# Foundations of Distributed Computing
# Distributed Systems
# COSC 1170/1171/1195/1197

# Assignment 1

## Remote Procedure Calling

**Aims**

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

**Method**

Students will attempt this assignment individually.

**Time frame**

Time allocated for this project: 4 teaching weeks (5 calendar weeks)
Due date: 29th April 2013, 9:30 pm (week 8)

**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 and any corresponding *make* file.

**Submission format**

The preferred format is when each of the above listed modules is in a separate file, and everything put together in a zip archive. You can submit directories and subdirectories, but do not use more than 3 levels of depth in subdirectories.

**Submission method**

Submission is through *Weblearn*. All your files should be zipped together, and the zip file uploaded to *Weblearn*. **No special consideration will be given to any student who has not used *Weblearn* properly.**

# A REMOTE MESSAGING SERVICE

**TASK 1.**
In this exercise you will create a simplified version of a live messenger service. It delivers text messages from users to other users, provided both sender and receiver are connected to the system at that point in time. Message delivery is according to the *pull paradigm*, that is, messages have to be fetched by users; the service does not send a notification about the arrival of a message.
There are five operations that users can activate:

(i) connect to the system
(ii) disconnect

and for any connected user

(iii) deposit a message for another connected user
(iv) retrieve a message or check if there is any message for the user and
(v) check if a particular user is connected. For privacy reasons, e.g. to avoid spamming, you cannot obtain a list of all connected users. In order to send a message, you need to know the recipient's ID.

Your task is to implement the client and server modules. The client module takes input from the user, and provides responses. On one hand, it accepts messages and addressees, and forwards them to the service; on the other hand, it fetches messages from the server and displays them to the user. A client module need not handle multiple users with different IDs simultaneously. The sender's and recipient's IDs must be included in the message. The user IDs are integer numbers, the messages are in plain text format and can contain any printable character. You need to ensure that messages are not longer than the available buffer space. When a message is deposited, the server module records the current time together with the message. When retrieving a message, the server tells the client how long the message was waiting before being picked up.

Communication is via RPC. In case there is an error on the server side or on the client side, an error message will be displayed to the user and the current operation will be aborted. Client and server are as follows.

**Client:** It repeatedly requests a string to be entered by the user. Each line typed at the client is to be interpreted as a command, two user IDs (sender, receiver) and data if any, in this order. This requires the expression to be separated into different arguments having the type of string, non-negative integer or character.

**Server:** This must implement the five remote procedures *connect(), disconnect(), deposit(), retrieve()* and *inquire()*. They will take the input arguments, and perform the required operation.

## Message structure definitions

The following message structures should be used.

### *a) C definition*

```
typedef struct
  {       enum {Request, Reply} messageType; /* same size as an unsigned int */
          unsigned int TransactionId          /* transaction id */
```

```
        unsigned int RPCId;                /* Globally unique identifier */
        unsigned int RequestId;            /* Client request message counter */
        unsigned int procedureId;          /* e.g.(1,2,3,4) */
        char csv_data[]                    /* data as comma separated values*/
        unsigned int length;               /*length of data in cvs_data*/
        unsigned int status;               /*status of the transaction*/
    } RPCMessage;
```

***b) 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;
}
```

## 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 validate 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 from *csv_data*.

**System Properties and Limits**

1. Each message will have an upper limit of 100 characters.
2. At most 10 clients can be connected to the server at a time.

**TASK 2.**
Make a copy of your implementation so far. This exercise requires you to write your own marshalling and unmarshalling procedures, which will work <u>on top of the marshalling procedures provided by the programming environment.</u>

**If you write your program in *C***

Your own marshalling and unmarshalling procedures will work <u>on top of the marshalling procedures provided by Sun RPC.</u> That is, your server and client functions must be modified to pass the structure of type Message (as below) over RPC.

```
typedef struct
{
        char data[]     /* a sequence of bytes that holds the marshalled message */
        int length;     /* The number of bytes in the sequence*/
} Message;
                  or
typedef struct
{
        char data<>     /* a sequence of bytes that holds the marshalled message */
        int length;     /* The number of bytes in the sequence*/
} Message;
```

You must implement the functions:

```
void marshal(RPCmessage *rm, Message *message);
void unMarshal(RPCmessage *rm, Message *message);
```

The call *marshal(&RPCm, &m)* flattens *RPCm* into *m.data* and puts its length into *m.length.* Marshalling a string is straightforward. Marshalling an integer is also very simple - it will occupy 4 bytes of *m.data*. Note that you should take network ordering into account, and therefore should apply the function *htonl* to each integer (or unsigned integer) before flattening it.

*unMarshal(&RPCm, &m)* unflattens *m.data*, taking each successive 4 bytes as an integer belonging to the struct *RPCm*. To take network ordering into account, you should apply the function *ntohl* to each integer (or unsigned integer) after unflattening it. It must ensure that the lengths of the marshalled fields are correct.

**If you write your program in *Java***
Your own marshalling and unmarshalling procedures 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**