Tiny Book Store: The Smallest of Book Stores

Sam Wischnewsky, Sean Fontellio, Dysron Marshall
Williams College

1 Introduction

The purpose of this project is to support the basic needs of a simple online web store. Our architecture links a persistent data store with a backend server that accepts clients over XMLRPC. Using Java, we created a server which provides a means of interacting with the database. The core functionalities of the project involves searching, looking up, and buying books in the web store. XMLRPC allows the web store to abstract incoming data so that clients can be created in any language and perform the same actions. Essentially, the client makes requests through the command line which are then translated into SQL statements that either update or return data from the web store. We assume that the three commands referenced are the only supported actions along with an exit command to close the connection to the sqlite database.

2 Design

2.1 TinyBookServer.java

The server holds onto a static connection that is used by all its instances. There are three main services provided by the server and there are matching functions for them. The functions are search, lookup, and buy.

1. Search
   The search function takes in a topic which will be used as the query for the SQL statement created. It then searches for all books in the database with a TOPIC parameter matching the input and returns an array of strings containing books with a matching topic.

2. Lookup
   The lookup function takes an item number as input in the form of a String. That item number is then converted into an Integer and an SQL statement is made which selects all the books in the web store with an ID. Since, the IDs are unique and primary keys for the database entry matching the input, there will only be one item returned. The return value is a string describing the item's information (id, title, topic, stock, price).

3. Buy
   The buy function also takes an item number as input in the form of a String. It then converts the item number to an Integer. Buy first checks the database to see if the item exists and, if it does, decrements the count to the current stock minus 1 or 0 if the stock is 1. And if the action was successful, the item and the time at which it was purchased are logged. In all cases, the client is prompted with feedback regarding the success of their request with the correct response if it was successful.

2.2 TinyClient.java

The client simply establishes a connection with the server and listens to all input from the command line. After the user enters a command, it is parsed in order to determine if the 4 commands supported are used and the rest of the request is not malformed. Making the actual request is as simple as adding the request parameters to a vector and making a request through XMLRPC with the vector, now converted to an array, as an argument. The server's result is printed to the console and the client may continue to make more requests provided he does not exit or receive an error.

3 Thought Questions

3.1 500 Subsequent Buys

```
import xmlrpclib, sys, time
from numpy import mean
name = "http://"+sys.argv[1]+":8888"

server = xmlrpclib.Server(name)

diffs = []
for i in range(500):
        start = time.time()
        server.service.buy("12365")
        end = time.time()
        diffs.append(end-start)

print "Average response time was "
+ str(sum(diffs)/float(len(diffs)))
```

Our first experiment was performing 500 sequential buys. We performed a buy and monitored the response time using the python time package. The client code is shown above. In our tests we ran the client on Reina and connected it to Rathi. After checking 500 buys we found that the average response time was 0.0128 seconds. This seemed extremely fast and oddly was actually faster than when we ran the test locally on Rathi. Maybe Rathi is just real slow at running through the hosts.txt file.

3.2 500 Subsequent Searches

```
import xmlrpclib, sys, time
from numpy import mean
name = "http://"+sys.argv[1]+":8888"

server = xmlrpclib.Server(name)

diffs = []
for i in range(500):
        start = time.time()
        server.service.search(
        "Distributed Systems")
        end = time.time()
        diffs.append(end-start)

print "Average response time was "
+ str(sum(diffs)/float(len(diffs)))
```

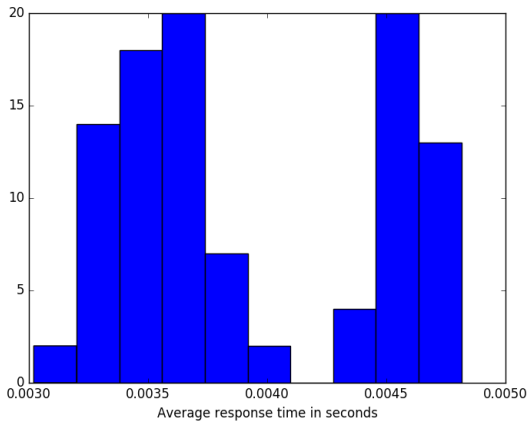
This experiment tested 500 sequential search operations on our server. We once again monitored the average time using the mini client shown above to connect Reina to Rathi. In practice our average response time was 0.00273 seconds. That was blazing fast compared to the buy response time. This is easily explained by the amount of work occurring on the back end. Search is only performing a read operation as we select the row with corresponding topic string. We then just do some string parsing and return to the client. In the case of our buy operation we have to do a SELECT followed by an update of the right row. We can't perform the update directly because we first need to verify that the stock is a positive integer otherwise we can't let buy be successful.

These two operations end up being about 6 times as slow on average. This could be because writes are fairly expensive.

3.3 Multiple Clients

```
import xmlrpclib, sys, time
from numpy import mean
from threading import Thread
import multiprocessing

name = "http://"+sys.argv[1]+":8888"

server = xmlrpclib.Server(name)

def thread_job():
  diffs = []
  for i in range(500):
    start = time.time()
    server.service.search
        ("College Life")
    end = time.time()
    diffs.append(end-start)

  print "Average response time was "
  + str(sum(diffs)/float(len(diffs)))

p_list = list()

for i in range(100):
  process = multiprocessing.Process
      (target = thread_job, args=())
  p_list.append(process)
  process.start()

for p in p_list:
  p.join()
```

We tested having concurrent clients by using processes in python2. They are fairly heavyweight in practice but very lightweight to implement. After running the above program we noticed that the average values were four to five times higher than we observed in a single client test. So we added functionality to our program that produced the histogram shown below.



This data set is bimodal. The first mode is around .0035 and the second is around .0045; both of which are greater than what we previously observed in a single client test. There is reasonable spread but generally very few clients are getting served as quickly as we previously saw in the single client test. Thus we can handle concurrency but there is a noticeable time trade off to doing it.

4 Conclusion:

In conclusion, this lab was quite fun. It was interesting working in three different languages and having them all interact with each other.

This lab was not especially difficult because each source of the assignment had a clear role: the database needed to pretty much only store the information that we needed to access; the clients essentially repeated the same actions as each other- only buy, lookup, and search for things through communication with the server; and the server (perhaps the most difficult) interacted with both components above, but really only needed to do prints and up-

dates after establishing connections with xmlrpc.

It was fascinating to see the time taken to execute multiple consecutive commands decrease when performed on different machines.

XMLRPC was also a really useful abstraction. When it is implemented in libraries within a language it's incredible how easily we can integrate remote procedure calls into a program. For example, the syntax differences between python and Java were honestly negligible. It was also nice that we didn't actually have to see any XML. It is all handled by the underlying library so we don't actually have to deal with any implementation portions to support XML-RPC. This is exactly what a well thought out protocol can accomplish.

This lab was useful in that we got to implement something more practical; a tiny bookstore is feasible and it shows us how basic structures like libraries and even more popular websites can be created and utilized with even our knowledge of computer science. Furthermore, we appreciated the opportunity to take what we learned in class and apply it to a lab in a way that built off the templates given to us, but also left room for innovation and creativity.

The ability to see how a python client and a java client can have such different amounts of code, but do the same processes exemplifies how certain design choices of a language can impact convenience and usability (i'm trying to say something else here, but i can't think of words). Maybe, it would have been cool to see how things might have differed were we to code the database, server, or client in other languages.

We were also able to recognize similarities between this lab and the last project. Thanks to the last lab (and some applicable experience from previous courses), we knew how to work with the localhost on the computers and were comfortable talking across the platforms whilst we experimented with procedure calls from this language version similar to html: xml.

We didn't really have many difficulties other than small annoyances of how to exit a program in the three different languages.

Therefore, this lab did as it was designed to do: raise our awareness of the network and connections across a server, database, and clients; all while expanding on our knowledge of distributed systems.

5 Sample Client Usage

Launched with python tiny_client.py localhost
Launched with java TinyClient localhost
usage: lookup item number buy item number search topic exit
Lookup 53477
Buy 53477
Lookup 53477
Search Distributed Systems
exit