Project 3 Report and Summary

Daniel Hunnicutt, Bryan Hickerson

Building and Running the Code

First to configure xmlrpc for a new machine, do ./configure and make from the project3 root folder. This will configure xml rpc and compile all the code for the project.

To Run Servers:

From the /examples folder, run separate instances of ./xmlrpc_sample_add_server *server_port* where server_port is a command argument. Use 8080, 8081, and 8082 for the ports on the 3 instances.

To Run the Client:

From the /examples folder, run ./xmlrpc_client [*server_urls*] *response_semantic synchronous firstSumValue secondSumValue*. Here are the valid arguments for each:

[*server_urls*] = "http://localhost:8080/RPC2" "http://localhost:8081/RPC2" "http://localhost:8082/RPC2"

response_semantic = 0 (Majority) || 1 (Any) || 2 (All)

synchronous = 0 (Asynchronous) || 1 (Synchronous)

firstSumValue, SecondSumValue = integers to be added together by the rpc

Implementation Summary

Multiple Servers for one RPC

To accomplish this, we opted to utilize the asynchronous capabilities of xmlrpc for all of your rpc calls. This allows us to make multiple requests to various servers and handle then client side in a variety of ways. Even our synchronous implementation makes asynchronous xmlrpc calls, but simply blocks until a response condition is satisfied.

Synchronous vs. Asynchronous

The flow of our program is as follows: we first determine based on the command argument provided whether we want a synchronous call or asynchronous call. If it is asynchronous, we simply take advantage of pthreads and launch a new pthread to make our asynchronous request. Because our call is on a completely different thread, our main thread is still able to do work. Each request is actual made up of multiple requests (one request for each server) and is handled by an asynchronous callback handler. Once a response is received, this callback handler is responsible for determining when our response semantic is satisfied and then invokes the original caller via pthread unlock that the request is ready for processing.

In the synchronous case, the only difference is that we do not launch a new thread for each call like we did asynchronously, and we instead block via a pthread mutex lock and wait. We do still launch a thread for each

individual server request. The callback function will then handle the response and determine if our response semantic is complete and unlock the main thread for processing.

Response Semantics

We implemented 3 response semantics. ANY is the simplest case. This condition is satisfied when the first response from a server is received. ALL requires all servers to report back before our response semantic is completed. MAJORITY means more than half the servers responded to our request. Note than none of these semantics require the responses to be the same therefore there is no guarantee that our response is correct. Another example of a MAJORITY semantic is one where the majority of the servers return the same value, this offers a little more reliability.

Summary

We were able to implement all of the requirements in the project with client modifications. The server was untouched which is an implicit requirement of the project. The whole idea behind multirpc is querying multiple servers at once. There are no known bugs at this time with our implementation.