TinyBookstore.com: Designing a Multi-Tier Distributed System

Michelle Wang and Priya Rajbhandary

CSCI 339: Distributed Systems

Professor Jeannie Albrecht

Tuesday, October 22

Introduction

We have designed TinyBookstore.com, which is the World's smallest online bookstore. This bookstore is a multi-tier distributed system with a front-end server that accepts client requests and a back-end database to store book information. It employs remote procedure calls from both the client-side and server-side in different programming languages, as well as interacting with, querying, and updating the database.

System Architecture

TinyBookstore.com currently carries four books:

1. Achieve Less Bugs and More Hugs in CSCI 339 (ID: 53477)
2. Distributed Systems for Dummies (ID: 53573)
3. Surviving College (ID: 12365)
4. Cooking for the Impatient Undergraduate (ID: 12498)

Our store employs a two-tier design – a front-end and a back-end. Our front-end server accepts user requests and performs initial processing, while the back-end consists of a database that maintains the inventory and catalog for all items in the store. For each book, the database stores the ID, title, number of items in stock, price, and topic/subject. All books belong to one of two topics: Distributed Systems and College Life. All books are set to a default price of $100.00, but this price can be changed internally. Our server also maintains a log of all purchases for record-keeping purposes.

Our server supports three operations exposed to clients: searching for books by topic (search), looking up a book's information (lookup), and buying a book (buy). Additionally, our server supports operations that are not exposed to clients but are used for internal maintenance of the database: printing the log of purchases (log), restocking books (restock), and updating the price of a book (update).

Clients interact with our server via XML-RPC, and our server interacts with the database via JDBC. Our server handles multiple requests simultaneously using threads. We use a Semaphore variable to control access to the buy and restock operations in a thread-safe way. It ensures that only one thread (or client) can execute the buy method at a time, preventing race conditions where multiple users might try to purchase the same book simultaneously, potentially leading to incorrect stock updates or double sales. It also ensures that internal server-side restocks do not interfere with the buying process for clients.

The restock interval in our implementation is an automated mechanism that updates book stock levels at set intervals. We specified two parameters—restock amount and restock interval—that define the quantity added during each restocking event and how frequently the restocking occurs, in milliseconds. By default, the system restocks each book by 5 units every minute. However, users can customize the restock amounts and interval values through command-line arguments when the server is started. The restocking functionality is managed by a dedicated thread that operates in an infinite loop, sleeping for the specified interval before increasing the quantity by the set amount. This thread allows the inventory to be restocked (i.e., written to the database) without requiring manual intervention.

The frontend server also includes a manual restock function, that can be accessed through the server's terminal, but is not exposed to the client. This allows the user to specify a book's item number and a quantity to restock that particular book with the specified amount. Since our implementation handles concurrent requests, similar to the buy operation, it also uses a binary semaphore to prevent race conditions. This ensures that when multiple clients are interacting with the system concurrently, only one restocking or buying operation can occur at any given time.

Our implementations of the Python and Java clients both provide a command-line interface that interacts with the XML-RPC server to manage client-facing operations such as searching for a book by topic, looking up books by their item numbers or IDs, and buying a book. The interface operates through an interactive loop, offering an intuitive terminal experience where users can input commands like search <topic>, lookup <item_number>, buy <item_number>, and exit. This design allows users to enjoy the bookstore's features while the system efficiently handles input parsing and processing. It also manages errors like type mismatches, invalid commands, and case sensitivity before calling the appropriate functions to communicate with the server.

In the Java client, the connection to the XML-RPC server is established using the Apache XML-RPC library, which requires explicit configuration via the XmlRpcClientConfigImpl object to specify the server's URL. Methods like searchBooksfromServer, lookupBookfromServer, and buyBookfromServer act as intermediaries between the client and server. These methods take user inputs like book IDs or topics, format these inputs in line with the server's API, and process the server's responses. These inputs are packaged into a Vector<Object> to be passed as parameters to the server-side methods (sample.lookup, sample.search, sample.buy, where sample represents the server instance), and the expected output is a String or an array of Strings containing information about the books. This interaction between the client and server uses marshalling and unmarshalling, where client-side data (like search requests or purchase details) are converted into XML for transmission, and once received by the server, data is unmarshalled back into a usable object format for processing. Java's strict type system ensures data consistency during this exchange. On the other hand, the Python client offers a more concise approach by utilizing Python's built-in xmlrpc.client module, which simplifies server communication while retaining the core functionalities and error-handling mechanisms.

In our design, we assume that the operations exposed to clients – search, lookup, and buy – are idempotent. Idempotency guarantees if the request is performed multiple times because of network retries or communication errors, repeated execution of the same operation does not return an incorrect or inconsistent result. Search and Lookup are inherently idempotent as they are read-only and multiple executions of these functions return the unchanged or correct result. Although write operations are usually non-idempotent, our semaphore-based locking mechanism guarantees the system correctly updates the book's stock only once per request.

Here is an example of the output generated by running our program:

```
Enter a command: search <topic>, lookup <item_number>, buy <item_number>, exit
> search Programming Languages

Books under 'Programming Languages':
No books found under this topic.

Enter a command: search <topic>, lookup <item_number>, buy <item_number>, exit
> search Distributed Systems

Books under 'Distributed Systems':
Item Number: 53477, Title: Achieve Less Bugs and More Hugs
Item Number: 53573, Title: Distributed Systems for Dummies

Enter a command: search <topic>, lookup <item_number>, buy <item_number>, exit
> lookup 53477

Book with item number '53477' found:
Title: Achieve Less Bugs and More Hugs, Cost: 100.0, Topic: distributed
systems, In Stock: Yes

Enter a command: search <topic>, lookup <item_number>, buy <item_number>, exit
> buy 53477

Book with item number '53477' bought:
Item 53477 purchased successfully
```

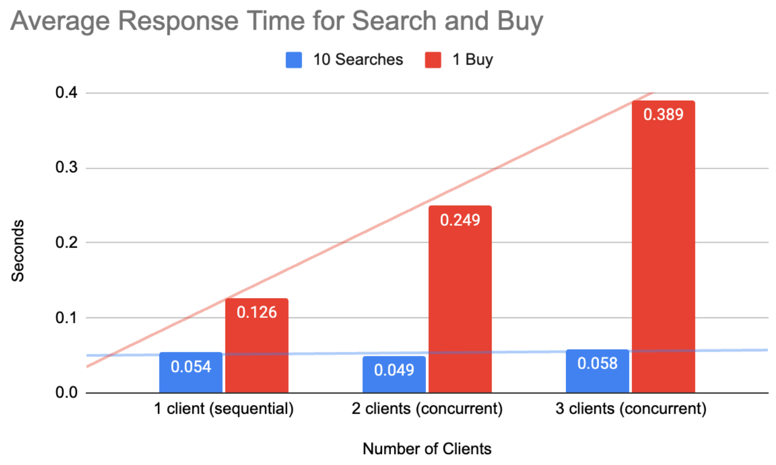
Experimental Setup

To measure the average response time (ART) per client request for our server, we conducted an experiment testing two client-facing commands: search and buy. We developed a simple Python script that made 500 sequential requests to our server and printed out the total response time it took for the server to respond to all requests. First, we tested our search command by running the script multiple times, recording the times, and calculating the average of those times. We then computed the ART per search request by dividing the average total time by 500 requests. We repeated this process for the buy command.

Next, we reran both of these experiments for search and buy; however, instead of running the testing script on its own, we executed two testing scripts simultaneously (in two different terminals) so that the server could handle concurrent requests. We conducted this experiment again with three concurrent testing scripts. After completing all the experiments, we computed the ART per search and buy request for both two-client and three-client scenarios.

Discussion

The results of this experiment are illustrated in the chart below.



This chart shows the ART per 10 client search requests (blue bars) and the ART per client buy request (red bars) as the number of clients making concurrent requests to our server increases. We chose to scale the ART per search request to represent 10 requests instead of a single request because the ART for a single search request was so small that it did not appear in the bar chart at all. From this data, we found that the ART per client search request remained mostly constant as the number of clients increased, while the ART per client buy request increased linearly as the number of clients grew.

The ART per client search request remains mostly constant because search operations are read-only and do not modify the database. When multiple clients perform search requests concurrently, the server can handle these requests efficiently without encountering race conditions or the need for synchronization mechanisms. Since search commands do not lock or modify any data, the server can process them in parallel without delays, resulting in consistent response times regardless of the number of concurrent clients.

On the other hand, the ART per client buy request increases linearly as the number of clients increases because buy operations involve writing to the database. These operations require synchronization to prevent race conditions. In our implementation, a Semaphore is used to ensure that only one thread can perform a buy operation at a time. As the number of clients making buy requests grows, they must wait for the semaphore to become available, which introduces delays. Additionally, updating the database is more time-consuming than simply reading from it, so the cumulative effect of handling multiple buy requests sequentially leads to a linear increase in response times. This reflects the overhead associated with maintaining data consistency and the locking mechanisms required during write operations.

Conclusion

In summary, TinyBookstore.com effectively demonstrates a multi-tier distributed system for managing an online bookstore. Our architecture facilitates key client operations – searching, looking up, and purchasing books – through a responsive front-end server and a back-end database. Our experiments revealed that average response times for search requests remained stable, while those for buy requests increased linearly with the number of concurrent clients, highlighting the impact of synchronization on write operations. These findings underscore the importance of using mechanisms like semaphores to ensure data integrity while handling concurrent requests. Overall, TinyBookstore.com serves as a solid foundation for future enhancements and scalability, showcasing how distributed systems can support e-commerce applications effectively.