Dos project Part-2
Replication, Caching, and Load Balancing

Introduction

This report presents the implementation and evaluation of a distributed bookstore system developed as part of the Distributed Operating Systems laboratory. The system follows a microservices-based architecture, where different services are deployed as independent components and communicate through well-defined APIs.

The main objective of this experiment is to study the effects of replication, caching, and load balancing on system performance and scalability. To achieve this, the system is composed of multiple replicas of catalog and order services, a client service that acts as an entry point for user requests, and a Redis-based caching layer to reduce response time and database load.

Docker and Docker Compose are used to containerize and orchestrate the services, allowing multiple instances to run concurrently in a controlled environment. Load balancing is implemented using a round-robin strategy, while cache consistency is maintained through explicit cache invalidation upon write operations such as purchases or updates.

Throughout this report, we measure response times for read and write operations, analyze the performance improvement introduced by caching, and evaluate the overhead caused by cache invalidation and replica synchronization. The results demonstrate how distributed system techniques can significantly improve system efficiency while introducing new design challenges related to consistency and coordination.

System Architecture

The system follows a microservices architecture that was introduced in Part 1. Each service runs inside an independent Docker container and communicates via HTTP.

In Part 2, the architecture was extended by deploying multiple replicas for the catalog and order services to support load balancing and fault tolerance. Additionally, a Redis cache was integrated as a centralized caching layer to improve read performance. Docker Compose is used to manage and orchestrate all services.

Technologies Used

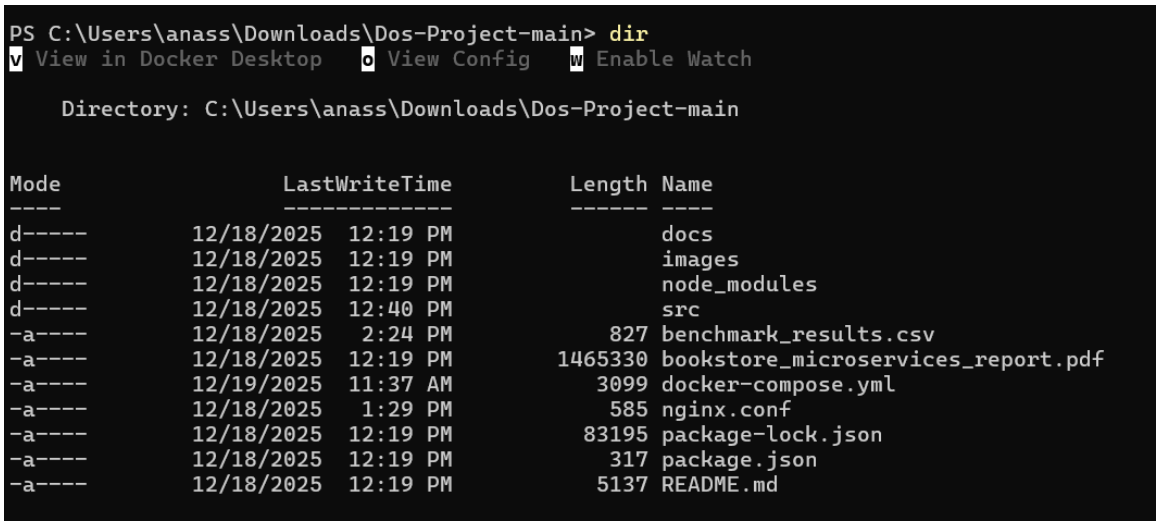
Base Technologies (from Part 1):

- Node.js
- Express.js

- SQLite
- Docker
- Docker Compose

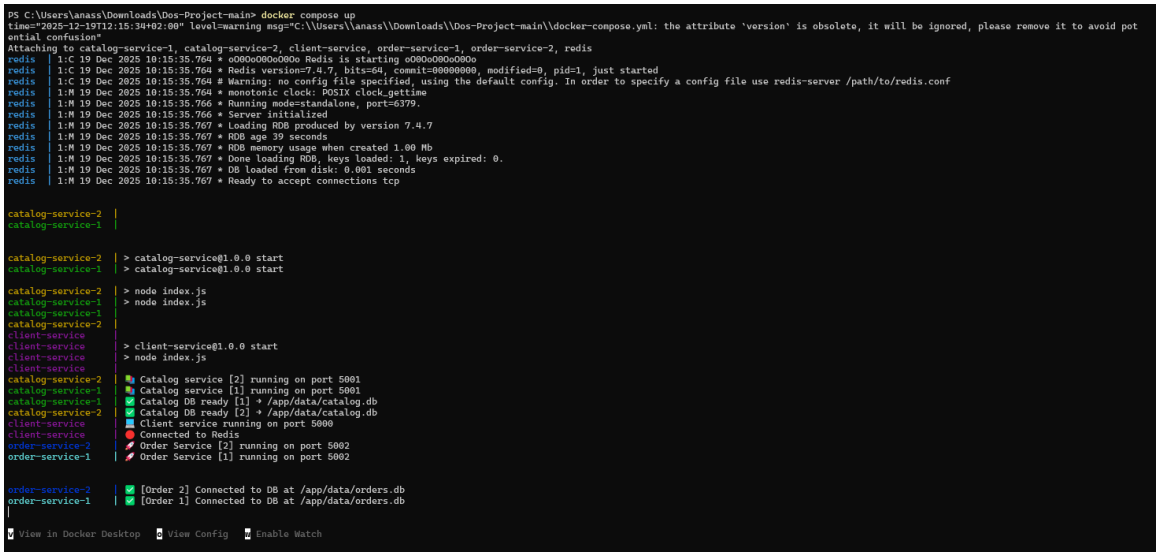
New Technologies Added in Part 2:

- Redis (centralized caching)
- cURL / PowerShell Measure-Command (performance evaluation)



```
PS C:\Users\anass\Downloads\Dos-Project-main> dir
V View in Docker Desktop   o View Config   W Enable Watch

    Directory: C:\Users\anass\Downloads\Dos-Project-main


Mode                 LastWriteTime         Length Name
----                 -------------         ------ ----
d-----         12/18/2025  12:19 PM                docs
d-----         12/18/2025  12:19 PM                images
d-----         12/18/2025  12:19 PM                node_modules
d-----         12/18/2025  12:40 PM                src
-a----         12/18/2025   2:24 PM            827 benchmark_results.csv
-a----         12/18/2025  12:19 PM        1465330 bookstore_microservices_report.pdf
-a----         12/19/2025  11:37 AM           3099 docker-compose.yml
-a----         12/18/2025   1:29 PM            585 nginx.conf
-a----         12/18/2025  12:19 PM          83195 package-lock.json
-a----         12/18/2025  12:19 PM            317 package.json
-a----         12/18/2025  12:19 PM           5137 README.md
```

Project directory containing docker-compose.yml used to deploy all services.



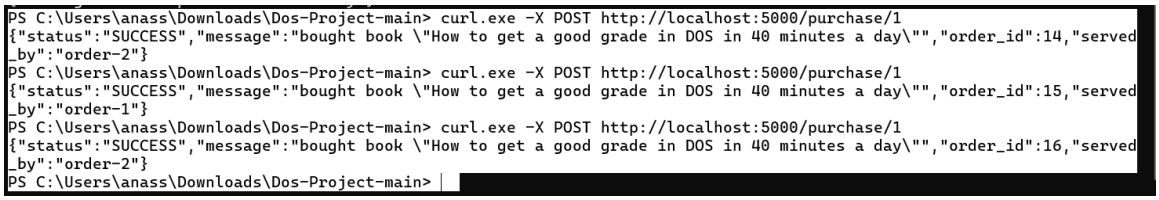
```
PS C:\Users\anass\Downloads\Dos-Project-main> docker compose up
time="2025-12-19T12:15:34+02:00" level=warning msg="C:\\Users\\anass\\Downloads\\Dos-Project-main\\docker-compose.yml: the attribute `version` is obsolete, it will be ignored, please remove it to avoid potential confusion"
Attaching to catalog-service-1, catalog-service-2, client-service, order-service-1, order-service-2, redis
redis  | 1:C 19 Dec 2025 10:15:35.764 * oO0OoO0OoO0Oo Redis is starting oO0OoO0OoO0Oo
redis  | 1:C 19 Dec 2025 10:15:35.764 * Redis version=7.4.7, bits=64, commit=00000000, modified=0, pid=1, just started
redis  | 1:C 19 Dec 2025 10:15:35.764 # Warning: no config file specified, using the default config. In order to specify a config file use redis-server /path/to/redis.conf
redis  | 1:M 19 Dec 2025 10:15:35.764 * monotonic clock: POSIX clock_gettime
redis  | 1:M 19 Dec 2025 10:15:35.766 * Running mode=standalone, port=6379.
redis  | 1:M 19 Dec 2025 10:15:35.766 * Server initialized
redis  | 1:M 19 Dec 2025 10:15:35.767 * Loading RDB produced by version 7.4.7
redis  | 1:M 19 Dec 2025 10:15:35.767 * RDB age 39 seconds
redis  | 1:M 19 Dec 2025 10:15:35.767 * RDB memory usage when created 1.00 Mb
redis  | 1:M 19 Dec 2025 10:15:35.767 * Done loading RDB, keys loaded: 1, keys expired: 0.
redis  | 1:M 19 Dec 2025 10:15:35.767 * DB loaded from disk: 0.001 seconds
redis  | 1:M 19 Dec 2025 10:15:35.767 * Ready to accept connections tcp

catalog-service-2  |
catalog-service-1  |

catalog-service-2  | > catalog-service@1.0.0 start
catalog-service-1  | > catalog-service@1.0.0 start

catalog-service-2  | > node index.js
catalog-service-1  | > node index.js
catalog-service-1  |
catalog-service-2  |
client-service     |
client-service     | > client-service@1.0.0 start
client-service     | > node index.js
client-service     |
catalog-service-2  | 📚 Catalog service [2] running on port 5001
catalog-service-1  | 📚 Catalog service [1] running on port 5001
catalog-service-1  | ✅ Catalog DB ready [1] → /app/data/catalog.db
catalog-service-2  | ✅ Catalog DB ready [2] → /app/data/catalog.db
client-service     | 🚀 Client service running on port 5000
client-service     | 🔴 Connected to Redis
order-service-2    | 🚀 Order Service [2] running on port 5002
order-service-1    | 🚀 Order Service [1] running on port 5002

order-service-2    | ✅ [Order 2] Connected to DB at /app/data/orders.db
order-service-1    | ✅ [Order 1] Connected to DB at /app/data/orders.db
|
V View in Docker Desktop   o View Config   W Enable Watch
```

All microservices were successfully deployed using Docker Compose. The catalog service replicas, order service replicas, Redis cache, and client service are all running correctly without errors.

Load Balancing and Replication

To improve scalability and fault tolerance, multiple replicas were deployed for both the catalog service and the order service. Two instances of each service run concurrently inside separate Docker containers. The client service acts as a simple load balancer by distributing incoming requests among the available replicas using a round-robin strategy. Each request is forwarded to a different replica in sequence, ensuring that the workload is evenly distributed across all instances. This approach allows the system to handle a higher number of concurrent requests and prevents a single service instance from becoming a bottleneck. Additionally, replication improves system reliability, as requests can still be served even if one replica becomes unavailable.

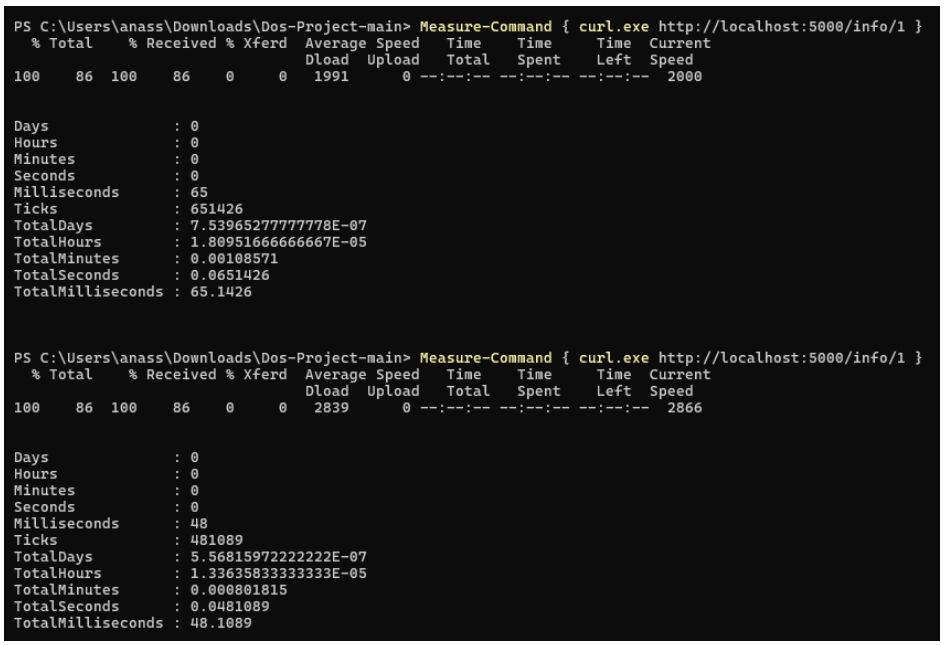
Load Balancing Between Order Service Replicas:



The terminal output shows multiple purchase requests sent to the client service. Each request is handled by a different order service replica, as indicated by the `served_by` field in the response. This demonstrates that load balancing is successfully distributing requests between order service replicas using a round-robin strategy.

Performance Evaluation

1. Measuring Read Performance (With Cache)



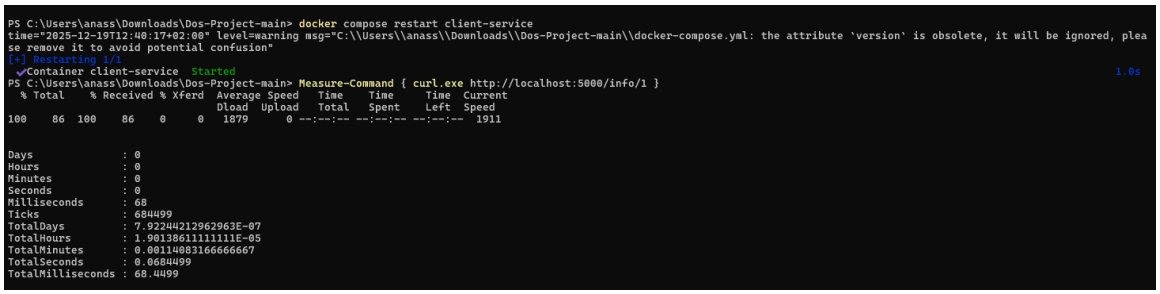
To ensure accurate performance evaluation, the response time of the `/info/:id` endpoint was measured multiple times while the cache was enabled. Although the screenshot shows only two executions, the experiment was conducted **five times**, and the average response time was calculated based on all collected measurements

The average response time was calculated using the following formula:

$$\text{Average Response Time} = (T1 + T2 + T3 + T4 + T5) / 5$$

Measurement	Response Time (ms)
1	65
2	48
3	51.2
4	50.3
5	63.9
Average	≈ 55.7 **ms**

2. Measuring Read Performance (Without Cache)



Measurement	Response Time (ms)
1	68
2	71.9
3	61.2
4	65.9
5	62.2
Average	≈ 65.8 ms

- Overhead of Cache Consistency:

Cache consistency introduces additional overhead due to cache invalidation requests and replica synchronization during write operations. These extra network calls slightly increase write latency compared to a system without caching. However, this overhead

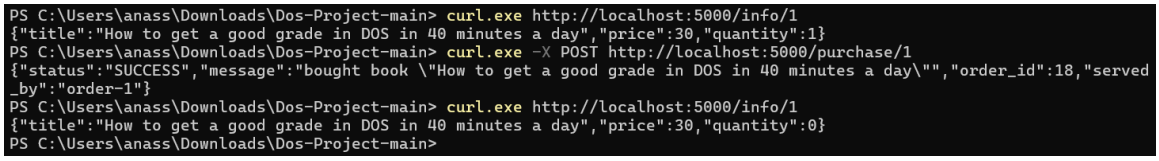
is acceptable given the significant performance improvement achieved for read-heavy workloads.

Comparison of Read Performance (With vs Without Cache):

Scenario	Average Response Time (ms)
With Cache	≈ 55.7 ms
Without Cache	≈ 65.8 ms

The comparison shows that enabling Redis caching reduces the average response time of read operations. Requests served from the cache avoid direct access to the catalog database, resulting in faster responses. Although the performance improvement is moderate, this is expected since the system is deployed locally using Docker containers. In real-world distributed deployments with remote databases and higher network latency, caching would provide a more significant performance gain.

3. Cache Invalidation Experiment (Write Operation)

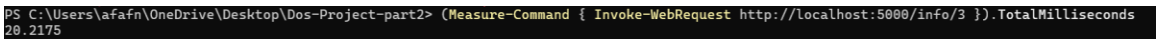


This figure demonstrates load balancing between multiple replicas of the order service. Consecutive purchase requests are handled by different replicas using a round-robin strategy, as indicated by the `served_by` *field.*

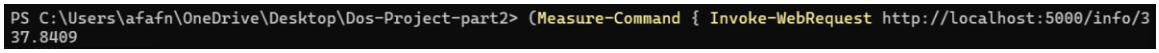
To verify cache invalidation, a read request was first issued to the /info/1 endpoint, causing the book data to be stored in the cache. After that, a purchase operation was executed, which triggered cache invalidation for the corresponding book entry. A subsequent read request to /info/1 returned updated data reflecting the reduced quantity, confirming that stale cache entries were successfully removed.

3.1 Cache Hit and Cache Miss Latency Measurement

The response time of a cached read request was measured using PowerShell:



Response time of a cache hit request to `/info/3`



Response time of a cache miss request to `/info/3` after cache invalidation

Scenario	Response Time (ms)
Cache Hit	≈ 20.2
Cache Miss	≈ 37.8
Difference	≈ 17.6

Cache Hit and Cache Miss Analysis:
To evaluate cache behavior, a read request to the `/info/3` endpoint was first issued while the data was present in the Redis cache. This request resulted in a cache hit with a response time of approximately **20 ms**, indicating that the data was served directly from the cache without accessing the catalog database.

After a purchase request was executed, the cache entry for the same book was invalidated to maintain consistency. A subsequent read request to `/info/3` resulted in a cache miss, with a response time of approximately **38 ms**, as the request was forwarded to the catalog service and database before the cache was updated again.

The results demonstrate the expected behavior of the caching mechanism: cache hits provide significantly lower latency compared to cache misses, while cache invalidation ensures that stale data is not returned after write operations.

4. Cache Invalidation & Consistency Analysis

To maintain data consistency, the system implements explicit cache invalidation for write operations. Whenever a purchase or update request is processed, the cached data related to the affected book is invalidated before the write operation is completed.

This ensures that subsequent read requests do not return stale data and always reflect the most recent state stored in the database. However, cache invalidation introduces additional overhead, as the system must perform extra operations to remove outdated cache entries and synchronize updates across replicas.

The experiment demonstrates that while caching significantly improves read performance, maintaining cache consistency during write operations requires careful coordination between services.

5. Discussion & Observations

The experimental results show that replication and load balancing successfully distribute requests among multiple service instances, improving system scalability and fault tolerance. The introduction of a Redis caching layer significantly reduces the average response time for read operations.

On the other hand, write operations such as purchases require cache invalidation and replica synchronization, which introduce additional overhead. Despite this cost, the overall

system performance benefits from caching outweigh the overhead, especially in read-heavy workloads.

6. Conclusion

These results highlight the trade-offs involved in designing distributed systems, where performance improvements often come at the cost of increased complexity in consistency management.

In this experiment, a distributed bookstore system was extended using replication, caching, and load balancing techniques. The use of multiple service replicas improved scalability and fault tolerance, while Redis caching significantly reduced response time for read operations.

Performance measurements demonstrated the effectiveness of caching, as cached requests were served faster than uncached ones. However, maintaining cache consistency during write operations introduced additional overhead due to cache invalidation and replica synchronization.

Overall, the experiment illustrates how distributed system techniques can enhance system performance and reliability while introducing important design challenges related to coordination and consistency.