An-Najah National University

Faculty of Engineering

Computer Engineering Department

Distributed Operation Systems

Lab 2: Turning the Bazar into an Amazon: Replication, Caching and Consistency

Sewar Aslan, Marwa AbuSaa

12028585, 12028718

Contents

Introduction

The rapid expansion of e-commerce platforms has highlighted the need for efficient, scalable, and consistent systems that can handle high user traffic and large volumes of data. This project aims to enhance the performance of Bazar.com, a simulated online bookstore, by introducing replication, caching, and consistency mechanisms. Initially developed as a simple online store, Bazar.com has seen a surge in demand following the addition of popular new books. This increased demand has led to performance bottlenecks and customer dissatisfaction due to slow response times.

Materials

To complete this lab, we used the following:

- **Docker Desktop**
- **Docker Compose**
- **Node.js (Express)**
- **Postman (Test API)**

Architecture

1. Front-end:

   The front-end tier will accept user requests and perform initial processing. And coordinates the communication of the user interface with the Catalog service or Order service according to the users' requests. So it acts as an entry point for users' requests.

2. Catalog Server:

   It has a database of the books available including details such as title, cost, stock, and topic. It is a microservice responsible for maintaining and providing information on the available books. It implements a RESTful API that allows for operations such as updating book information and performing topic or item number queries for books, and it's implemented as an independent process.

3. Order Server:

   It's responsible for managing customer orders. The OrderServer, being a microservice, talks to the front-end and catalog server in order to ease the purchasing process. This microservice provides a single operation endpoint via RESTful API and is designed to handle the purchase process efficiently.

4. Catalog and Order Replicas:

   Each server is replicated their code and their database files on multiple machines to handle more requests concurrently and improve system reliability.

Implementation

The project was developed using Express in conjunction with Node.js, which made efficient use of its ability to facilitate the creation of lightweight microservices. Express.js is a minimalist, flexible Node.js web application framework that provides an extremely powerful set of features for web and mobile applications. This project utilizes Axios to make HTTP requests between servers. Axios is a promise-based HTTP client for the browser and Node.js, which lets one make asynchronous HTTP requests to REST endpoints. And for the database we used DB.js file which contains a JavaScript object that represents a small collection of books. Each book entry includes an id, title, stock (number of copies available), cost (in some currency), and topic.

Replication, Caching and Consistency:

After made a replica from each server by making container for each one from the same docker-image, the front-end distributes incoming requests across replicas using a load-balancing algorithm (round-robin), ensuring even load distribution and reducing bottlenecks. By this we achieved the Load Balancing.

Consistency Across Replicas: The replicas use an internal protocol to ensure data consistency. For example, updates are mirrored across all replicas to keep data synchronized, which is crucial for write operations to maintain database integrity, so if one replica updates the stock of an item, it notifies the other replica to make the same update.

Caching: the cache is a key-value store, where keys represent specific data requests (e.g., book ID) and values store the results. We add an in-memory cache that caches the results of recent requests to the order and catalog servers.

For the Cache Policy, we implemented an LRU (Least Recently Used) cache replacement policy to limit the cache size and maintain efficiency. A server-push invalidation mechanism was

implemented to remove cache entries when data changes occur in the catalog/catalog replica database. This ensures that outdated data is not served to clients.

Setup and Installation

1. Clone the repository to your local machine: https://github.com/SewarAslan/Bazarcom/tree/Part2.
2. Navigate to the project directory: cd Bazarcom
3. Build the Docker images and start the containers: docker-compose up –build
4. The system should now be running on Docker containers.

Testing the APIs using API test tool

You can test APIs using tools like Postman. Here are the endpoints available:

- Frontend APIs:

URL	METHOD	DESCRIPTION
/Bazarcom/info/:id	GET	Get information about a book by its <ID>
/Bazarcom/search/:topic	GET	Search for books by their <topic>,return all matching books IDs and Titles
/Bazarcom/ purchase/:id	POST	Buy a book by its <ID>

- Catalog APIs:

URL	METHOD	DESCRIPTION
/catalogServer/query	GET	Allows users to search a book based on either topic or id
/CatalogServer/updateStock/:itemNumber	PUT	Update the stock of a specific book based on its (ID) when it has bought.

- Order APIs:

URL	METHOD	DESCRIPTION
/OrderServer/purchase/:itemNumber	POST	Handles the purchase of a specific book by its (ID)

Examples:

- From Frontend:

Search Books by Topic:

Endpoint: http://localhost:2000/Bazarcom/Search/:topic

Example: http://localhost:2000/Bazarcom/Search/ distributed systems

Get Book Information:

Endpoint: http://localhost:2000/Bazarcom/info/:id

Example: http://localhost:2000/Bazarcom/info/102

Purchase Book:

Endpoint: http://localhost:2000/Bazarcom/purchase/:id

Example: http://localhost:2000/Bazarcom/purchase/101

How system handles the requests

1. http://localhost:2000/Bazarcom/Search/ distributed systems **or** http://localhost:2000/Bazarcom/info/102

when this request occurs, it will keep the catalog replica that has been chosen by round-robin algorithm.

```
// function to get catalog replica URL
function getCatalogReplicaURL() {
    const replicas = ['http://catalog:2001', 'http://catalogReplica:2001'];
    const replica = replicas[catalogReplicaIndex];
    catalogReplicaIndex = (catalogReplicaIndex + 1) % replicas.length;

    //test which replica catch the request
    console.log(`Using ${replica}`);

    return replica;
}
```

Then, it checks if the request results exist in cache, if it's, it will return the result from cache. If not, it will forward the request to the chosen replica and added the result to the cache.

These are the cache functions to do that:

```
// checks the cache first if data in it
function getFromCache(key) {
    if (cache.has(key)) {
        // Move new item to the end to mark it as recently used/ we used LRU cache replacement policy
        const value = cache.get(key);
        cache.delete(key);
        cache.set(key, value);
        return value;
    }
    return null;
}

// set new data in cache
function setCache(key, value) {
    if (cache.size >= CACHE_SIZE) {
        // Remove the first (least recently used) item from the cache according
        const firstKey = cache.keys().next().value;
        cache.delete(firstKey);
    }
    cache.set(key, value);
}
```

2. http://localhost:2000/Bazarcom/purchase/101

   when this request occurs, it will keep the order replica that has been chosen by round-robin algorithm.

```
// function to get order replica URL
function getOrderReplicaURL() {
    const replicas = ['http://order:2002', 'http://orderReplica:2002'];
    const replica = replicas[orderReplicaIndex];
    orderReplicaIndex = (orderReplicaIndex + 1) % replicas.length;

    //test which replica catch the request
    console.log(`Using ${replica}`);

    return replica;
}
```

Then, the chosen order replica will forward the request to the corresponded Catalog Server to update the stock of item, so if updated done successfully, the catalog server will notify the in-memory cache with an invalidate request if the item in cache, the invalidate request causes the data for that item to be removed from the cache. If not, no invalidation is needed.

```
// Endpoint for push technique where backend replicas send invalidate requests to the in-memory cache (request cache invalidation)
app.post('/invalidateCache', (req, res) => {
    const { id } = req.body;
    const cacheKey = `id-${id}`;


    if (cache.has(cacheKey)) {
        //delete item from cache
        cache.delete(cacheKey);
        console.log(`Done deleted item with id: `+ id);

        console.log(`Cache invalidated for item with ID: ${id}`);
        res.json({ message: `Cache invalidated for item with ID: ${id}` });
    } else {
        console.log(`Item with ID ${id} not found in cache, no invalidation needed`);
        res.json({ message: `Item with ID ${id} not in cache` });
    }
});
```

Also, when updated done successfully, this catalog replica will notify the other replica that the stock is changed for this item. So it will update it in its database.

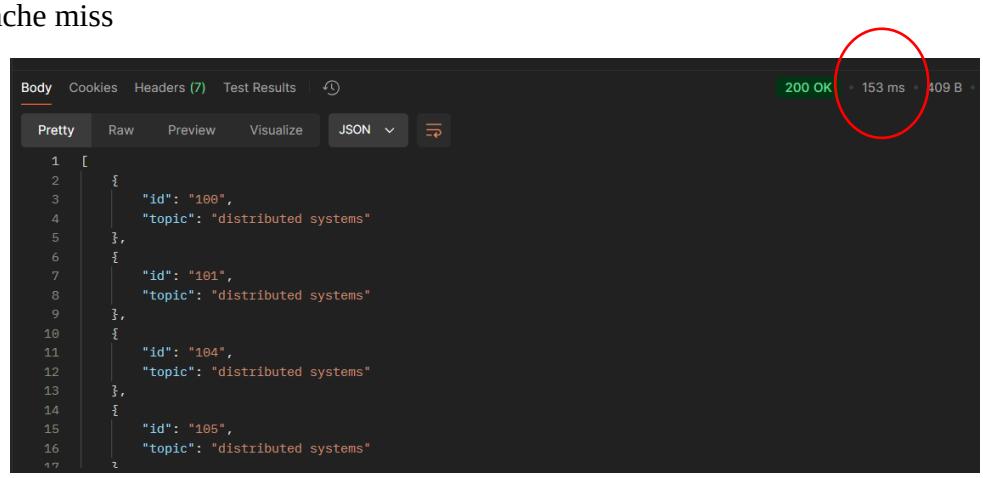
```
app.put('/CatalogServer/updateReplicaStock/:itemNumber', (req, res) => {
    const itemNumber = req.params.itemNumber;
    const { stock } = req.body;
    const item = data.find(book => book.id === itemNumber);

    if (item) {
        item.stock = stock;
        console.log(`origin stock updated for item ${itemNumber}. New stock: ${item.stock}`);
        res.json({ message: `Stock updated for item ${itemNumber}` });
    } else {
        res.status(404).json({ message: 'Item not found on replica' });
    }
});
```

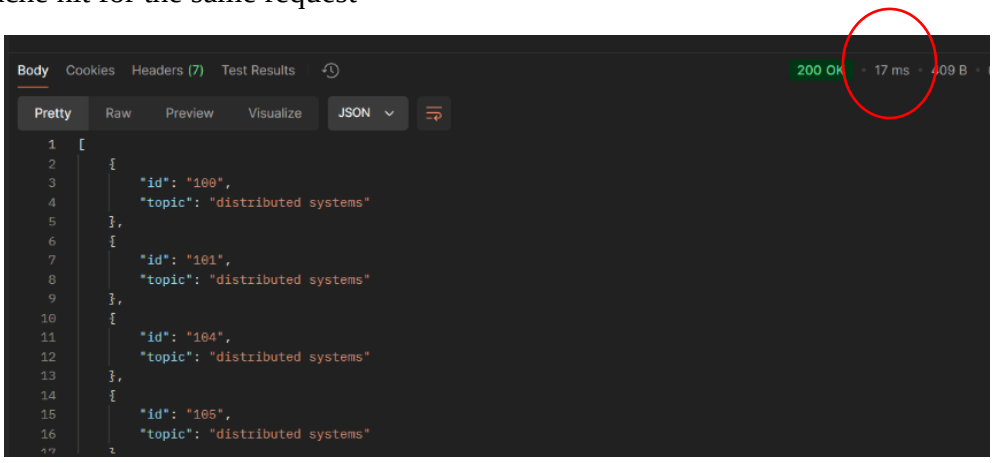
Output

Search Books by Topic:

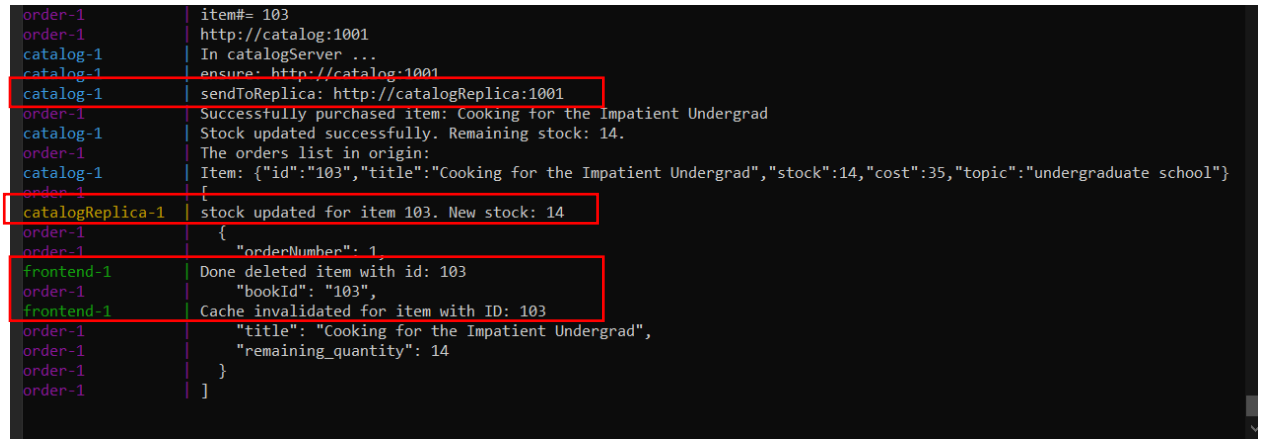
a) Cache miss



b) Cache hit for the same request



Purchase Book:



Purchase item with id 103 that is in cache, catalogServer did this and notify the catalogReplica, also send invalidation to cache to delete the item.

Note: The rest of outputs are existed on our github repository: https://github.com/SewarAslan/Bazarcom/tree/Part2.

Experimental Evaluation and Measurements

1) a. Average response time (search/info).

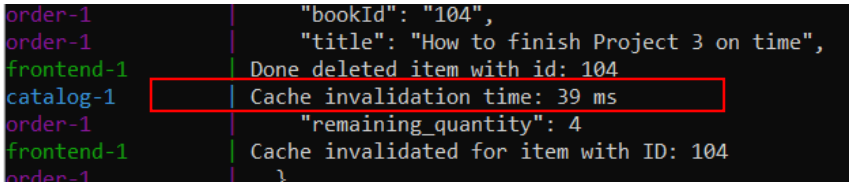
Experiment Run	Without Cache	With Cache	improvement
1	153	17	153/17=9
2	32	12	32/12=2.67

b.How much does caching helps?

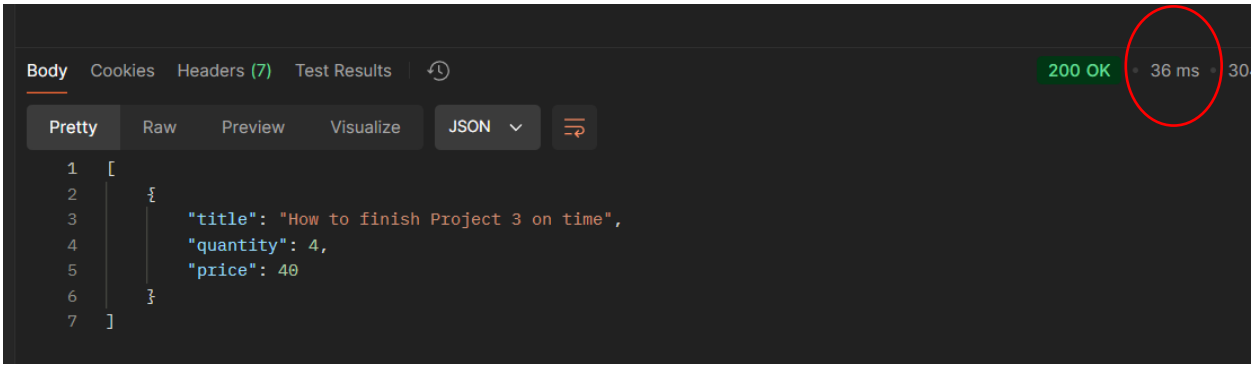
It helps us to reduce so much time, so we'll have better performance, and it improves the system availability and reliability.

2) The overhead of cache consistency operations

```
// Notify the frontend about cache invalidation
const startTime = Date.now();
await axios.post('http://frontend:1000/invalidateCache', { id: itemNumber });
const endTime = Date.now();
console.log(`Cache invalidation time: ${endTime - startTime} ms`);
```



3) Latency of Cache Miss**:** After invalidation



Conclusion

To improve performance, we implemented caching, replication, and consistency mechanisms. These features allowed the system to handle higher user demand while keeping data accurate and response times fast. Testing through Postman confirmed that the APIs worked smoothly for searching books, updating stock, and processing orders.

Overall, this project taught us how modern tools and approaches—like distributed systems and microservices—can solve practical challenges. It highlighted the importance of efficient design in creating systems that are not only functional but also adaptable to future growth.