COMP 512 – Distributed Systems

Reservation System Implementation

Project Report

Joseph Vinish D'silva

&

Dhirendra Singh

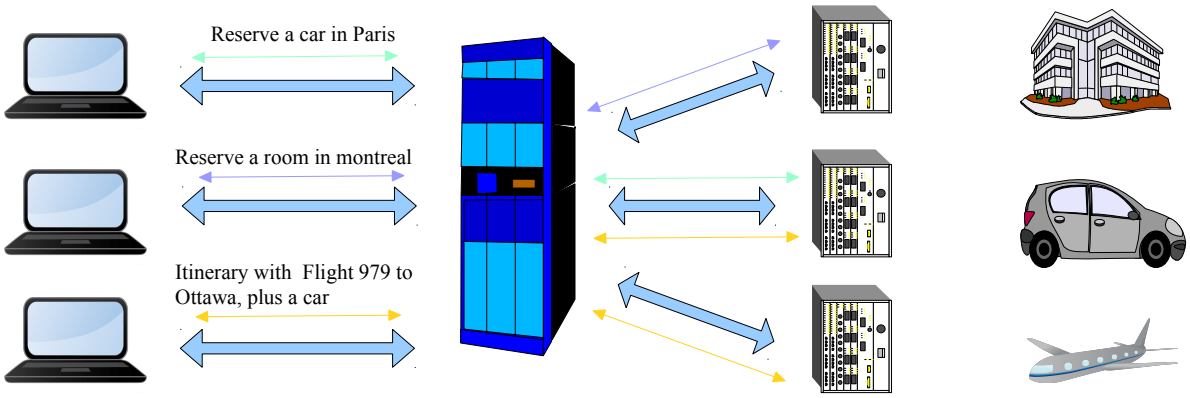
Table of Contents

Revision History:

Version	**Remarks**
1.0	Intial Project Report – 1

Three tier architecture of the reservation system.

The High level diagram of the 3 tier architecture of the reservation system is as depicted in Figure below.



Here the clients connect to a well known middleware layer, and is oblivious to the existence of multiple resource managers that actually implements and executes the various requests. Clients pass their requests to the middleware layer who executes the requests on the various reservation systems on behalf of the client.

The middleware layer in turn is aware of the various resource manager servers that implement different functionality. It knows that it needs to direct the hotel reservation requests to the hotel resource manager, car reservation request to car resource manager and so forth. It also takes care of any requests that spans across the three reservation systems. Hence its capable of looking at an itinerary request and splitting it down to flight reservation, car reservation and hotel room booking. The middleware layer also provides a consolidated view of the customer info, in terms of the person's billing by collecting the reservation information of a particular customer from all the three reservation systems.

Java RMI Implementation

In the Java RMI version of the implementation the client systems are aware only of the Middleware layer interface which provides an abstract view of all the reservation functionalities available in the system. Clients obtain a remote reference to the middleware layer object which has implemented the functionalities and manage their reservation requests through that object.

The middleware layer object knows about the three resource management servers and have access to interfaces that abstract their functionality. The middleware layer thus obtains remote references to these three objects and implements its own advertised middleware layer functionality by redirecting the requests to these three objects as needed and returning the output results to the client process.

The data of the system resides completely on the three resource management servers. The reservation system support concurrent execution, by virtue of java RMI system, and as such the data integrity mechanism is maintained by the synchronization of the access to data items in the resource manager servers. This is accomplished by making use of the synchronization support of Java, which enables thread safe components to be developed.

TCP Implementation

The TCP version of the reservation system was created by using a custom developed tool called charon. How charon works and the development methodology of using charon will be explained in the sections later.

In the TCP version, the remote object is implemented via an interface similar to the Java RMI implementation. However there are proxy classes which also implement the object interface to which the client has access to. These proxy classes are capable of opening network connections to the server side which has the real object implementation. The server process is also equipped with skeleton classes that services any request coming from the client side.

As in the case of Java RMI implementation, the client knows and have access only to the middleware layer via a middleware proxy object. The proxy object is in charge of communicating any client requests with the skeleton object residing on the middleware server. The skeleton object has access to the actual middleware implementation which then executes the requests on various reservation systems as is needed. In order to facilitate this remote execution, it makes use of the same proxy – skeleton architecture to interact with the resource manager objects residing in remote hosts.

Similar to the Java RMI implementation, charon's remote object management system is capable of supporting concurrent requests making it possible for multiple clients to connect and issue requests at the same time. Thus data integrity is ensured by using synchronization at the resource manager servers.

Design principles of Charon

The purpose of developing Charon was to implement an automated mechanism that, given an interface definition can provide a complete system that will generate not only a proxy class and a skeleton class which the client program and server program can correspondingly utilize, but also to supply a complete runtime support to loading those definitions and managing the objects transparently without having the programmer to explicitly write the functionality to do so. By achieving this we were able to reduce the effort to build a TCP based software to that of using corresponding Java RMI functionality.

In order to achieve complete automation, we built up on the powerful support that Java provides in reflection package both in terms of static interpretation of compiled class definitions as well as runtime loading of classes and object instantiations.

The starting point of Charon is a definition of an interface, which we are intending to develop for remote access. For example consider the case where the developer wants to build a centralized object which will provide a unique “sequence number” to various clients. This could be a case where in a distributed system architecture we are having a centralized controller to maintain ordering of messages across various processes.

We will define Sequencer.java which is the interface definition as follows.

```
package mydistpkg;

import charon.*;

public interface Sequencer extends charon.CharonRObject
{
  public int getNextSeq() throws CharonException;
}
```

Here charon.CharonRObject is an interface that all remote interfaces are required to extend. Also similar to Java's RMI requiring a RemoteException to be thrown for all methods implemented over RMI, CharonException is required to be thrown by all methods to be the part of remote interface.

This interface definition is next compiled. (make sure to point the class path to include charon.jar)

```
javac -cp ~/mytools/charon.jar mydistpkg/Sequencer.java
```

This will build Sequencer.class in the mydistpkg direstory.

Next we will execute Charon to build the proxy and skeleton classes for us.

```
java -cp ~/mytools/charon.jar:. charon.Charon mydistpkg.Sequencer
mydistpkg/
```

As always, ensure the classpath is correct so that java runtime can pickup both the charon package and the interface definition of remote object.

In the above command charon.Charon is the main of the process that will build both the Proxy and Skeleton classes of the mydistpkg.Sequencer interface definition. Next, mydistpkg/ is the directory into which we want the tool to write the Proxy and Skeleton class definitions. This is usually the same directory in which we have the interface definition.

You will now notice two new java files Sequencer_Proxy.java and Sequencer_Skel.java in the mydistpkg/ directory.

Though the programmer is not required to edit the Proxy and Skeleton classes, we will take a peek into some of the critical components in the definitions for these classes to better understand how Charon works internally.

```
package mydistpkg;

public class Sequencer_Proxy implements Sequencer,
charon.CharonROProxy
{
```

Notice how the original package name is retained. Also we can see that the Proxy class is implementing our interface definition Sequencer as well as a special interface charon.CharonROProxy which is implemented by all Proxy classes.

We also notice two static integers defined. These are used in the Proxy <==> Skeleton communication to pass execution status of remote methods.

```
private final static int CHARON_MTHD_EXEC_OK = 0;
private final static int CHARON_MTHD_EXEC_ERR = 1;
```

Take a look a the next integer definition. Charon assigns each and every function that it is exporting in a remote interface a unique identifier. It is this identifier that the Proxy class passes on to the Skeleton class to indicate which function it is invoking. This also helps support function polymorphism.

```
private final static int CHARON_MTHD_NUM_GETNEXTSEQ_0 = 10000;
```

We also see a different implementation of our interface function getNextSeq()

What we notice is that the Proxy function actually writes its method number into the ObjectOutputStream which sends it to the Skeleton Object. It next reads the status of execution from Skeleteon and if it's successful, returns the output returned by Skeleteon Object.

```
public int getNextSeq() throws charon.CharonException
{
try{
synchronized(this){
oos.writeInt(CHARON_MTHD_NUM_GETNEXTSEQ_0);
oos.flush();
int __status = ois.readInt();
if( __status !=  CHARON_MTHD_EXEC_OK) throw new
charon.CharonException("Error Remote method returned error during
execution.");
return ois.readInt();
}

}
catch(Exception e){ throw new charon.CharonException("An unexpected
error occured in proxy class method.", e); }

}
```

We also see that the entire communication code is in a synchronized block. This makes the Proxy Objects thread safe by ensuring that one Proxy object in a client does one communication at anytime, if it is shared between different threads with in the same client process. Note that this doesn't interfere with multiple clients talking to the remote object separately. You can even create multiple Proxy objects within the same client if its required to have different threads communicate simultaneously.

Let's now take a look into the Sequencer_Skel.java file

```
package mydistpkg;

public class Sequencer_Skel implements charon.CharonROSkel
{
```

Here we note that the class doesn't implement our original interface, but does implement a special CharonROSkel interface that's implemented by all Skeleton classes.

We also notice some integer variables that we are already familiar from the Proxy class definition.

```
private final static int CHARON_MTHD_EXEC_OK = 0;
```

```
private final static int CHARON_MTHD_EXEC_ERR = 1;

private final static int CHARON_MTHD_NUM_GETNEXTSEQ_0 = 10000;
```

Next take a look at the charon_execMethod()

This method is used to figure out which is the function that needs to be invoked by the Skeleton Object when it receives the request from the Proxy object. This is a function that all Skeleton Objects implements.

```
public void charon_execMethod(int method) throws
charon.CharonException
{
try{
switch((int)method) {
case CHARON_MTHD_NUM_GETNEXTSEQ_0:getNextSeq_0();break;
}

}
catch(Exception e){ throw new charon.CharonException("An unexpected
error occured in proxy class method. unable to establish
communication streams.", e); }
}
```

Also notice how the function definition is slightly different ... its named getNextSeq_0() and it takes no arguments and returns void. This is because all of the skeleton functions will read their arguments from the Object Input Stream connected to the Proxy Object that it is serving and write back any results to the stream.

Here's what will be inside the getNextSeq_0() function.

```
public  void  getNextSeq_0() throws charon.CharonException
{
try{
int __ret_arg = obj.getNextSeq();
oos.writeInt(CHARON_MTHD_EXEC_OK);
oos.writeInt(__ret_arg);
oos.flush();

}
catch(Exception e){ throw new charon.CharonException("An unexpected
error occured in proxy class method.", e); }

}
```

Notice the obj.getNextSeq() function call ? That's the actual implementation function being executed, and the result is being stored to __ret_arg which is then eventually written back to the Proxy via the Object stream.

Now let's build an implementation of the Sequencer interface. We will call it SequencerImpl.java .

```
package mydistpkg;

import charon.*;

public class SequencerImpl implements Sequencer
{
  int seqNum;

  public String charon_getRMOInterfaceName()
  { return "mydistpkg.Sequencer"; }

  public SequencerImpl()
  { seqNum = 10000; }

  public int getNextSeq() throws CharonException
  {
    synchronized(this) { return ++seqNum; }
  }
}
```

As can be seen, the source code of the object is very simple, just like an RMI object definition. Notice the synchronized block in the getNextSeq() function to provide thread safety to the seqNum counter so that two clients calling the function at the same time do not end up with the same sequence number.

There's a special function that all objects need to implement, charon_getRMOInterfaceName(). This is required for Charon's remote manager to understand which of the interfaces that the object is implementing (in case of multiple inheritance) that the Object is planning to export, so that the remote manager can load corresponding skeleton class to register it.

We need a server system that can now load the Sequencer Object and share it with the clients that needs to get a sequence number. Below is the code for ROServer.java

```
package mydistpkg;
```

```
import charon.*;

public class ROServer
{

  public static void main(String args[]) throws CharonException
  {
    CharonROManager grm = new CharonROManager(2099);
    Sequencer seq = new SequencerImpl();

    grm.registerObject("SeqObj", seq);
    grm.waitOnManager();
  }

}
```

Look how simple the source for server is. First we created a Remote Manager object which we attached to port 2099 of the host system. Next we created an instance of the SequencerImpl object and registered it to the remote manager system with a name SeqObj

The final line of the source grm.waitOnManager() is to ensure that the main process thread now goes idle and let the manager threads to do its jobs.

Here's the ROClient.java source , it shows how to get the next sequence from the object.

```
package mydistpkg;

import charon.*;

public class ROClient
{
  public static void main(String args[]) throws CharonException
  {

    Sequencer seqObj = (Sequencer)
CharonROManager.getRemoteObjectReference("localhost", 2099,
"SeqObj");

    System.out.println("Seq Num is : " + seqObj.getNextSeq());

  }
}
```

We use the call CharonROManager.getRemoteObjectReference(“localhost”, 2099, “SeqObj”) to indicate to charon where is the server that manages the object resides and what is the name of the object. Charon manages to contact the Remote Server, gets its interface definition, loads the corresponding Proxy class and returns the object.

We can compile all the sources using a simple javac command.

```
javac -cp ~/mytools/charon.jar  mydistpkg/*.java
```

To start the server.

```
java -cp ~/mytools/charon.jar:. mydistpkg.ROServer
```

And below is the multiple clients that's connecting to the remote object and obtaining a different sequence number.

```
java -cp ~/mytools/charon.jar:. mydistpkg.ROClient
Seq Num is : 10001

java -cp ~/mytools/charon.jar:. mydistpkg.ROClient
Seq Num is : 10002

java -cp ~/mytools/charon.jar:. mydistpkg.ROClient
Seq Num is : 10003
```

Internally when the server creates a new CharonROManager Object, it binds to the socket and starts a new thread in the background that listens for connections from clients. It also keeps track of all objects and names associated with it, that gets added to it by means of registerObject method via an internal map. Thus one can say that Charon's Proxy – Skeleton connections are maintained in a persistent manner.

When a client connects and requests for an Object by its name, it checks its object repository to see if an object by that name exists, in which case it loads the corresponding Skeleton class and instantiates a Skeleton Object to serve that particular client. It then notifies the client as to what Proxy Object it needs to create. A Skeleton object is created per client and is provided its own thread to run on. This ensures that each client has its own concurrent connection and experience no blocking.

When the client code invokes a function call on the Proxy object, the function passes the function number along with any of the arguments passed on to it to the Skeleton object. Skeleton object makes uses of the charon_execMethod() to redirect the processing to a corresponding skeleton function. This skeleton function unmarshals the arguments and invokes the function on the actual object by passing the arguments to it. Any return results are then captured into local variables which are then written back to the Proxy object which returns it to the client program.

As such it can be seen that Charon's implementation supports only passing by value of its

arguments and return values.

Testing the software system.

The same suite of test cases were executed on both the RMI version and the TCP version with comparable results.

Following is a sample of testcases from an elaborate suite of test cases that we performed.

1. Started rmi registry on each of the Htl, Flt, Car and Middleware servers, each of them running on different machines.
2. Started client from a remote machine.
3. Added 5 flights through the client.

```
newflight,1,3031,200,1200
newflight,1,3032,300,1100
newflight,1,3033,550,1000
newflight,1,3034,200,900
newflight,1,3035,500,800
```

The middleware passes the request to Htl Server, which adds the above 5 flights to its data structure.

2. Add 5 cars through the client.

```
newcar,1,crescent,10,50
newcar,1,mcgill,5,60
newcar,1,lessalle,3,70
newcar,1,gurudwara,3,80
newcar,1,atwater,5,90
```

The MWLayer passes this request to Car server, which adds these cars to its data structure.

3. Add 6 rooms to locations, such that the intersection of set of Car locations and Room locations is not empty set.

```
newroom,1,mcgill,1,105
newroom,1,lessalle,3,80
newroom,1,gurudwara,5,75
newroom,1,atwater,15,70
newroom,1,airport,9,65
newroom,1,f1circuit,2,60
```

4. Add 6 customers with their id through client.

```
newcustomerid,1,11
newcustomerid,1,12
newcustomerid,1,13
newcustomerid,1,14
newcustomerid,1,15
newcustomerid,1,16
```

The MWLayer passes this request to all the three servers.

5. Now we make some reservations, and do some queries to test the system

```
reserveflight,1,11,3031
```

=> Flight 3031 reserved for customer 11

```
itinerary,1,12,3031,3032,atwater,true,true
```

=>Flight 3031,3032, Car and room at atwater reserved for customer 12.

```
reservecar,1,13,crescent
```

=> Reserves car at crescent for customer 13.

```
reserveroom,1,14,mcgill
```

=>reserves room at McGill for customer 14

```
queryroom,1,mcgill
```

=> gives '0' as the result.

```
Reserveroom,1,15,mcgill
```

=>since 0 rooms left at McGill now, so returns 'Not reserved'.

```
Querycustomer,1,11
```

=> Customer id: 11
Customer info:Bill for customer 11
1 car-crescent $50

1 flight-3031 $1200