

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

Distributed Operating Systems

Dr Samer Arandi

By Zain Abubaker && Abd-Alkareem Al Masri

Overview

A containerized microservices-based bookstore.

Enhanced from Lab 1 with caching, replication, load balancing, and consistency management.

System Architecture

5 services

- frontend: manages all incoming requests, load balancer, and in-memory cache.
- Catalogs 1 and 2: replicate updates and serve book data.
- Orders 1 and 2: manage purchases and duplicate logs and updates.
- REST APIs are used by all services to communicate.
- Orchestration is done with Docker Compose.

```
attribute version is obsolete, it will be ignored, please remove it to avoid potential
NAME                IMAGE                COMMAND              SERVICE    CREATED
bazar_com-catalog1-1 bazar_com-catalog1  "python app.py"      catalog1    59 seconds ago
bazar_com-catalog2-1 bazar_com-catalog2  "python app.py"      catalog2    59 seconds ago
bazar_com-frontend-1 bazar_com-frontend  "python app.py"      frontend    59 seconds ago
bazar_com-order1-1   bazar_com-order1    "python app.py"      order1      59 seconds ago
bazar_com-order2-1   bazar_com-order2    "python app.py"      order2      59 seconds ago
PS C:\Users\Msys\OneDrive\Desktop\bazar\bazar\bazar_com\
```

Caching Mechanism

- Implemented in-memory cache in frontend.
- Cache stores `/info/item_id` responses.
- TTL is 60 seconds.
- Cache is invalidated when a purchase or update occurs.
- Optional extension: LRU policy can be added.

```
1  {  
2    "price": 30.0,  
3    "quantity": 9,  
4    "title": "How to get a good grade in DOS in 40 minutes a day"  
5  }
```

Replication & Load Balancing

- Two replicas each for catalog and order.
- Frontend uses round-robin strategy for both services.
- Replicas sync using REST calls:
 - **catalog1** sends updates to catalog2 and vice versa.
 - **order1** syncs purchase to order2 and vice versa.
- Backend updates invalidate the cache via `/invalidate/<item_id>` on frontend.

When running POST `http://localhost:5000/purchase/1`

```
1  {  
2    "message": "bought book How to get a good grade in DOS in 40 minutes a day"  
3  }
```

And then we run GET `http://localhost:5000/info/1`, we will notice a decrease in quantity

```
1  ✓ {  
2    "price": 30.0,  
3    "quantity": 8,  
4    "title": "How to get a good grade in DOS in 40 minutes a day"  
5  }
```

Consistency Strategy

- Strong consistency via server-push invalidation.
- Writes are synchronized across replicas.
- Only original replica performs the stock update; replica just acknowledges.

Performance Evaluation

| Operation | Time (ms) |
|-------------------------------|-----------|
| /info (1st call - cache miss) | 35.48 |
| /info (2nd call - cache hit) | 6.87 |
| /purchase | 53.10 |
| /info after invalidation | 12.58 |

```
PS C:\Users\Msys\OneDrive\Desktop\bazar\bazar\bazar_com> python performance_test.py
=== Measuring /info with Caching ===
1st /info (cache miss): 35.48 ms
2nd /info (cache hit): 6.87 ms

=== Measuring /purchase (forces invalidation) ===
/purchase: 53.1 ms
/info after purchase (cache miss): 12.58 ms
```

Observations:

- Cache hits reduce latency by -80%.
- Invalidation ensures correctness with acceptable latency overhead.

