

Project 3: MultiRPC using XML-RPC

Instructor: Professor Karsten Schwan(schwan@cc) TA: Minsung Jang (minsung@gatech)

1. Outline

The goal of the project is to understand the MultiRPC idea and then implement it. This project can be done by teams of 2 students each, and as usual, you are allowed to discuss your ideas with other teams, but sharing or copying from others is **NOT** allowed. Please refer to the Georgia Tech honor code available at <http://www.honor.gatech.edu/>.

Turn in your final deliverables for the project on **April 8, 2012 by 11:59pm on the T-square** website. Emails or other methods for submission will not be accepted.

2. Goal

You are to add the MultiRPC idea, which is explored in the Coda paper, to an existing RPC mechanism. The RPC mechanism in this case is XML-RPC, which uses XML to encode remote procedure calls. You are to implement MultiRPC with both asynchronous and synchronous semantics using XML-RPC. The RPCs should include the ability to specify whether the caller is interested in all responses, any responses, or a majority of responses to be gathered before returning.

To receive full credit for this project you must implement the following:

- MultiRPC calls in the XML-RPC library
- RPC Semantics as described below
- Service & Client as described and their execution on the Factor clusters
- A write up discussing your design and the performance of your RPC mechanism
- Adherence to the submission practices (!)

3. Details

3-1. XML-RPC Overview

You will first need to download the XML-RPC package. You should first check out the XML-RPC web site. The web site (<http://xmlrpc-c.sourceforge.net/>) has information on what XML-RPC is and how to use it. Next, you should download latest **stable** version (1.16.40) of XML-RPC. You should then install XML-RPC locally without requiring a root privilege. You can do this as follows:

```
./configure --prefix=installation_base_directory && make && make install
```

An example of the installation on the Factors clusters:

```
[mjang9@factor040 ~]$ ls
xmlrpc-c-1.16.40  xmlrpc-c-1.16.40.tgz
[mjang9@factor040 ~]$ cd xmlrpc-c-1.16.40
[mjang9@factor040 xmlrpc-c-1.16.40]$ ./configure --prefix=/nethome/mjang9/xmlrpc &&
make && make install
checking whether build environment is sane... yes
checking whether make sets $(MAKE)... yes
checking for working aclocal... found
checking for working autoconf... found
checking for working automake... found
```

```
checking for working autoheader... found
checking for working makeinfo... found
checking build system type... x86_64-unknown-linux-gnu
...
/bin/sh /nethome/mjang9/xmlrpc-c-1.16.40/mkinstalldirs /nethome/mjang9/include/xmlrpc-
c
/nethome/mjang9/xmlrpc-c-1.16.40/install-sh -c -m 755 xmlrpc-c-config
/nethome/mjang9/bin/xmlrpc-c-config
[mjang9@factor040 xmlrpc-c-1.16.40]$ ls /nethome/mjang9/xmlrpc
bin include lib
```

Your XML-RPC library, include files, and binaries will all be installed in:
installation_base_directory/lib, /include, and /bin as you can see in the example.
There are examples under the XML-RPC's examples directory that are also built.

3-2. Asynchronous MultiRPC

First, you will need to implement an asynchronous call to multiple URLs for your MultiRPC implementation. Your function should be a replacement for the *xmlrpc_client_call_async* function located in the `src/xmlrpc_client_global.c` file. Your asynchronous RPC will need to include a variable number of servers as input. The asynchronous RPC must wait until all responses are received before invoking the callback handler.

3-3. Synchronous MultiRPC

Next, you will need to implement a synchronous MultiRPC mechanism. Like the asynchronous RPC, you should add in a variable number of server URLs to be accepted by the *xmlrpc_client_call* function. The *xmlrpc_client_call* function is also located in `xmlrpc_client_global.c`. The synchronous MultiRPC mechanism should block until all of its responses are received before allowing the caller to continue.

3-4. Any, Majority, All Semantics

You should implement semantics that determine on how many responses the RPC should wait before the call is considered complete. The first such semantic is the **ANY** response, which means that the RPC waits for the first response before continuing. The second such semantic is the **MAJORITY** response, where the RPC call only waits for a majority of responses before continuing. Note that this is important in distributed services, where a majority of calls with the same response is needed. The third semantic is **ALL** where your call must wait for all the responses before continuing. Because of the service to be implemented (as below), you have to wait on a majority of the responses to be the same. If in the majority semantic, you do not have a major number of responses correct, then you must return the RPC as failed. In the all semantic, all of our responses should have the same values, else report the RPC as failed.

3-5. A Sample Implementation

Finally, you will build a simple service and client using your MultiRPC library, which should cover asynchronous and synchronous implementations. You are required to implement a service, which returns a result of the toss of a coin. Your client should generate a request for the toss and send it to, at least, three deferent services. In project 2, you implemented a server and client communication using the shared memory mechanism and the idea is to now use the RPC abstraction for that communication.

1. Service Descriptions

- 1-1. Once receiving a request from a client, the service returns either “Head” or “Tail” randomly. (Hint: recall `/dev/urandom` in Project 2)

- 1-2. You can use different sockets when creating multiple servers
- 1-3. You need not have multithread version of the server but it may be recommended to handle multiple RPC simultaneously
- 1-4. The service must be started as "\$./bin/service num_of_services start" and stopped as "\$./bin/service stop"

2. Client Descriptions

- 2-1. Your client makes a request and send it to N servers, where $N \geq 3$ (odd numbers only)
- 2-2. The client makes both sync and asynch calls for the MultiRPC with all semantics. Your MultiRPC waits responses from services based on a semantic given by the client
- 2-3. The client receives returns from servers and then prints out the toss result based on each semantic. That is, the client shows results based on semantics of all, any, and majority, respectively. As an example, three servers give the client back two heads and a tail. If the tail comes first, it prints out "Tail" for the "any" and "Head" for the "majority" while "RPC fail" for the "all."
- 2-4. It must be run as "\$./bin/client num_of_requests" (Do not consider multithreaded clients)

4. Directions

A demo to the TA for this project is not required, but may be requested when necessary. For full credit, therefore, please use the following guidelines for your project:

- We will execute your project automatically in the following sequence:

```
$make clean
$make

$./bin/service 3 start
/* There are three services running now */

$./bin/client 1000    /* Flip a coin 1000 times */
... results ...

$./bin/service stop
```

- Each client should produce output on the screen in the following format :

```
semantic|head|tail|RPC-error|sync/async|Ave. res. time in "ms"
where 1: all, 2: any, 3: majority for semantic
      1: sync, 2: async for call-type

(example)
$client 1000 [Enter]
1|190|210|600|1|20
2|496|504|0|1|10.1
3|481|519|0|1|20.2
1|349|351|300|2|15.2
2|490|510|0|2|5.5
3|411|589|0|2|14.8
/* The output on a screen must adhere to the given format. Extra
output will result in deduction of your marks */
```

- Any deviation from this will result in deduction of points. From this project, the professor as well as the T.A. will see if your submission adheres to the guidelines.
- The “stop” command must terminate any/all of your background service processes. We will deduct points if they do not
- Make sure that your submission runs on a user privilege. If it demands a “root” access during its installation and execution, it will not get graded on any Factor nodes at all (because the TA does not have the access either), which could seriously harm your points.
- Instructions for use of the Factor clusters are the same as those of Project 2.

5. Project Report



You need to prepare a detail write-up that describes how you implemented the MultiRPC. Also, provide plots of the following:

- Average response time for RPC for each semantic with increasing number of servers ($N \leq 11$)
- Average response time for each client as the number of requests increase ($reqs \leq 1000$, and the number of servers are constant)

6. Submission

Include all source codes including Makefile, etc., in the top (and same) directory, compress them into a ZIP file ONLY, and then upload that on T-Square. Submit the report on T-Square as a separate entry. Please see the figure below.

Submitted Attachments

-  [project3 report minsung.pdf](#) (200 KB)
-  [project3 minsung.zip](#) (13 KB)

[Back to list](#)

7. Hints and Resources

The figure below explains how the client in the project interacts with services.

