Foundations of Distributed Computing
Distributed Systems
COSC 1170/1197

Assignment 1

Remote Procedure Calling

Aims

The purpose of this assignment is to exemplify and explore some important issues of remote procedure calling. This assignment requires an understanding of the basics of remote procedure calling.

Method

Students will attempt this assignment individually.

Time frame

Time allocated for this project: 7 teaching weeks (7 calendar weeks)
Due date: 16th April 2014, 9:30 pm

Submission

What to submit

1. a clear and brief (less than one page) description of what you did, how your program works
2. a record of running your program, e.g. produced by the *script* command (for further reference see: *man script*), that includes the date and time when the program ran
3. the source code

Submission format

The preferred format is when each of the above listed modules is in a separate file. You can submit directories and subdirectories, but do not use more than 3 levels of depth in subdirectories. You may submit one archive with all your files (zip or tar).

Submission method

Submission is through WebLearn. All your files should be uploaded to WebLearn. A zip or tar file should contain all the aforementioned files. **No special consideration will be given to any student who has not used WebLearn properly**

Distributed Computing

In this assignment you will implement a simulation of a tram tracking system. You will produce a server component to track locations of trams and a client component that simulates trams.

The system consists of one server.

Part A: Tracking Service – Tracks locations of trams.

This server is a singleton.

Tracking Service

The tram stops in each route are hard coded on the server.

1=1,2,3,4,5
96=23,24,2,34,22
101=123,11,22,34,5,4,7
109=88,87,85,80,9,7,2,1
112=110,123,11,22,34,33,29,4

Route 1 consists of trams stops 1, 2, 3, 4 and 5, and route 96 consists of stops 23, 24, 2, 34 and 22.

This service exposes two functions

i. *retrieveNextStop()* – Accepts a route id, the current tram stop number, which is the last stop visited by the tram and the previous tram stop number. Returns the next stop in the route. If the current tram stop number is the last one, the next will be the previous one. This simulates the fact that a tram travels up and down a particular route. Only tram clients are allowed to access *retrieveNextStop()*. The server should use the current stop and the previous stop to identify the direction in which the tram is travelling.
ii. *updateTramLocation()* – Accepts a tram id and stop id, and updates the location details. Only tram clients are allowed to access *updateTramLocation()*.

Tram Clients

These clients are used to simulate actual trams. Tram clients continuously update the tracking service regarding their locations. Between each update request, each tram client sleeps for a time interval that randomly varies between 10 to 20 seconds. This simulates the time taken by a tram to travel from one stop to another.

Updating the location of a tram – Consists of two operations.

1. A tram client retrieves the next stop from the tracking service.
2. Then, it updates the next stop as the current location of the tram. A request is sent to the tracking service to perform this. The next stop and the current time should be printed on the client console before sending an update request to the tracking service.

Message structure definitions

The following message structures should be used.

Java definition

```
public class RPCMessage implements Serializable{
        public static final short REQUEST = 0;
        public static final short REPLY = 1;
        public enum MessageType{REQUEST, REPLY};
        private MessageType messageType;
        private long TransactionId        /* transaction id */
        private long RPCId;       /* Globally unique identifier */
        private long RequestId;/* Client request message counter */
        private short procedureId;        /* e.g.(1,2,3,4) */
        private String csv_data /* data as comma separated values*/
        private short status;
}
```

Format of csv_data

1. Requests sent from a tram client to the tracking server to invoke *retrieveNextStop()*.
   route id, current stop number, previous stop number

2. Response from the route server to a tram client when *retrieveNextStop()* is invoked.
   next stop number
   If the next stop number cannot be retrieved (i.e. parameters route id, current stop number and previous stop number are invalid) **-1** should be set to the next stop number.

3. Request from a tram client to the tracking server to invoke *updateTramLocation().*
   tram id, stop id

4. Response from the tracking server to a tram client when *updateTramLocation()* is invoked.
   csv_data is empty

Validations

Both clients and server should make use of the *messageType* field to test that requests and replies are not confused (i.e. validations should be performed at both clients and server).

Each transaction should have a unique *TransactionId*; and server should copy the *TransactionId* from the request to the reply so that it can be validated by the client.

Each client should generate a new *RPCId* for each call; and server should copy the RPCId from a request to the relevant reply. Clients should validate the *RPCId* in each reply. *RPCId* is a globally unique identifier, i.e. no two messages in the system should have the same *RPCId*, even if different hosts generate them.

Each client should also generate a new *RequestId* for each request; and the server should copy the *RequestId* from a request to the relevant reply. Clients should validate the *RequestId* in each reply. *RequestId* is used to keep track of the number of requests issued by a client, and each *RequestId* is unique within the issuing client.

ProcedureId is used to ensure that the request is handled by the appropriate remote method. This field should be validated; a) at the server before commencement of any processing, and b) at clients before processing any replies.

The status parameter is used to indicate the status of a transaction. 0 indicates that the transaction was completed successfully. A non-zero value indicates that there was an error at a server, and clients should display suitable error messages.

Only the number of characters indicated by the length field should be retrieved using *csv_data*.

System Properties and Limits

1. Each route will have an upper limit of 5 trams.
2. Each route will have minimum 5 stops.
3. Each trams stop should have a unique id.
4. Each tram should have a unique id.

Part B: Marshalling / Unmarshalling

Make a copy of your implementation so far. This exercise requires you to write your own marshalling and unmarshalling procedures, which will work on top of the marshalling (Serialization) procedures provided by Java RMI. That is, your server and client functions must be modified to pass objects of type Message over RMI.

```
import java.io.Serializable;

public class Message implements Serializable{
   protected byte data[] = null;
   protected int length = 0;

   public void marshal(RPCMessage rpcMessage){
   }

   public abstract RPCMessage unMarshal(){
   }
}
```

When marshalling, an integer should be stored in 4 bytes and a character should be stored in 2 bytes. The length field should be used to validate marshalled messages.

What to demonstrate

In your submission include a sample run with the following output: the result sent back to the client by the server that performs operations for several clients, and which behaves sensibly when dealing with exceptional conditions. (For recording a session, please see the *script* command under Unix.) No concurrency control needs to be implemented, so users can interfere with each other's operations in a way that leads to error conditions, as allowed by the Unix file system.

The record of a session should include date and time as outputted by Unix.

Provide the results (including source code) for both versions: with and without your own marshalling procedures.

The End