2024

Distributed operating system

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

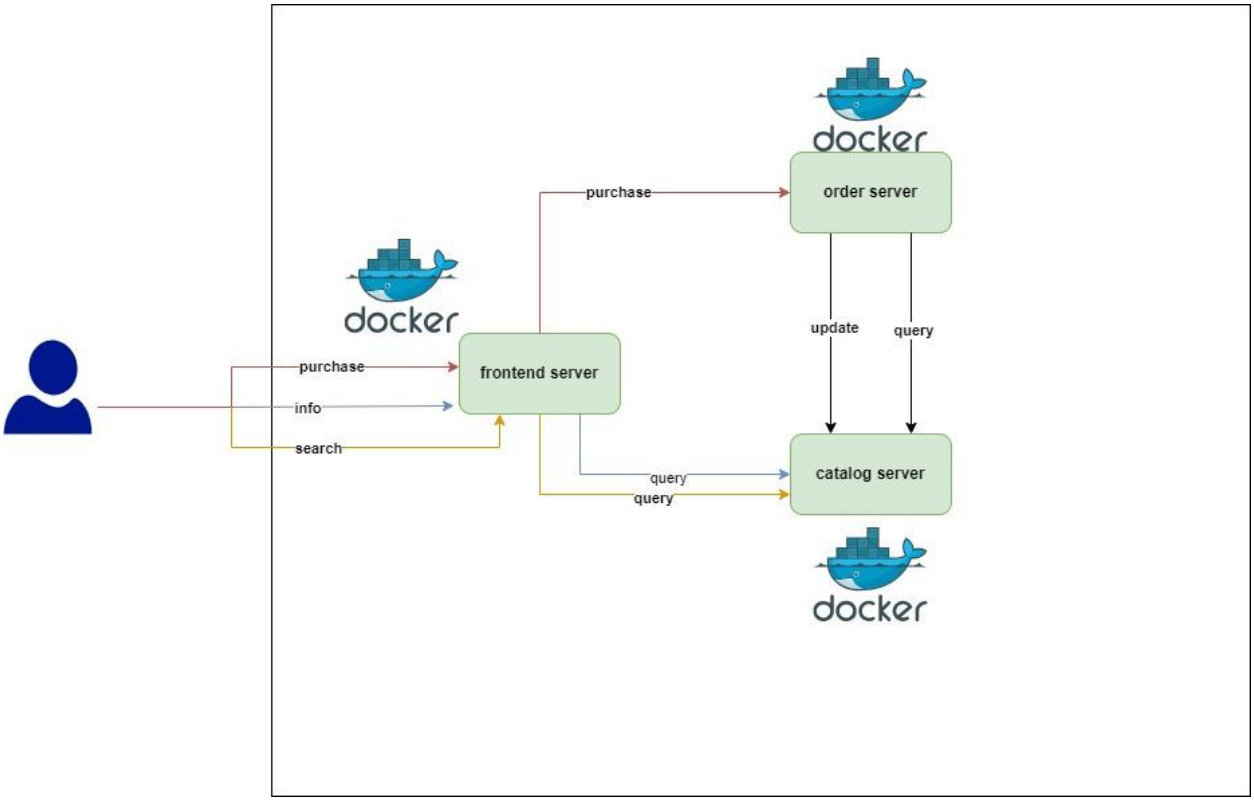
Naser Tayeh

Ameer mialeh

objectives

The primary objective of this project is to design, implement, and deploy Bazar.com, the World's smallest bookstore, with a focus on efficient and scalable two-tier web architecture utilizing micro services. The specific goals include:

Overall design



Design Description

This project aims to provide a comprehensive analysis of the utilization of replication, caching, and consistency in a Node.js project consisting of three servers (frontend, catalog, and order) and a database. The servers have been Dockerized, and replication is achieved through independent instances of the catalog and order servers. Caching, facilitated by cache-node in the frontend server, is employed to optimize response times. The report delves into an in-depth examination of the implementation details, challenges, and key considerations for ensuring consistency across the replicated servers.

Why do we need a container for each service?

Utilizing containers, such as Docker containers, for each replication in a distributed system provides numerous benefits. These advantages encompass enhanced isolation, scalability, and consistency. Containers offer a lightweight and encapsulated environment, guaranteeing that each replication instance functions autonomously without any interference.

What is the need for docker-compose?

The use of Docker Compose further streamlines the orchestration of these containers by defining the services, network configurations, and interdependencies in a single, easily maintainable YAML file. This approach not only simplifies the deployment process but also enhances scalability, as additional instances of each micro service can be effortlessly spun up to meet growing demand

Consistency and Load-Balancing Algorithm:

Ensuring consistent data across replicated servers is essential for maintaining data integrity and providing a seamless user experience. To achieve this, our implementation incorporates robust measures such as transactions and adherence to ACID properties in the database. These measures guarantee that all replicated servers have the same data state.
Additionally, we utilize a synchronization mechanism to keep all instances updated with the latest data changes. This ensures consistency and accuracy across all servers.
To further enhance our efforts, we have implemented a carefully designed load-balancing algorithm. This algorithm efficiently distributes incoming requests among different replication instances, optimizing resource utilization and preventing any specific server from becoming overloaded.
By adopting this comprehensive approach, we not only reinforce data consistency, but also contribute to improved system performance and user satisfaction.

How the program work?

The program operates seamlessly upon execution of the 'docker-compose build' command, initiating the orchestrated deployment of three distinct services within Docker containers.

1. **Front-End Server:**
   The front-end server, encapsulated within **frontend-container**, is designed to handle user requests for purchasing, information retrieval, and search operations. It operates on port 8000 and resides within its own local network in the Docker environment – this to make sure the 3 services is connected with each other - , having a it's own IP address. The front-end service, implemented as a micro service, plays a crucial role in the user interface, providing a seamless experience for customers interacting with Bazar.com. It communicates with both the catalog and order servers to fulfill user requests, establishing dependencies on these backend services for smooth operation. The service is initiated by the 'docker-compose up' command and gracefully stopped by 'docker-compose down', ensuring controlled lifecycle management ,note that this service is depend on order and catalog server , this mean this docker container will not run until it's sure that the order and catalog server is running.

2. **Order Server:**

   The order server, a distinct microservice running in order-container, specializes in processing purchase requests called from the front-end server. This service operates on port 8002 and is assigned its own IP address within the internal Docker network environment – this to make sure the 3 services is connected with each other -. The order server maintains a dependency on the catalog server, ensuring accurate verification of item availability before completing a purchase. Its functionality is critical in managing and updating the order database, contributing to the overall responsiveness and reliability of the book store's transactional processes. The service is initiated by 'docker-compose up' and gracefully stopped by 'docker-compose down'.,note that this service is depend on catalog server , this mean this docker container will not run until it's sure that the catalog server is running .

3. **Catalog Server:**
   The catalog server, implemented as a microservice within catalog-container, serves as the backbone for processing both purchase and query requests. Operating on port 8001, this service has its own IP address and is part of an internal Docker network. The catalog server is responsible for accessing an SQLite database to retrieve and update book-related data, including stock for each book. It supports query operations, responding to requests based on either book name or book number. The front-end and order servers depend on the catalog server to ensure accurate and up-to-date information for user interactions, making it a central component in the overall functionality of Bazar.com. Similar to the other services, it is started with 'docker-compose up' and stopped with 'docker-compose down', providing a streamlined approach to the container lifecycle.

design tradeoffs considered and made

- Microservices Architecture vs. Monolithic Design:
  The decision to adopt a microservices architecture introduces benefits such as modularity and scalability. However, it also introduces increased complexity in terms of service communication and potential overhead. The tradeoff was made to prioritize the advantages of microservices for flexibility and scalability while managing the associated complexities through effective communication patterns.

- Containerization with Docker:
  Choosing Docker for containerization provides portability and consistency across different environments. However, this decision comes with the tradeoff of increased resource consumption due to the need for separate containers for each service. The benefits of isolation and reproducibility, particularly in deployment, were deemed more significant than the incremental resource overhead.

- SQLite Database for Catalog Server:
  The selection of SQLite as the database management system for the catalog server involves a tradeoff between simplicity and scalability. While SQLite is lightweight and requires minimal setup, it may have limitations in handling extensive concurrent transactions compared to more robust database systems. This tradeoff was accepted to streamline development and deployment while recognizing that the book catalog is relatively small, making SQLite suitable for the task.

- Docker Compose for Orchestration:
  While Docker Compose simplifies the orchestration of multiple services, it introduces a tradeoff in terms of the learning curve for users unfamiliar with the tool. The decision to use Docker Compose was based on its ability to streamline the deployment process and manage the interdependencies between services efficiently, outweighing the initial learning curve.

- Cache for Frontend server:
  The frontend server employs cache-node in the caching layer to store frequently accessed data in memory, enhancing the overall performance of the system by reducing response times for repeated requests. To uphold data integrity, a robust cache invalidation strategy has been implemented. This strategy ensures that the cached data is refreshed whenever it becomes outdated or inconsistent with the database. By actively managing the cache, we guarantee the accuracy and timeliness of the information.

- Replication for Frontend server:
  The replication architecture encompasses the containerization of servers, specifically the frontend, catalog, and order components, utilizing Docker. This approach provides numerous benefits, including isolation, scalability, and streamlined deployment across various environments. The replication strategy entails generating multiple instances of both the catalog and order servers, each with their respective replication files. This methodology guarantees fault tolerance through the duplication of crucial server components while optimizing load distribution. By containerizing and replicating these servers, the system attains enhanced fault resilience.

possible improvements and extensions

- **Health Checks and Automatic Recovery:**

  Implement health checks for servers and containers to ensure continuous availability and reliability. In the event that a server or container becomes unhealthy, automatically redirect traffic to healthy instances.

- **Integration of a Robust Database System:**
  Consider transitioning from SQLite to a more robust and scalable database system, especially if the book catalog is expected to grow significantly. Database options like MySQL or PostgreSQL could provide better support for concurrent transactions and scalability.

- **Security Enhancements:**
  implement security best practices, including data encryption, secure communication protocols (HTTPS), and regular security audits. This ensures the protection of user data and prevents potential vulnerabilities.

The average response time

API	Time response with cache	Time response without cache
Info	1.4118ms	1.4118ms
Search	0.2767ms	97.3080ms
Purchase	Here just if item is out of stock or not found Item not found: 0.2658ms Item out of stock: 0.275199ms	54.53ms Item not found: 10.139679ms Item out of stock: 6.777005ms

What are the overhead of cache consistency operations?

Cache consistency operations, while crucial for data integrity, introduce overhead in parallel and distributed computing by requiring communication, synchronization, and additional processing, impacting system performance. Balancing this overhead is essential for optimizing the efficiency of these operations.

Some of the common overheads include:

Invalidation Overhead:

When a data base modifies a data item due to update query, it needs to inform the cache that might have a old copy of the same data to invalidate or update their copies. This involves sending invalidation messages, which introduce overhead in terms of communication and coordination.

Latency Overhead:

Cache consistency operations can introduce latency as processors may need to wait for updates or invalidations to propagate through the system before accessing shared data.

What is the latency of a subsequent request that sees a cache miss?

The latency of a subsequent request that encounters a cache miss is influenced by multiple factors within the system. Network latency, determined by the distance between the client and server and the quality of the network connection, contributes to the overall delay. On the server side, there are three primary factors: processing time, database access time, and cache warm-up time. Processing time involves computational tasks, while database access time depends on query complexity and the performance of the database server. Cache warm-up time occurs when data needs to be fetched from the underlying data source and populated into the cache after a cache miss. Additionally, other factors such as cache invalidation time, concurrency, system load, and the effectiveness of caching strategies and optimizations all contribute to the latency experienced by a subsequent request in the case of a cache miss. Continuous monitoring and optimization efforts are crucial for minimizing latency and ensuring optimal system performance.

How we can run the program

using a docker-compose only

You will find this in readme.txt

Using docker-compose just to build the images

Here you can use these command , to build the images , then build each container sperately

run# docker-compose build // to build the 3 images

run# docker images // to show you 3 images

docker network create --subnet=172.18.0.0/16 internal-network

run# docker run -it -d -v .\frontend-server\src:/app/src -p 8000:8000 --network internal-network --ip 172.18.0.6 --name frontend-container testing-frontend-server

run# docker run -it -d -v .\order-server\src:/app/src -p 8002:8002 --network internal-network --ip 172.18.0.8 --name order-container testing-order-server

run# docker run -it -d -v .\catalog-server\src:/app/src -p 8001:8001 --network internal-network --ip 172.18.0.7 --name catalog-container testing-catalog-server

Conclusion

In conclusion, this report emphasizes the successful implementation of replication, caching, and consistency in a Node.js project that incorporates Dockerized servers. The integration of replication ensures fault tolerance and enhances the distribution of workloads, ultimately optimizing system performance. Caching, facilitated by cache-node in the frontend server, significantly reduces response times for frequently accessed data. By combining replication and caching, the system becomes resilient and efficient, offering both high availability and responsiveness. Continuous monitoring and optimization are crucial to address evolving challenges and guarantee the ongoing success of the system in maintaining consistency, minimizing latency, and delivering enhanced user experiences.