

CS 2510 Distributed Operating System

Report for
Simple Remote Procedure Call

Qiao Zhang

November 5, 2014

Overview

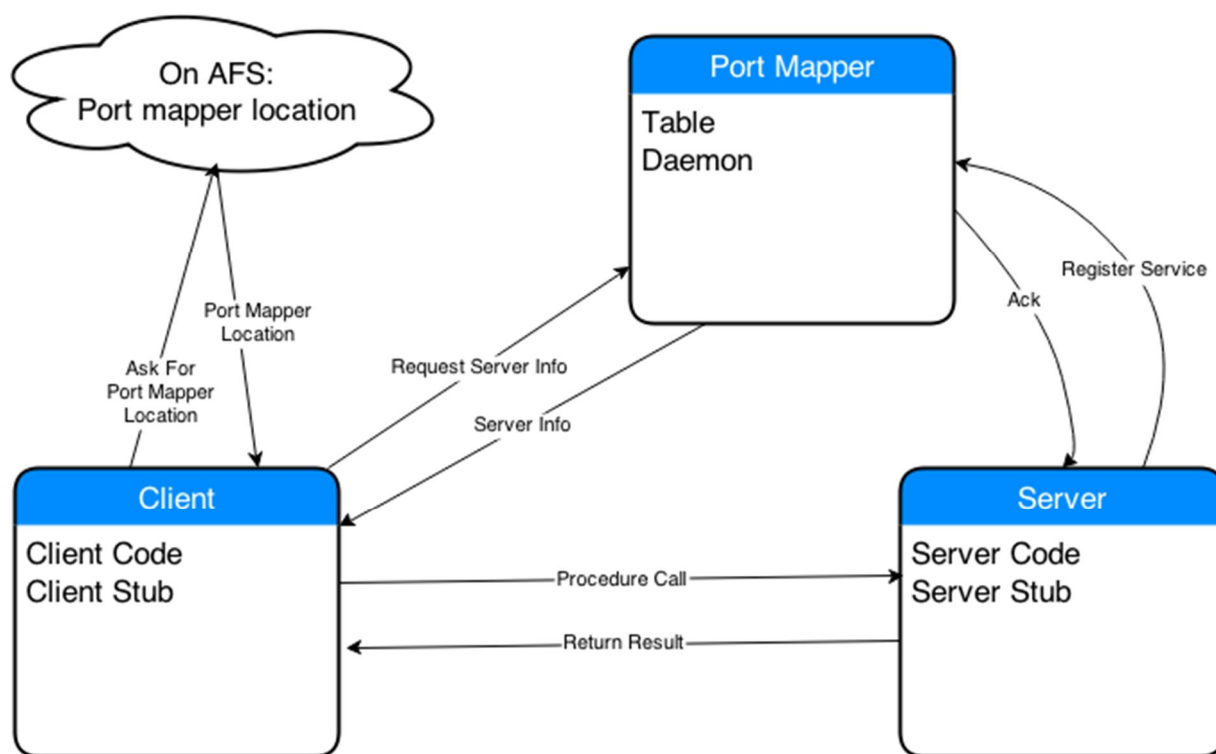


Fig 1. The RPC workflow

The program structure I use for SRPC is based on the concept of stubs. When making a remote call, five pieces of program are involved: the user, the user-stub, the SRPC communications package, the server-stub, and the server. The relationship is shown in Fig 1.

On the start of the port mapper, its location is written onto a “well-known” file. Then servers register their services to the port mapper by sending requests using TCP connections. When the client wishes to make a SRPC, it actually makes a local procedure call (LPC) which invokes a corresponding procedure in the user stub. To make a SRPC, the client stub first sends the port mapper a request of the service. After getting the service location from the port mapper, the client sends the server a “lookup” request through TCP connection to ask if the service is available or not. If the service is available, the server side will acknowledge that message and create a UDP connection with the client. On receiving the acknowledgement from the server, the user-stub will marshal the arguments and ask the SRPCRuntime to transmit these reliably to the server. I will use batch acknowledgement for transmission; the server will only acknowledge after it receives certain numbers of packets. After receiving all the data, the server-stub de-marshals the arguments and invokes the specific LPC and waits for the result. The

server-stub will then marshal the result and ask the SRPCRuntime to transmit the packets back to the suspended client. Received data is then de-marshaled and the user-stub returns them to the user. SRPCRuntime is responsible for retransmissions and acknowledgments etc. In this way, I provide transparency to the user-level applications.

Port mapper

As the server starts, it registers its services by sending register requests, and the port mapper maintains a table that keeps the information of the servers for clients' reference. When a client process wants to execute a SRPC, it blocks its process and sends the port mapper a request asking for the service location. The port mapper will return the corresponding IP address and port number of the required service.

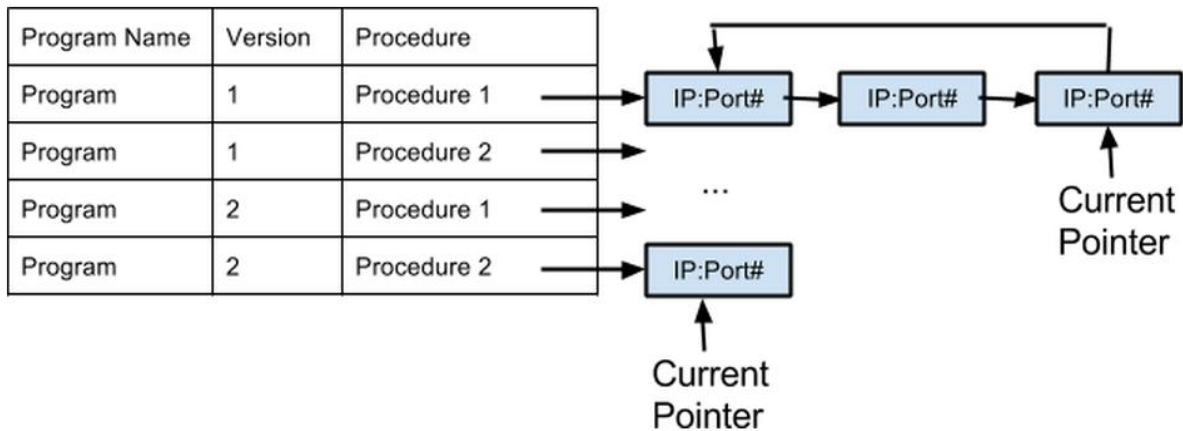


Fig 2. The port mapper table

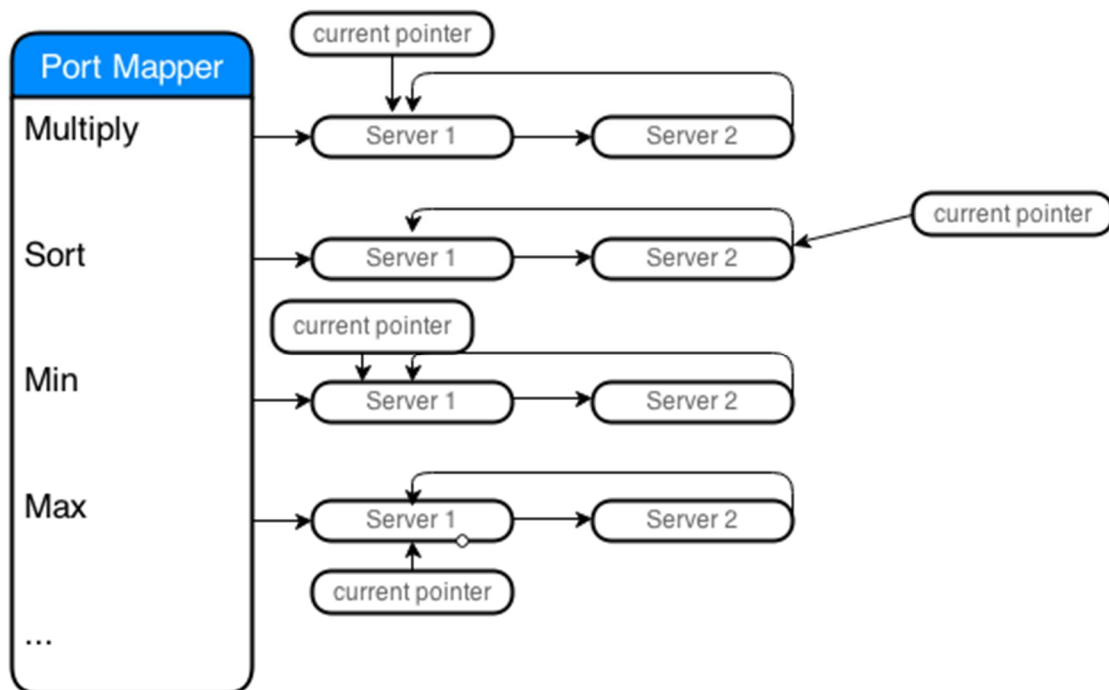


Fig 3. Port mapper load balancing scheme

The above diagram shows how port mapper maintains a table of all procedures that are available. Because the user might need an older version of procedure, the Port mapper will keep track of the program version too. Each entry is linked to a circular linked list of available services. I use round robin scheduling for each procedure. When a lookup request arrives, the server that the current pointer points to will be returned, and the pointer will be shifted to the next node. In this way, the clients can get the services they want without having to know the exact IP addresses and port numbers of the servers.

Client and Server

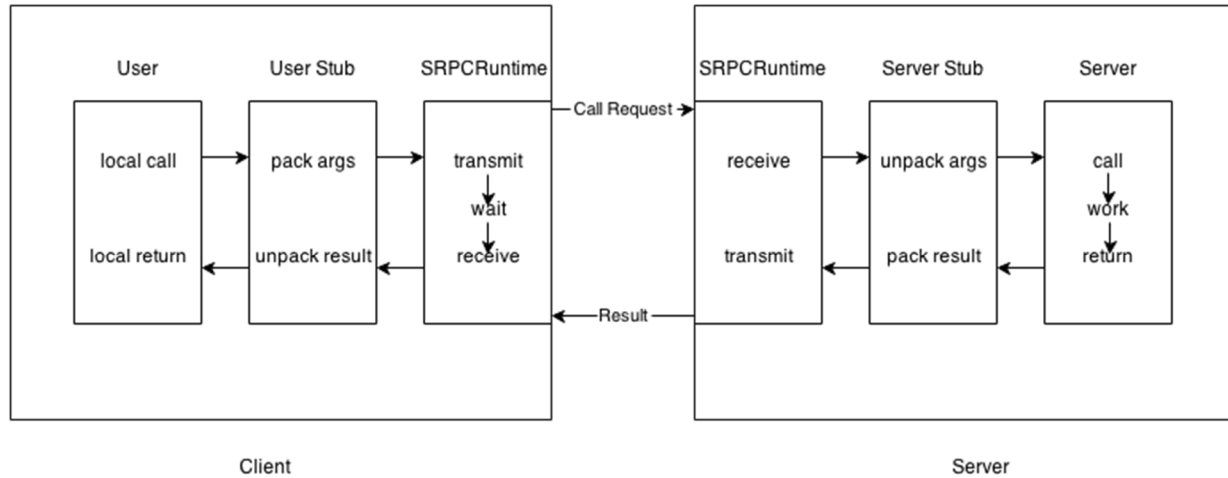


Fig 4. The communication between client and server

Before the client sends actual parameters towards the servers, it requires a handshake between the two parties. The client needs to send a service request packet to the server that contains:

- The requested service (procedure) name
- The number of parameters
- The dimensions of each parameter
- The dimensions of returned value

After the server acknowledging that the service is provide, the dimensions of parameters are correct, the server will terminate the TCP connection and establish a UDP connection with the client to transfer the actual data.

TCP Packets Definition

All TCP packets are in a generic format:

Packet Type (1 byte)	Packet Content (Non-deterministic Length)
----------------------	-------------------------------------------

There are in total 6 packet types,

Register service (from server to port mapper) (0)

Register acknowledge (from port mapper to server) (1)

Request server location (from client to portmapper) (2)

Response server location (from portmapper to server) (3)

Request service (from client to server) (4)

Response service request (from server to client) (5)

The packet content depends on the type of the packet. The actual packets are:

[Register service](#) (from server to port mapper)

Packet Type	Program Name	Program Version	Procedures	Location
-------------	--------------	-----------------	------------	----------

An example packet:

0	Math	1	Min\$Max\$Multiply\$Sort	150.212.2.52:61678
---	------	---	--------------------------	--------------------

[Register acknowledge](#) (from port mapper to server)

Packet Type	Program Name	Program Version
-------------	--------------	-----------------

An example packet:

1	Math	1
---	------	---

[Request server location](#) (from client to port mapper)

Packet Type	Program Name	Program Version	Procedure
-------------	--------------	-----------------	-----------

An example packet:

2	Math	1	Multiply
---	------	---	----------

Response server location (from port mapper to server)

Packet Type	Service Location
-------------	------------------

An example packet:

3	150.212.2.52:50359
---	--------------------

Request service (from client to server)

Packet Type	Program Name	Program Version	Number of Parameters	Parameter Dimensions	Return Value Dimensions
-------------	--------------	-----------------	----------------------	----------------------	-------------------------

An example packet:

4	Math	1	2	2x3#3x1	2x1
---	------	---	---	---------	-----

Response service request (from server to client)

Packet Type	Service Location
-------------	------------------

An example packet:

3	150.212.2.52:51228
---	--------------------

Notice I used special characters as separators for different sections of the packet content. Such that I can have more flexibilities for the content length.

UDP Packet Format Definition

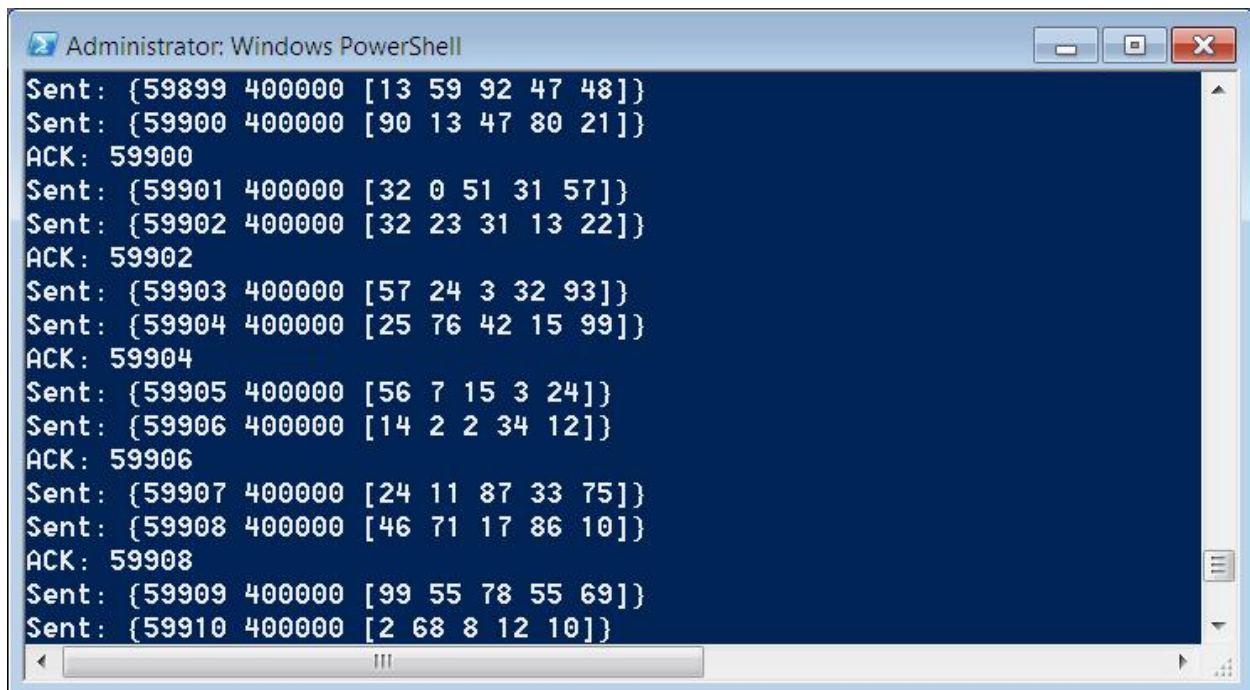
Since both client stub and server stub have a handshake before data transmission, they should already have reached the agreement on parameters and returned value. Hence, there is no need to specify anything related to application on this level.

The generic UDP packet is formatted as:

Packet Number	Total Packets	Data Slice (length specified by IDL)
---------------	---------------	--------------------------------------

Some example packets:

1	3	13 12 4 5 7
2	3	1 1 0 59 2
3	3	76 243 1 0 0



```
Administrator: Windows PowerShell
Sent: {59899 400000 [13 59 92 47 48]}
Sent: {59900 400000 [90 13 47 80 21]}
ACK: 59900
Sent: {59901 400000 [32 0 51 31 57]}
Sent: {59902 400000 [32 23 31 13 22]}
ACK: 59902
Sent: {59903 400000 [57 24 3 32 93]}
Sent: {59904 400000 [25 76 42 15 99]}
ACK: 59904
Sent: {59905 400000 [56 7 15 3 24]}
Sent: {59906 400000 [14 2 2 34 12]}
ACK: 59906
Sent: {59907 400000 [24 11 87 33 75]}
Sent: {59908 400000 [46 71 17 86 10]}
ACK: 59908
Sent: {59909 400000 [99 55 78 55 69]}
Sent: {59910 400000 [2 68 8 12 10]}
```

Example: Client Side

```
Administrator: Windows PowerShell
59387 400000 [54 30 91 74 50]
59388 400000 [48 38 46 40 81]
59389 400000 [45 63 33 22 34]
59390 400000 [92 34 52 78 9]
59391 400000 [88 85 7 77 93]
59392 400000 [98 74 41 31 36]
59393 400000 [87 10 12 29 38]
59394 400000 [33 14 15 88 45]
59395 400000 [87 34 75 61 96]
59396 400000 [39 16 71 40 89]
59397 400000 [99 0 82 42 31]
59398 400000 [61 36 0 62 79]
59399 400000 [45 99 18 25 51]
59400 400000 [73 23 95 48 8]
59401 400000 [50 44 57 88 80]
59402 400000 [6 66 9 6 67]
59403 400000 [7 40 92 32 34]
```

Example: Server Side

Data Marshaling and Demarshaling

To illustrate how data is transferred, let me make a simple example. Say a client requests to compute the matrix multiplication of two matrices:

$$\begin{bmatrix} 1 & 2 \\ 3 & 4 \end{bmatrix} * \begin{bmatrix} 2 & 3 \\ 4 & 5 \end{bmatrix}$$

The client stub first “flattens” the multi-dimensional arrays into a 1-dimensional array:

$$[1 \ 2 \ 3 \ 4 \ 2 \ 3 \ 4 \ 5]$$

The array is then sliced into several chunks, let’s say 3, in order to be packed into packets:

$$[1 \ 2 \ 3] \ [4 \ 2 \ 3] \ [4 \ 5]$$

Then the stub marshals the data into several packets with each packet containing 3 numbers.

The actual packets will be sent are:

1	3	1 2 3
2	3	4 2 3
3	3	4 5 0

The stub sends the packets in bulk and wait for acknowledgement after sending each bulk of packets. The size of the bulk should be specified by IDL, however here I just use a predefined constant. Let's say the server needs to acknowledge after receiving every two packets:

Client sends:

1	3	1 2 3
2	3	4 2 3

Server acknowledges:

2

The client checks if the acknowledgement is indeed the last packet it has sent, then continues:

3	3	4 5 0
---	---	-------

After transmission, the packets are first be concatenated using the packet number:

[1 2 3 4 2 3 4 5 0]

The parameters are reconstructed using the agreement reached at the TCP step, by knowing there are 2 parameters, with the first parameter be a 2*2 matrix and the second parameter 2*2 matrix:

$$\begin{bmatrix} 1 & 2 \\ 3 & 4 \end{bmatrix}, \begin{bmatrix} 2 & 3 \\ 4 & 5 \end{bmatrix}$$

After computation, the server sends the result back to the client using the exact same approach above.

Client Application

The client application consists of the following functionalities:

A command line interface that allows users to execute procedure calls using randomly generated data.

A help function that prints out how to use the program.

A Debug function that can request server info from port mapper for certain service.

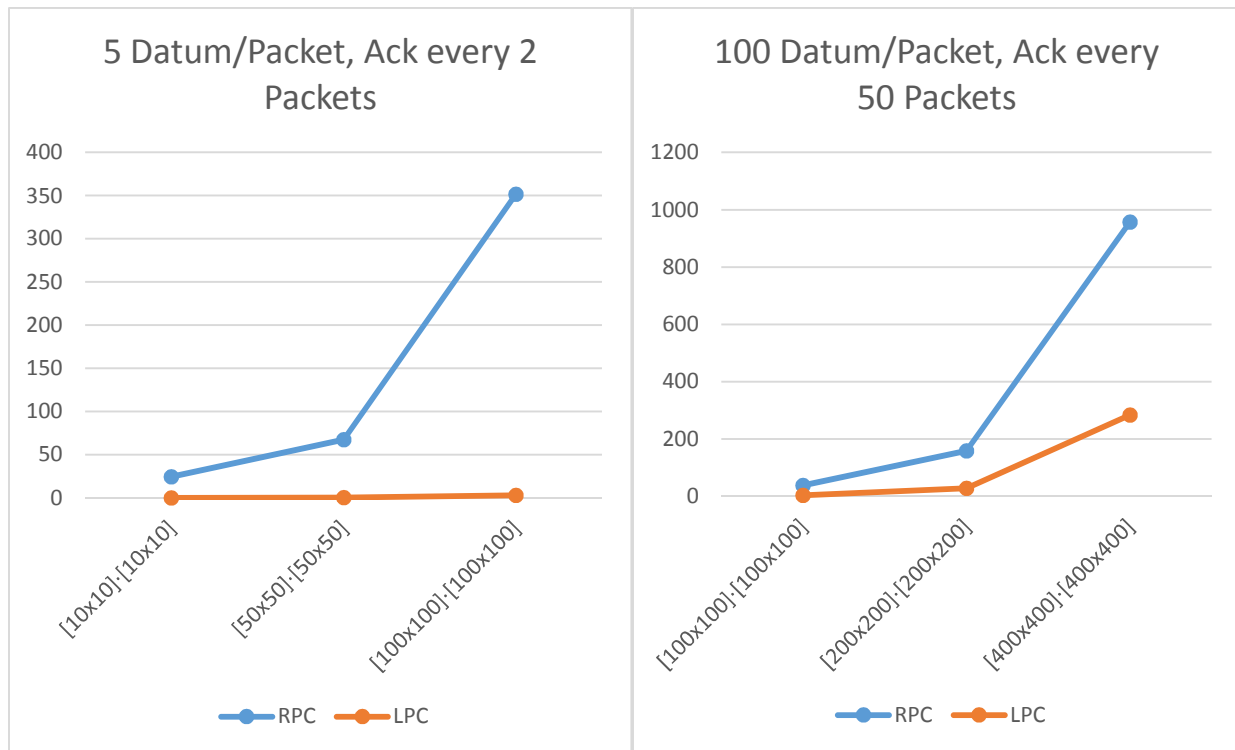
Server Service Implementation

The server support a simple scientific library of the following functionalities:

Due to time constraints, I only implemented non-optimized brute-force solutions for each function.

Analysis

I did two experiments under different settings. The raw data is in the ANALYSIS.txt file.



Each point is the average of 5 replicas. The y-axis is in MS (milliseconds).

As can be seen from the chart above, RPC indeed perform worse than LPC. The main reasons are that (1) LPC needs to encode data; (2) Data transmission due to network latency. Although my RPC performs significantly worse than the LPC, the overall time complexity is acceptable; To get the result of the multiplication of two 400 by 400 matrices, the client only needs to wait for 1 second on average.

Design Decision and Future Work

I choose synchronous model, and choose synchronous point at when the procedure computation result is transferred back to the client.

I choose UDP/IP as underlying protocol for bulk data transmission between client and server, TCP/IP as underlying protocol for packet transmission with the port mapper and the handshake before bulk data transmission between the client and server.

I choose NOT to cache server location in client and do load balance globally.

I choose use version number explicitly in both client and server protocols and allow server to add or withdraw functionalities from port mapper via a version update.

I only support passing value parameters and do not support function override.

I support multiple dimension arrays.

I choose NOT to support failure handling. I found that during packet transmission, it is very common to have packet loss. However, I did not deal with this problem due to time constraint.

I choose NOT to support heartbeat packets between server and port mapper due to time constraint.

Reference

- [1] A. D. Birrell and B.J. Nelson, "Implementing Remote Procedure Calls", ACM Transactions on Computer Systems, Vol. 2, No. 1, February 1984, Pages 39-59.
- [2] A. D. FreeBSD Software, "rpcgen Programming Guide", FreeBSD Documentation Server Server.
- [3] A. S. Tanenbaum and M. Van Steen, Distributed Systems - Principles and Paradigms, 2nd Edition, Prentice Hall, 2006.
- [4] G. Coulouris, J. Dollimore, T. Kindberg and G. Blair, Distributed Systems - Concepts and Design, 5th Edition, Addison Isley, 2012.
- [5] Technical Standard- DCE 1.1 Remote Procedure Call, The Open Group.
- [5] Wikipedia, "Interface description language",
http://en.wikipedia.org/wiki/Interface_description_language