
```

Receive

```
1   package RabbitMQ;
2
3   import com.rabbitmq.client.*;
4
5   import java.io.IOException;
6
7   public class Recv {
8
9     private final static String QUEUE_NAME = "hello";
10
11    public static void main(String[] argv) throws Exception {
12      ConnectionFactory factory = new ConnectionFactory();
13      factory.setHost("localhost");
14      Connection connection = factory.newConnection();
15      Channel channel = connection.createChannel();
16
17      channel.queueDeclare(QUEUE_NAME, false, false, false, null);
18      System.out.println(" [*] Waiting for messages. To exit press CTRL+C");
19
20      Consumer consumer = new DefaultConsumer(channel) {
21        @Override
22        public void handleDelivery(String consumerTag, Envelope envelope,
    AMQP.BasicProperties properties, byte[] body)
23            throws IOException {
24          String message = new String(body, "UTF-8");
25          System.out.println(" [x] Received '" + message + "'");
26        }
27      };
28      channel.basicConsume(QUEUE_NAME, true, consumer);
29    }
30  }
```

Correlation ID

In the case that there are multiple clients connecting to the queue, an ID must be assigned to each request to ensure that each request is sent back to the correct client. This can be done through the use of a correlation ID:

Server

```
Consumer consumer = new DefaultConsumer(channel) {

  @Override
    public void handleDelivery(String consumerTag,
                               Envelope envelope,
                               AMQP.BasicProperties properties,
                               byte[] body) throws IOException {

        AMQP.BasicProperties replyProps = new AMQP.BasicProperties
                                .Builder()
                                .correlationId(properties.getCorrelationId())
                                .build();

        System.out.println("Responding to corrID: "+
                           properties.getCorrelationId());

        String response = "";

        try {
            String message = new String(body, "UTF-8");
            int n = Integer.parseInt(message);

            System.out.println(" [.] fib(" + message + ")");
            response += fib(n);
        } catch (RuntimeException e) {
            System.out.println(" [.] " + e.toString());
        } finally {
            channel.basicPublish("", properties.getReplyTo(), replyProps,
response.getBytes("UTF-8"));
            channel.basicAck(envelope.getDeliveryTag(), false);
         // RabbitMq consumer worker thread notifies the RPC server owner thread
            synchronized (this) {
                this.notify();
            }
        }
    }
};
```

Client

```
String corrId = UUID.randomUUID().toString();
System.out.println(corrId);
AMQP.BasicProperties props = new AMQP.BasicProperties
                                    .Builder()
                                    .correlationId(corrId)
                                    .replyTo(replyQueueName)
                                    .build();

channel.basicPublish("",
                     requestQueueName,
                     props,
                     message.getBytes("UTF-8"));

final BlockingQueue<String> response = new ArrayBlockingQueue<String>(1);
```

```
15
16  channel.basicConsume(replyQueueName, true, new DefaultConsumer(channel) {
17
18  @Override
19  public void handleDelivery(String consumerTag,
20                             Envelope envelope,
21                             AMQP.BasicProperties properties,
22                             byte[] body) throws IOException {
23
24      if (properties.getCorrelationId().equals(corrId)) {
25          response.offer(new String(body, "UTF-8"));
26      }
27   }
```

Acknowledgements

By default RabbitMQ leaves message acknowledgements to the developer to handle, this is to allow the server to remove the message from its queues once it is sure that it has been received and handled by the client. The following line of code handles that:

```
1  channel.basicAck(envelope.getDeliveryTag(), false);
```

If an acknowledgement has not been received it will stay in the server and will be sent again to the worker.

Publish-Subcribe

To send one message to multiple listening parties, a Publish-Subscribe model needs to be utilized.

Publisher

```
1  package PublishSubscribe;
2
3  import com.rabbitmq.client.BuiltinExchangeType;
4  import com.rabbitmq.client.ConnectionFactory;
5  import com.rabbitmq.client.Connection;
6  import com.rabbitmq.client.Channel;
7
8  public class EmitLog {
9
10   private static final String EXCHANGE_NAME = "logs";
11
12   public static void main(String[] argv) throws Exception {
13     ConnectionFactory factory = new ConnectionFactory();
14     factory.setHost("localhost");
15     Connection connection = factory.newConnection();
16     Channel channel = connection.createChannel();
```

```
17
18      channel.exchangeDeclare(EXCHANGE_NAME, BuiltinExchangeType.FANOUT);
19
20      String message = getMessage(argv);
21
22      channel.basicPublish(EXCHANGE_NAME, "", null, message.getBytes("UTF-8"));
23      System.out.println(" [x] Sent '" + message + "'");
24
25      channel.close();
26      connection.close();
27    }
28
29    private static String getMessage(String[] strings){
30      if (strings.length < 1)
31            return "info: Hello World!";
32      return joinStrings(strings, " ");
33    }
34
35    private static String joinStrings(String[] strings, String delimiter) {
36      int length = strings.length;
37      if (length == 0) return "";
38      StringBuilder words = new StringBuilder(strings[0]);
39      for (int i = 1; i < length; i++) {
40          words.append(delimiter).append(strings[i]);
41      }
42      return words.toString();
43    }
44  }
```

Subscriber

```
1   package PublishSubscribe;
2
3   import com.rabbitmq.client.*;
4
5   import java.io.IOException;
6
7   public class ReceiveLogs {
8     private static final String EXCHANGE_NAME = "logs";
9
10    public static void main(String[] argv) throws Exception {
11      ConnectionFactory factory = new ConnectionFactory();
12      factory.setHost("localhost");
13      Connection connection = factory.newConnection();
14      Channel channel = connection.createChannel();
15
16      channel.exchangeDeclare(EXCHANGE_NAME, BuiltinExchangeType.FANOUT);
17      String queueName = channel.queueDeclare().getQueue();
18      channel.queueBind(queueName, EXCHANGE_NAME, "");
19
20      System.out.println(" [*] Waiting for messages. To exit press CTRL+C");
21
22      Consumer consumer = new DefaultConsumer(channel) {
23        @Override
```

```
24        public void handleDelivery(String consumerTag, Envelope envelope,
25                                   AMQP.BasicProperties properties, byte[] body)
    throws IOException {
26          String message = new String(body, "UTF-8");
27          System.out.println(" [x] Received '" + message + "'");
28        }
29      };
30      channel.basicConsume(queueName, true, consumer);
31    }
32  }
```

Maven

Maven is a Java project management tool, used it to manage the dependencies of a project as well as generating a JAR from the project. There is a maven repository that contains a wide assortment of JARs.

Maven has 9 tools:

1. Clean: removes previously generated JARs
2. Validate: validate the project is correct and all necessary information is available, any missing JAR is downloaded from the repository
3. Compile: compile the source code of the project
4. Test: test the compiled source code using a suitable unit testing framework. These tests should not require the code be packaged or deployed
5. Package: take the compiled code and package it in its distributable format, such as a JAR.
6. Verify: run any checks on results of integration tests to ensure quality criteria are met
7. Install: install the package into the local repository, for use as a dependency in other projects locally
8. Site: generates project's site documentation
9. Deploy: done in the build environment, copies the final package to the remote repository for sharing with other developers and projects

When defining a Maven project in Eclipse/Intellij, a *GroupID* and *ArtificatID* needs to be chosen, these reference:

- GroupID: will identify the project uniquely across all projects, so we need to enforce a naming schema. It has to follow the package name rules, what that means it that it has to be at least as a controllable domain name control, multiple subgroups can be created

Example: org.apache.maven, guc.facebook, guc.facebook.chat-app

- ArtificatID: is the name of the JAR without version. If you created the JAR then any name can be chosen with only lowercase letters and no symbols. If it's a third party JAR use the name of the JAR as it's distributed.

Example: maven, commons-math, chat-app

After creating the project, there will be a POM file created, this is what maven uses to control the project. This file specifies, the Java version to use, the dependencies to add and how to structure the project.

```
1  <build>
2      <plugins>

3          <plugin>
```

```
            <groupId>org.apache.maven.plugins</groupId>
            <artifactId>maven-compiler-plugin</artifactId>
            <version>3.6.1</version>
            <configuration>
                <source>1.8</source>
                <target>1.8</target>
            </configuration>
        </plugin>
    </plugins>
</build>

<dependencies>
    <dependency>
        <groupId>org.mongodb</groupId>
        <artifactId>mongodb-driver-async</artifactId>
        <version>3.5.0</version>
    </dependency>

    <dependency>
        <groupId>org.mongodb</groupId>
        <artifactId>mongo-java-driver</artifactId>
        <version>3.5.0</version>
    </dependency>
</dependencies>
```

Runnable JARs

To produce runnable JARs in Maven, the *pom.xml* file needs to have the following plugin:

```

<plugin>
    <groupId>org.apache.maven.plugins</groupId>
    <artifactId>maven-shade-plugin</artifactId>
    <version>3.1.0</version>
    <executions>
        <execution>
            <phase>package</phase>
            <goals>
                <goal>shade</goal>
            </goals>
            <configuration>
                <transformers>
                    <transformer
implementation="org.apache.maven.plugins.shade.resource.ManifestResourceTransformer"
>
                        <mainClass>Main</mainClass>
                    </transformer>
                </transformers>
            </configuration>
        </execution>
    </executions>

</plugin>
```

The main class tag should be changed accordingly, the JAR can now be produced by running the *Package* life-cycle in Maven.