An-Najah National University

Faculty Of Engineering

Computer Engineering Department

Distributed Operation Systems (10636456)

Lab 2: Bazar.com: A Multi-tier Online Book Store

Students:

Wala’ Essam Ashqar

12027854

Doaa Yasin Jararaa

12029152

-May 2025-

- # Introduction

1. Bazar.com, the online bookstore system from Lab 1, has been editing in Lab 2 to improve request processing latency and develop it.
2. This was achieved by implementing key distributed systems concepts: replication of backend services, caching at the front end, load balancing to distribute requests, and mechanisms for cache consistency and data synchronization between replicas.
3. The system continues to use a microservices architecture with Node.js and Docker containers, featuring frontend, catalog (now replicated), and order (now replicated) services.

- # System Design

1. Overall Architecture:

- The system maintains its multi-tier, microservices-based design.
- **Frontend Service: A single** instance acts as the entry point and now includes caching and load balancing.
- **Catalog Service:** Replicated into two instances (e.g., catalog1, catalog2), each with its own SQLite database.
- **Order Service:** Replicated into two instances (e.g., order1, order2), each with its own SQLite database.
- Services communicate via HTTP REST APIs over a Docker network.

2. A. Frontend Service:

- **Caching:**

1. Implements an in-memory cache for info requests to reduce latency.
2. Checks cache before forwarding requests to the catalog service.
3. Cache invalidation endpoint (POST /cache/invalidate/:item_id) allows backend services to clear cache entries.

- **Load Balancing:**

Implements a round robin algorithm to distribute requests to

1. catalog1 and catalog2 for search and info operations.
2. order1 and order2 for purchase operations.

- **Core Operations:**

1. **Search Item (GET /search/:topic):**
   Forwards request to one of the catalog replicas (round robin).
2. **Get Item Info (GET /info/:item_id):**
   Checks local cache first. On a cache miss, forward the request to one of the catalog replicas (round robin) and cache the result.
3. **Purchase Item (POST /purchase/:item_id):**
   Forwards request to one of the order replicas (round robin).

3. B. Catalog Service (Replicated: catalog1, catalog2):

- Handles product data, search, info, and stock updates. Each replica has its own database.
- Data Synchronization & Cache Invalidation on Update (PUT /update/:item_id):
- When a replica (e.g., catalog1) receives an update request (either directly or from an order service):

1. It updates its local database and sends the same update request (with an x-sync-request header) to the other catalog replica (e.g., catalog2) to synchronize data.
2. Sends a cache invalidation request (POST /cache/invalidate/:item_id) to the frontend service (if it wasn't a sync request itself).

- A replica receiving a sync request (with x-sync-request header) only updates its local database.

4. C. Order Service (Replicated: order1, order2):

- Handles purchase requests. Each replica has its own database.
- Purchase Process (POST /purchase/:item_id):

1. Receives purchase request from the frontend.
2. Queries one of the catalog replicas (e.g., using round robin) for item stock (GET /info/:item_id from catalog).
3. If the item is in stock:

→ Sends stock update requests (PUT /update/:item_id) to **both** catalog replicas (catalog1 and catalog2). (The catalog replicas then handle their internal sync and cache invalidation.).

→ Records the order in its local database.

4. Returns purchase status to the frontend.

- **Discussion**

- Replication & Load Balancing: Using two replicas for backend services with round robin load balancing at the frontend provides improved availability and distributes load simply.
- Caching: In-memory cache at the frontend significantly improves read latency for info requests.
- Consistency: Cache consistency is maintained via server-push invalidation from the catalog service. Catalog stock is attempted by having the order service update both catalog replicas, and catalog replicas sync further direct updates.
- Order data is not synchronized between order service replicas. The data synchronization protocol between catalog replicas is basic.

- **Experimental Evaluation and Measurements**

Performance measurements were conducted using Postman, which I will provide in the output pics for P2 in the project folder on GitHub, and also a custom Python script to evaluate the system.

This is the output of the script:

Operation	No. of Requests	Avg. Response Time (ms)	Notes
`GET /info` (Cache Enabled - Mostly Hits)	30	20.01	Represents typical read performance with cache
`POST /purchase` (for item 7, sufficient stock)	5	144.32	All purchase operations were successful

Experiment Step	Response Time (ms)	Observed Stock	Notes
1. Initial `GET /info` (Cache Miss, primes cache)	14.79	15	Item fetched from catalog and cached

2. Second GET /info (Cache Hit)	5.27	15	Item served quickly from cache
3. POST /purchase (Triggers cache invalidation)	115.01	N/A	Purchase involves stock update & invalidation
4. GET /info (Post-purchase, new Cache Miss)	50.52	14	Cache was invalidated, fresh data fetched

- So caching reduced GET /info response times by approximately **64.4%** (from 14.79 ms on a miss to 5.27 ms on a hit), significantly speeding up read operations.
- POST /purchase operations averaged **144.32 ms**, which includes stock updates across catalog replicas and cache invalidation signaling.
- Successful purchases correctly invalidated the frontend cache. Subsequent GET /info requests for the same item were cache misses (e.g., 50.52 ms), ensuring data freshness.

• Running the Project with Docker Compose

➢ The system is managed using docker-compose.yml.

➢ **Steps to Run:**

1. Ensure the following directories are created manually if they don't exist: src/catalog/db_c1, src/catalog/db_c2, src/order/db_o1, src/order/db_o2
2. Navigate to the project's root directory (where docker-compose.yml is located).

→ Run: docker-compose up --build

→ This command builds the Docker images for all services and starts the containers.

→ The frontend service will be accessible at http://localhost:3000.

➢ **To Stop the System:**

Run docker-compose down (this stops and removes containers and networks).