Description of System:

The server that all the other pieces are based on, or talk to, is the catalog server. The underlying data structure behind our Tiny Bookstore's catalog is a TreeMap with Integers, representing the book's ID, as the keys and Books as the values. We could have used a HashMap to represent the catalog, but a TreeMap felt like a better alternative just because all the books are ordered by their IDs. The Book class is, for the most part, a simple data structure for representing one book in the bookstore. It contains methods to pack arrays as Books and to unpack Books as arrays. Our reason for representing a book as an array of type string is because an object of the user-defined Book class can not be transmitted over XMLRPC. XMLRPC supports only a handful of types, and in order to comply with this requirement, we had to use arrays to represent books. In order to transmit a book, we had to pack it as an array such that the first index of the array represents the book ID, the second index represents the book title, and so on. Then, upon the reception of the "book", the client side had to repack the array as a Book object. We were initially planning to implement the representation of multiple books as 2-dimensional arrays, where every row of the array would correspond to a book. This turned out to be more complicated than expected, so we decided to implement the representation of multiple books as a long one-dimensional array, with the first five indexes representing the first book, and the next five representing the next book, and so on. The trade-offs that we had to consider for this alternate representation included the fact that representing multiple books as a one-dimensional array looked like a less elegant design decision but was much easier to implement.

With XMLRPC comes the ability to have multiple connections, each running on their own thread. This could potentially cause issues in buying books; for example, the case where there was only one more book available and two clients were trying to buy the book at the same time. You wouldn't want both clients to be able to purchase the book. So, we implemented synchronization by using the Java *synchronized* keyword on our book catalog. We initially were planning on synchronizing on the CatalogServer class itself using the *this* keyword. This did not work out for us because we were renewing the stock automatically at set intervals from the main method, which is a static context. As such, we were not allowed to synchronize on the *this* keyword from a static context. We synchronize in two places: when we renew the book's stock automatically in the catalog server every minute and when we a processing a buy request and decrementing the stock of a book. This ensures that if a client thread is buying a book, the catalog is going to be locked for any other transactions, including stock renewal. Similarly, when the stock is being renewed automatically, we lock the catalog so that buy requests cannot be processed at the same time. This ensures that our system is properly synchronized.

Assumptions Made and Problems:

One assumption that we made in this project was that OrderServer and FrontEndServer were, in addition to being servers, supposed to have some client functionality implemented as well. We didn't know how to get OrderServer and FrontEndServer to talk to CatalogServer without also treating them as clients at the same time. Our entire bookstore functions as expected in doing it this way, because we were unsure if there was a more elegant/appropriate way to get the servers to talk to each other. The main concern here was that in object oriented programming, a class is supposed to do just one function. A OrderServer and a FrontEnd Server are supposed to be exactly that: servers, and not also clients. So, in the end, it would have been more ideal if we had found out some other way to get the servers to talk to each other. That said, at this point we are not convinced that a better solutions exists to begin with, and our implementation of treating servers as clients as well ended up working very well.

Testing:

For the preliminary testing of the servers, we modified the client.java file included with the project and used it to test connectivity between servers. At this point, our main concern was having the servers talk to each other and transmit a simple string from the back-end servers to the front-end server, and then to the client. After successful preliminary testing of the servers, we wrote a Python client for the Tiny Bookstore. We used the Python client to thoroughly test functionality of the servers. We tried multiple buy, lookup, and search requests. We tried buying or looking up items that did not exist or were out of stock to ensure that the client did not crash. We did not think that JUnit test cases were appropriate for this project, so our testing was done entirely manually by consider a number of scenarios.

To test that our system is properly synchronized, for our worst-case scenario, we tried having two clients concurrently attempt to buy a book with only 1 in stock. In our test, we observed that one client successfully managed to buy the book, while the other got an error (and the program did not crash or behave in an unexpected way).

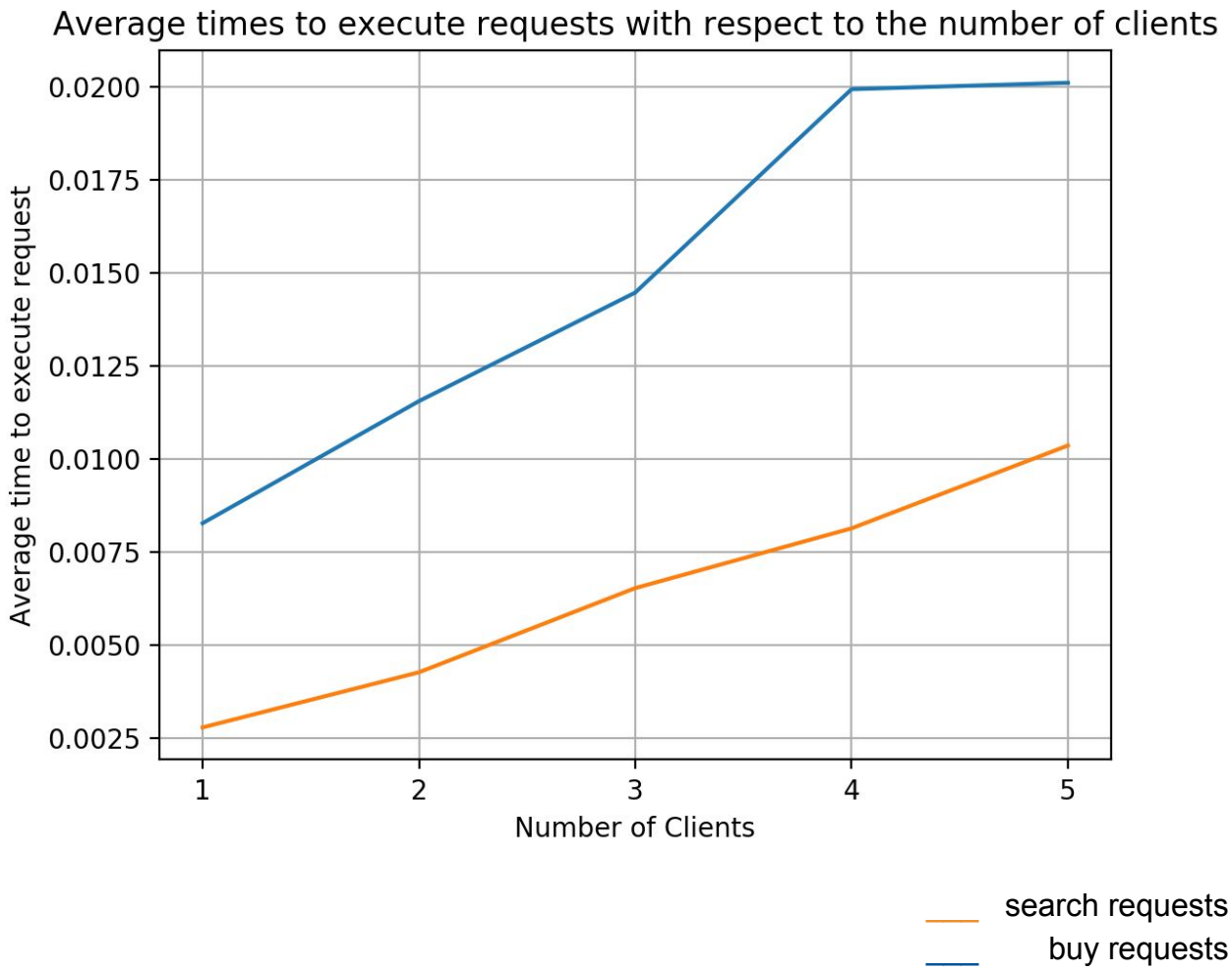
After our testing, we made a design decision to have the client catch exceptions whenever needed, but print out a user-friendly message for the client (instead of the exception itself). This is because we believe that the user does not need to know what kind of exception was raised; the only relevant information for the user is the fact that something went wrong.

Experimental Setup and Graphs:

We wrote Python scripts to compute the average response time per client request by measuring the response time seen by a client for 500 sequential requests. At this point, we realized would need to modify our existing Python client to a Client class

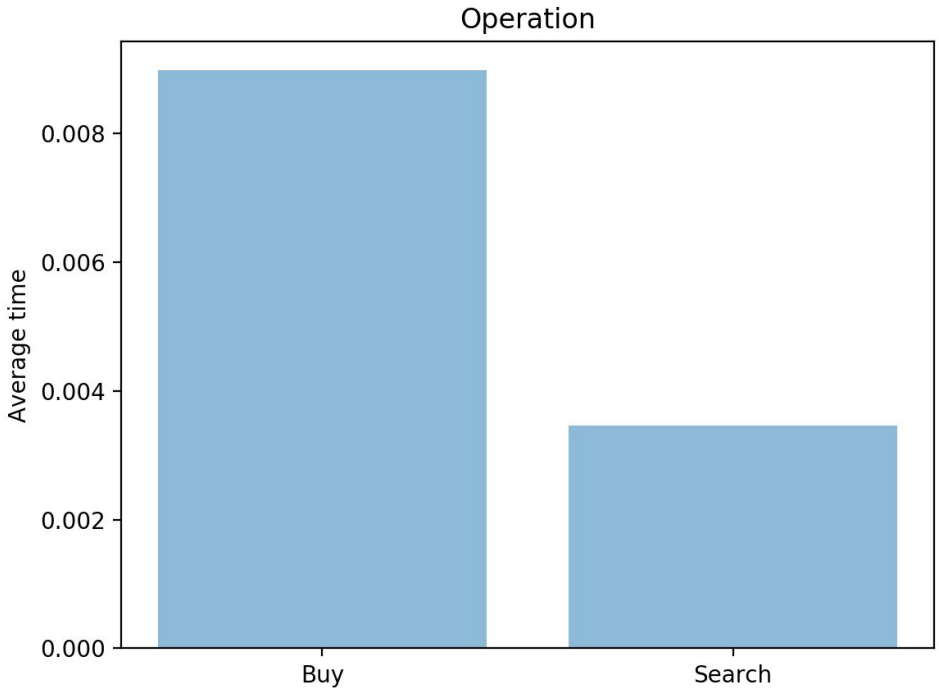
in order for the scripts to evaluate client functionality by instantiating Client objects. For client search requests, we fixed the search keyword to be “distributed systems” throughout the experiments. For client buy requests, we fixed the book ID to be “54377”. The primary reason behind this is consistency across experiments. We did not want the book ID or search title to be variables in our experiments and result in inaccuracy.

In order to test with multiple clients making requests concurrently to the servers, we decided to implement a threaded client that keeps an instance of a client for itself, among other instance variables. We tested with an increasing number of clients making 500 sequential buy or search requests, from 1 client to 5 clients, and then computed the average time per client for the requests to complete. The results are shown in the graph below:



We observed that as the number of clients increased, the time taken per client to complete 500 sequential search requests increased almost linearly. We also noted that the time taken per client to complete 500 sequential buy requests increased linearly *at first* with increasing number of clients, and then tended to level out. We repeated the experiments and observed similar results. We could not figure out why the graph for buy requests tends to level out as the number of clients increases. We thought that this might be just an inaccurate experiment run, but re-runs yielded similar results (though the graphs never leveled out to be completely flat eventually). One thing to note about our experimental setup is that due to the way we designed our threaded client, we had to run the threadedClient script multiple times, incrementing the number of clients every time the script was run. We manually recorded the most commonly occurring average times, and used these times in a separate Python script to generate the graph.

Additionally, we also observed that the time taken to complete 500 buy requests is significantly higher than the time taken to complete 500 search requests, regardless of the number of clients involved. We think that part of the reason behind this could be the fact that for a buy operation, there are more steps involved than search operations, i.e. the item needs to be looked up, the stock level needs to be checked, and if the item is in stock, the stock level needs to be decremented by 1--as opposed to simply looking up the item in search requests. This difference in the time taken by one client to complete 500 buy vs. search requests is shown in the bar chart below:



As is evident in the bar chart, buy requests tend to take more than twice as long to complete than do search requests.

We tried to limit the number of applications running in the background when running our experiments, so that excessive load on the processor does not cause a variance in our results or interfere with our results. We also ran the experiments multiple times. We observed that every experiment run resulted in a slightly different average time--we attribute this difference to the random nature of the experiments.