Designing a Web Server

Noah Nsangou, Geoffrey Lu
Williams College
nnn1@williams.edu, jgl3@williams.edu

1 Introduction

With this project, we are tasked to develop the front-end and back-end infrastructure for a very small bookstore with big ambitions. We take care to design a solution including a client-facing prompt with which a customer can search the bookstore's inventory and make a purchase, as well as an internal command input allowing for easy restocking, logging and updating the bookstore database. To actualize these specifications, we implemented a server with functionality to interact with the database. Such a design presents an interesting problem for students learning about distributed systems, as this calls for a means through which server and client can communicate. We provide this functionality using the remote procedure call with XML-RPC, which provides useful abstraction and automated communication management. In this report, we delve into the architecture and performance of our solution and reflect on the benefits of remote procedure calls.

2 Architectural Overview

Our solution consists of three major components: server, database, and clients. Our database, managed with sqlite 3, is only directly read or mutated by the server. The client makes remote procedure calls to methods implemented in the server which are exposed to the client. This design, with the inclusion of remote procedure calls, greatly simplifies the complexity of both client and server code, as remote procedure calls allow for clients to call functions remotely almost as if it were a simple local call. Instead of having to manage argument passing on both client and server, XML-RPC does this and more of the heavy-lifting. When a remote procedure call is made by the client, XML-RPC manages a client stub which takes and marshals the arguments, transmits the requests via RPC protocol, and blocks and waits for a return after the server does the same within its server stub, calls the requisite local methods and subsequently re-marshals and transmits the return over the same protocol back to the client. Thus, little work must be done and virtually any type of value or standard object can be easily passed between server and client.

We made the design decision to use XML-RPC rather than Java RMI. RMI is very similar to XML-RPC, allowing for facile calling of remote methods but in an object-oriented fashion. While Java RMI is in many cases simpler and allows for less disruption to local code, it sacrifices flexibility as it can be more difficult to interact easily with non-Java clients. In our particular use case, we were required to implement both a Java and Python client, so for this reason it was clear that XML-RPC was the appropriate choice.

This design allows the database to be securely hidden from the client. A solution in which each client implements database-interactive code rather than going through a server is extremely impractical and constitutes poor design. Our design as implemented allows for easy maintenance and modification to server-database interaction, as well as great portability as XML-RPC affords easy communication across different platforms. Having simple server-side code also affords easy and safe handling of concurrency.

3 Evaluation

The average response time per client search request is 0.003s for both 1, 2, and 3 clients accessing the server concurrently. For the buy request, the average response time for a client search request is 0.008s when 1 client is accessing, 0.012s when 2 access concurrently, and 0.020s when 3 clients access concurrently. While the average response time for the search request stays stable with increasing number of concurrent requests, the response time for the buy request increases significantly, almost doubling when 2 to 3 clients access simultaneously. This difference in increase is likely due to how the buy request needs to update the database, which likely invokes a locking mechanism that prevents clients from simultaneously making changes to the data, whereas the search request only makes a read query that the database can deliver to many different clients simultaneously.

4 Conclusion

The tiny bookstore we have developed comprises of a front end server that manages a database, as well as clients that can make queries to that database via the server. The server we have created effectively supports search, lookup, and buy queries on a SQLite database representing the stock of books in the store from the client-side, while also allowing the front-end of the server to restock and update prices of items in the database. Additionally, the front-end server maintains a log of past buy requests that can be accessed from the front-end of the server. The architecture of the system relies heavily on XML-RPC, which handles the creation of the server as well as the transmission of queries using the correct RPC protocols to the clients, instances of which we have implemented in both Java and Python. The benefit to XML-RPC becomes thusly apparent, as it wraps the functionality of the database into a standardized protocol that functions across different platforms. In operation, the client-side requests are dispatched efficiently and consistently when reads occur, even with multiple concurrent clients, although the buy-requests take more time when concurrent calls are made due to the locking mechanism of the database that prevents different clients from simultaneously updating the data. Overall, the tiny bookstore works well within its limited scope, and could truly become something big one day.