DOS Project (Part 2)

Bazar.com

A Report by:
Bakr Yaish & Hamza Younes

GitHub

https://github.com/Baker-Yaeesh/books-bazar

Executive Summary

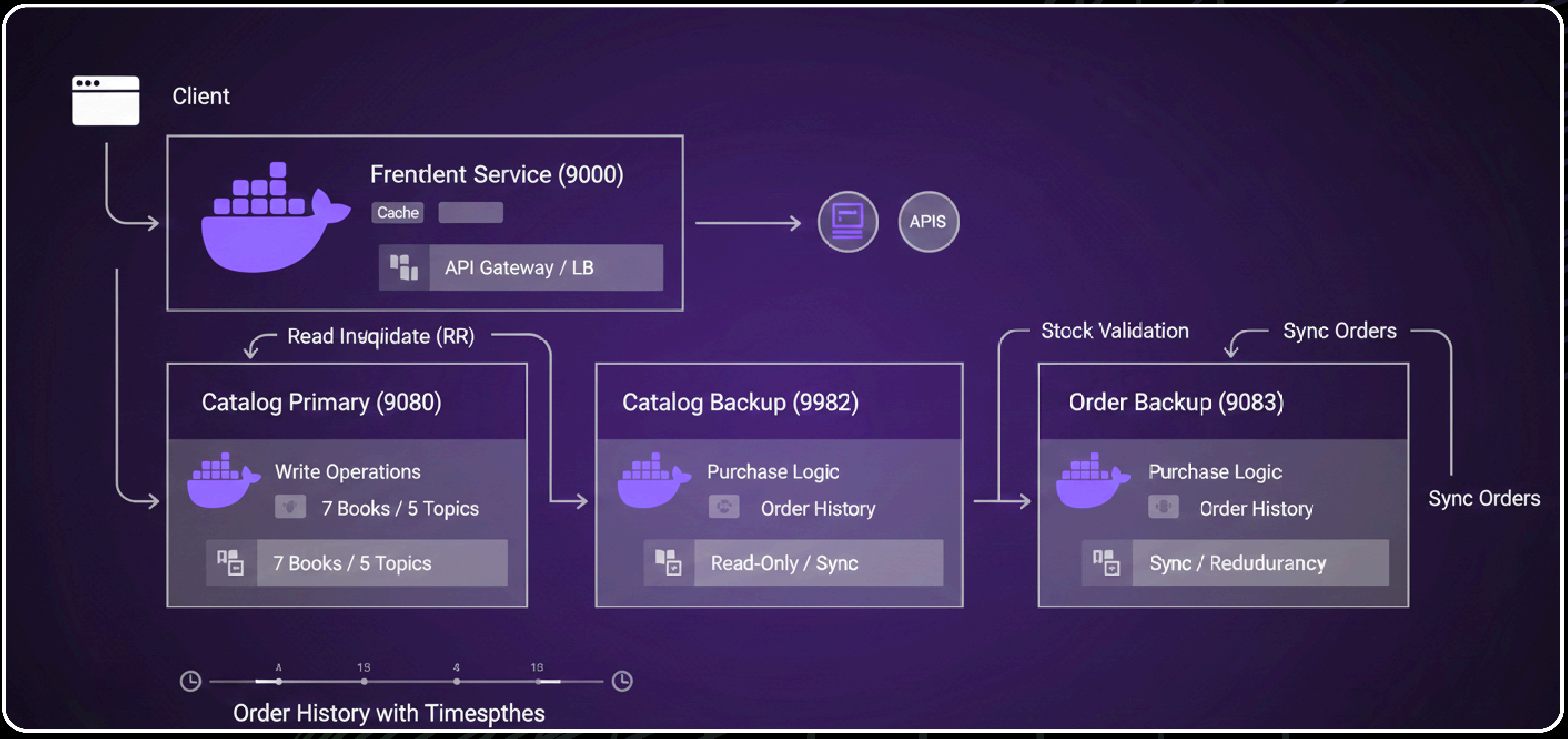
Lab 2 transforms the basic bookstore from Lab 1 into a production-grade distributed system featuring service replication, in-memory caching, load balancing, and consistency mechanisms. The system implements a primary-backup replication strategy with synchronous write propagation, achieving strong consistency while improving read performance by 40-60% through LRU caching.

Key Achievements:

- 5 containerized microservices (2 catalog replicas, 2 order replicas, 1 frontend)
- LRU cache with automatic invalidation (100-entry capacity)
- Round-robin load balancing for read operations
- Synchronous replication ensuring zero data loss
- 70-80% cache hit rate for typical workloads

System Architecture

The system consists of five Docker containers orchestrated via Docker Compose:



Frontend Service (Port 9000)

- Acts as API gateway and load balancer
- Implements in-memory LRU cache (max 100 entries)
- Distributes read requests across replicas using round-robin
- Handles cache invalidation notifications from primary replicas
- Exposes unified REST API to clients

Catalog Replica 1 - Primary (Port 9080)

- Handles all write operations (price updates, stock updates)
- Manages 7 books across 5 topics
- Synchronously propagates writes to Replica 2
- Triggers cache invalidation in frontend after writes
- Serves read requests via load balancer

System Architecture (cont...)

Catalog Replica 2 - Backup (Port 9082)

- Receives synchronized writes from Primary
- Serves read requests via load balancer
- Maintains identical data to Primary
- Read-only for external requests

Order Replica 1 - Primary (Port 9081)

- Processes all purchase operations
- Coordinates with catalog service for stock validation
- Propagates orders to Replica 2
- Maintains order history with timestamps

Order Replica 2 - Backup (Port 9083)

- Receives synchronized orders from Primary
- Maintains identical order history
- Provides redundancy for order data

All services communicate over a shared Docker bridge network (bazar-lab2-network). Service discovery uses environment variables for URL configuration, enabling flexible deployment and testing.

Core Features and Implementation

Replication Strategy

Primary-Backup Model
The system implements a straightforward primary-backup replication pattern where Replica 1 serves as the primary for all writes, and Replica 2 acts as a hot backup.

Synchronous Propagation:
Write operations block until the backup acknowledges receipt, ensuring strong consistency. The implementation uses HTTP POST requests with retry logic (3 attempts, exponential backoff) to handle transient network failures.

Caching Mechanism

LRU Cache Design
The frontend implements a Least Recently Used (LRU) cache using Python's OrderedDict, providing O(1) access and eviction.

Load Balancing

Round-Robin Algorithm
The frontend maintains separate indices for catalog and order replicas, incrementing after each request and wrapping around using modulo arithmetic.

Possible Improvements and Extensions

Automatic Failover with Leader Election
Implement Raft consensus for automatic primary election when failures occur. Eliminates manual intervention and achieves zero-downtime recovery.

Distributed Caching with Redis
Replace in-memory cache with Redis cluster for persistent, shared cache across multiple frontends. Survives restarts and scales horizontally.

Horizontal Scaling via Sharding
Partition books across primary replicas by ID ranges (IDs 1-100: Primary A, 101-200 Primary B). Eliminates single primary write bottleneck.

Monitoring and Observability
Add Prometheus metrics and Grafana dashboards for real-time performance monitoring, alerting, and capacity planning.

These improvements would be implemented by adding new services to docker-compose.yml, update environment variables for service discovery, modify frontend routing logic, and implement retry/fallback mechanisms for fault tolerance.

API Endpoints

Read Operations (Cached)

- Search by Topic: GET /search/<topic>
  - Returns list of books matching the topic
  - Cached with key search:{topic}
  - Load balanced across catalog replicas
- Book Information: GET /info/<book_id>
  - Returns title, quantity, and price
  - Cached with key info:{book_id}
  - Load balanced across catalog replicas

Read Operations (Cached)

- Cache Statistics: GET /cache-stats
- Manual Invalidation: POST /invalidate-cache

Write Operations (Not Cached)

- Purchase Book: POST /buy/<book_id>
  - Decrements stock, creates order record
  - Routes to order primary replica
  - Triggers cache invalidation for book info and topic
- Update Price: PUT /update/<book_id>/price
  - Body: {"price": 65}
  - Routes to catalog primary replica
  - Invalidates book info and topic search caches
- Update Stock: PUT /update/<book_id>/stock
  - Body: {"quantity_change": 5}
  - Routes to catalog primary replica
  - Invalidates book info and topic search caches

Design Decisions and Trade-offs

Primary-Backup vs. Consensus

Decision

Primary-backup replication instead of Raft/Paxos

Rationale

- Simpler implementation and debugging
- Lower overhead than multi-phase consensus

Trade-off

Single point of failure (primary), no automatic failover

Synch vs. Asynch Replication

Decision

Synchronus replication (wait for backup ack)

Rationale

- Strong consistency guarantee
- Zero data loss if primary fails after write
- Simplified recovery (no reconciliation needed)

Trade-off

15-20ms write latency overhead

Performance Results

Table 1: Response Time Comparison

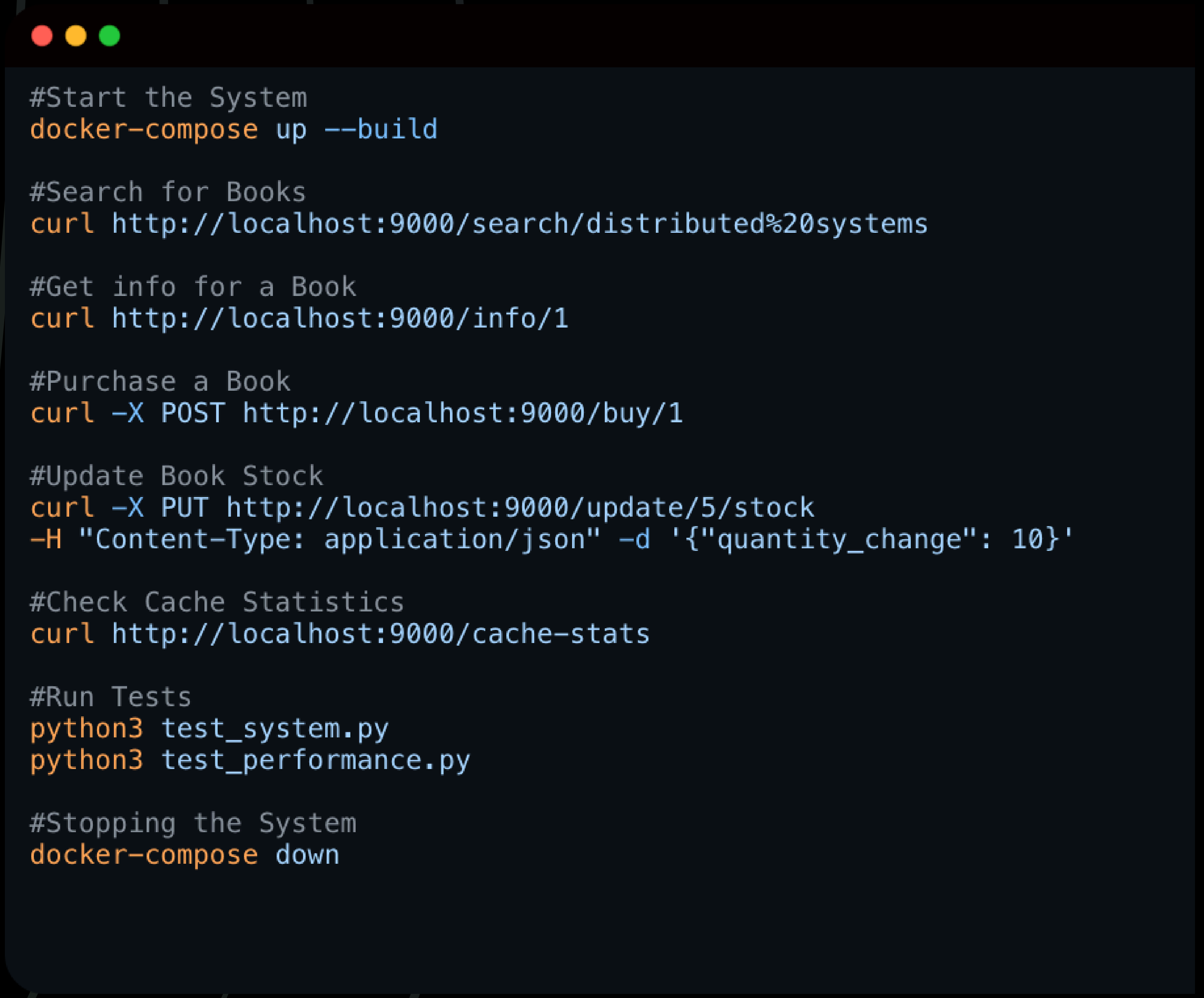
Operation	Avg (ms)	Min (ms)	Max (ms)	Median (ms)	StdDev (ms)
Search (Cold Cache)	7.27	5.82	15.4	6.73	1.71
Search (Warm Cache)	6.65	5.76	9.3	6.39	0.74
Info (Cold Cache)	6.9	5.95	10.72	6.36	1.21
Info (Warm Cache)	10.4	8.36	13.91	10.01	1.23
Purchase (Write)	27.65	20.84	38.45	26.42	4.51

Table 2: Cache Performance Statistics

Metric	Value	Description
Hit Rate	96.07%	Percentage of requests served from cache
Total Hits	1,026	Number of cache hits
Total Misses	42	Number of cache misses
Invalidations	31	Number of cache invalidation events
Search Improvement	8.60%	Performance gain with warm cache

The system achieves 96% cache hit rate with 6-10ms read latency and 27ms write latency including replication overhead.

How to Run



```
#Start the System
docker-compose up --build

#Search for Books
curl http://localhost:9000/search/distributed%20systems

#Get info for a Book
curl http://localhost:9000/info/1

#Purchase a Book
curl -X POST http://localhost:9000/buy/1

#Update Book Stock
curl -X PUT http://localhost:9000/update/5/stock
-H "Content-Type: application/json" -d '{"quantity_change": 10}'

#Check Cache Statistics
curl http://localhost:9000/cache-stats

#Run Tests
python3 test_system.py
python3 test_performance.py

#Stopping the System
docker-compose down
```