CS-6210 Project 3 Report: MultiRPC using XML-RPC

Vinay Bharadwaj (902825919)

Building and running the code:

To build and run the code, unzip the archive and from the xmlrpc folder, run make. This builds the library and all the files necessary to run the code. Then run ./bin/service start <num_servers> to start the MultiRPC service followed by ./bin/client <num_requests> to run the client.

The client makes requests using both synchronous and asynchronous calls with ALL, MAJORITY and ANY semantics and displays the result in the specified pattern.

MULTIRPC Service:

The service starts the specified number of servers simultaneously and binds them to individual ports. Thereafter, the servers stay bound to the ports and continue listening on their ports until the service is stopped. Once the service is stopped, all the servers are killed.

To implement the MultiRPC service, I make use of the asynchronous capabilities of xmlrpc for all the client calls. This allows us to make multiple requests to all servers simultaneously and handle the responses at the client side. Even synchronous resuests make use of the asynchronous calls but simply block until the call has completed and a response condition has been satisfied.

Synchronous and Asynchronous requests:

The project requires us to implement both Synchronous and Asynchronous requests. To make asynchronous requests, I simply launch a new pthread to handle each request and join on those threads from the main thread. Each asynchronous request thread in turn spawns new threads (one for each server) and the responses are handled by a callback handler. The callback handler is responsible for determining when a response semantic has been satisfied. When the response semantic has been satisfied, it signals the parent thread using the pthread_cond_signal(). Since we are creating a separate thread to perform the asynchronous request, the main thread is free to perform other work.

For synchronous requests, I do not create a separate thread as in asynchronous, but instead block using a pthread_mutex and wait till the call has completed. The callback function determines whether a response semantic has been satisfied and if so, unlocks the main thread.

Response Semantics:

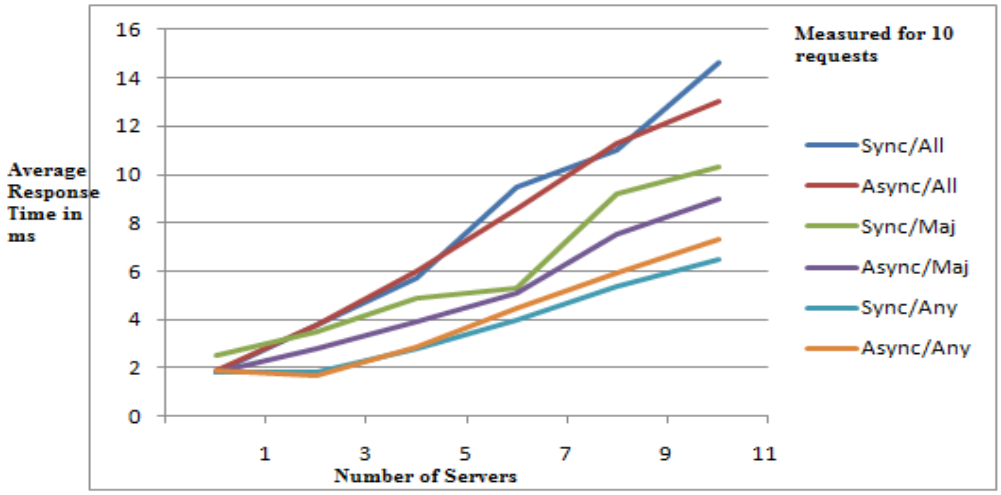
As per the requirements of the project, all three semantics – ALL, MAJORITY and ANY semantics have been implemented. The callback handler is responsible for determining whether the specified response semantic has been satisfied. If so, it sends a pthread_cond_signal() to the parent thread to indicate that the semantic has been satisfied. The ALL semantic blocks until responses from all servers has been received, the MAJORITY semantic blocks until majority of the servers have replied and the ANY semantic blocks until any one of the servers reply.

Server:

The server is a simple server, not so different from the ones implemented in the XML-RPC examples. However, instead of adding two numbers (as in the example server), my server just returns a random number (0 for head or 1 for tail). The server is not multithreaded.

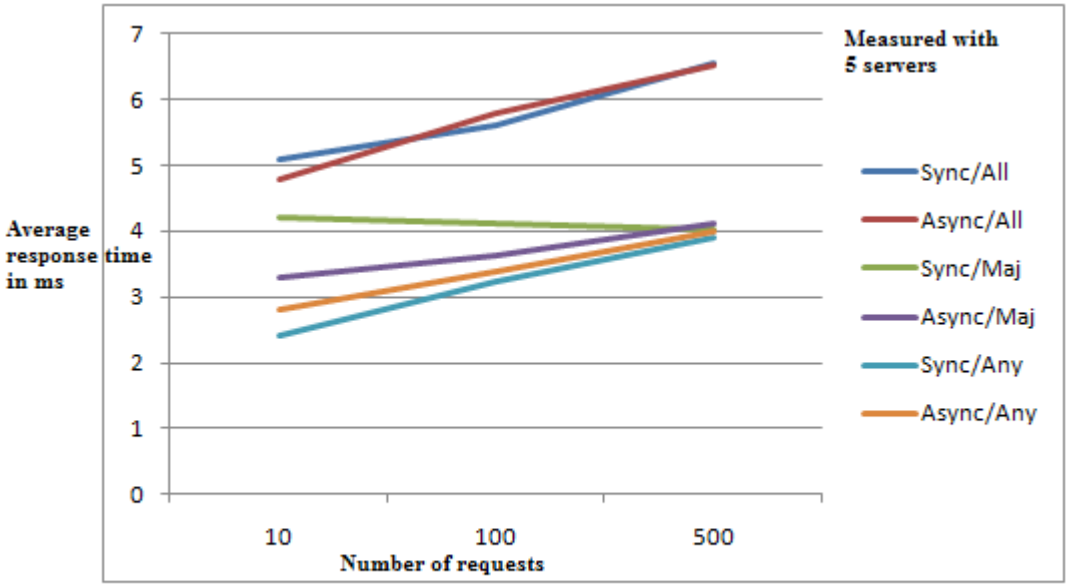
Results:

1) Average Response Time (in ms) with varying number of servers and constant number of requests:



As we can see in the above graph, the trend is that in general, synchronous requests take longer to complete. This is intuitively correct because synchronous requests block until the response semantic is satisfied. As a result, the main thread cannot invoke another synchronous request until the previous request has completed. In contrast, asynchronous requests do not block and as a result, the main thread can simultaneously launch new requests while previous request is being serviced.

2)Average Response Time (in ms) with constant number of servers and varying number of requests:



In the above graph, the trend is that as the number of requests increase, the average response time increases too which is again intuitively correct because it takes the servers longer to service the requests. Another important phenomenon we observe in the graph is that requests with ALL semantics take the longest to execute and have higher response time followed by requests with MAJORITY semantics followed by requests with ANY semantics. This is again intuitively correct as ALL semantic waits for responses from all the servers (thus takes the longest time to complete) followed by MAJORITY and ANY takes the least time because it waits only for the first response.

Summary:

In summary, I have implemented all the requirements required by the project. In my observations, the XML-RPC library is still buggy and sometimes not compatible with latest transport libraries. For instance, the library was not compatible with the latest libwww transport library but worked well with the curl library.