DESIGN DOCUMENT LAB 3

SHREYA BIRTHARE
SIMRAN MALIK

INTRODUCTION:

This lab is an extension of the second lab. The Online Toy Store System is designed to handle people shopping from the toy store's catalog with various options such as querying the availability and price of the toy or buying the toy. The system is built to manage multiple client connections simultaneously, providing real time information on toy prices and stock levels. We employ a two-tier design for the Toy Store (a front-end tier and a back-end tier) using microservices at each tier. The front-end is implemented as a single microservice, while the back-end is implemented as two separate services: a catalog service and an order service. In addition to this, for this lab, the toy store system also adopts modern distributed systems design practices to ensure high performance and tolerance to failures. We implement caching and replication logics along with fault tolerant mechanisms to ensure high availability, consistent and robust systems. Finally, we deploy our services over the cloud using AWS and measure the performance of our system. This document outlines the system's goals, proposed solution, detailed design, along with evaluation metrics.

GOALS AND OBJECTIVES:

The goal of this lab is to:

- Design distributed server applications using a multi-tier architecture and microservices.
- Design interfaces for web applications.
- Learn how to implement a REST API server.
- Learn about caching, replication, and consistency.
- Learn about the concepts of fault tolerance and high availability.
- Learn about how to deploy your application on the cloud.
- Learn to measure the performance of a distributed application.
- Learn how to test a distributed application

SOLUTION OVERVIEW:

The Online Toy Store System is designed to provide a reliable and responsive service architecture by leveraging distributed systems principles. This section offers a detailed overview of our solution, emphasizing system architecture, technology stack, caching strategies, data replication, fault tolerance mechanisms, and performance evaluation criteria.

System Architecture:

The architecture is structured into two main tiers to optimize separation of concerns and scalability:

- Front-end Tier: Acts as the primary contact point for end-users, facilitating all client interactions with the system. It processes requests such as product query and purchase orders, and directs these requests to the appropriate back-end services. This tier is implemented as a robust microservice that efficiently manages session states, ensuring that multiple clients can interact with the system concurrently without performance degradation. Additionally, we also implement the LRU caching mechanism to enhance the front-end service by introducing an in-memory cache to reduce response times for toy queries. Initially empty, the cache fills as toy queries are made, checking the cache first before forwarding requests to the catalog service if necessary. Responses are then stored for future use. To maintain cache accuracy, especially after inventory changes due to purchases or restocking, the catalog server issues invalidation requests to the front-end, which then removes outdated items from the cache.

- Back-end Tier: This tier is split into two specialized microservices:
    - Catalog Service: The Catalog Service is crucial for managing inventory details such as product availability and pricing. It operates dynamically to handle queries and transactions:
        - Product Query Handling: When a product query is received, the service checks its local in-memory cache first. If the product is not found, it retrieves the details from its database and updates the cache.
        - Restocking and Cache Invalidation: The service automatically restocks products when their quantity falls to zero and sends cache invalidation requests to the front-end service. This ensures that the front-end cache reflects the latest inventory status.
        - Transaction Processing: When a purchase is made, the catalog updates the inventory quantities and triggers cache invalidation to maintain data consistency across the system.
    - Order Service: The Order Service is designed for high availability and fault tolerance through replication:
        - Replication Strategy: Multiple replicas of the Order Service are maintained, each with a unique identifier and its own database. This setup ensures redundancy and facilitates quick recovery in case of failures.
        - Leader Selection: The front-end service selects the leader among the order service replicas based on the highest available ID number, using health check requests to confirm availability.
        - Consistency and Synchronization: The leader node handles all write operations. Upon processing a new order, it propagates the details to the follower nodes to keep all replicas synchronized. If a node goes down and later recovers, it queries the other replicas to synchronize any missed transactions, ensuring no data is lost even in crash scenarios.

        - Automatic Failover: If the leader of the Order Service becomes unresponsive, the front-end service automatically initiates a new leader selection process to maintain continuous service availability.
        - Data Recovery: When a replica of the Order Service comes back online after a crash, it performs a catch-up operation to retrieve all updates missed during the downtime.

The entire system is deployed on AWS to benefit from cloud scalability and resilience. We assess system performance by measuring latencies and executing comprehensive unit tests to verify end-to-end functionality of our APIs.

DESIGN DETAILS:

1. **client.py:**
   The client.py script simulates a client for the Online Toy Store, designed to test the responsiveness and functionality of the front-end service by making automated product queries and placing orders. A session is created at client side to communicate with the front end service, to implement thread per session model. Sessions use keep-alive property and reuse the same connection to send all subsequent requests sent using this session. Using Python's requests library, the script performs HTTP operations against the system's REST API.
   Detailed Flow and Logic:
   - Initialization:
     The script starts by configuring the front-end service's host and port through environment variables, providing flexibility to adapt to different deployment environments without code changes.
   - Session Simulation:
     A session is represented by the perform_session() function, which runs a predefined number of iterations to simulate multiple interactions (product queries and potential purchases) with the store.
   - Product Query and Decision Logic:
     Products for querying are chosen randomly from a predefined list to simulate a variety of user interests. For each iteration, the script selects a product at random and sends a GET request to the /products/{product} endpoint. Upon receiving the product details, the script calculates the response time and aggregates this data for later analysis. It then uses a random number generator compared against a predefined probability (probability_order) to decide whether to place an order. This probability simulates the likelihood that a user decides to buy a product after viewing its details.
   - Order Placement:

If the decision to buy is affirmative, the script constructs an order payload with the product name and a randomly chosen quantity. A POST request is sent to /orders/ to attempt the purchase, and the script measures the response time for this transaction. The order response is checked for a valid order number, which, if present, is used in subsequent steps to verify the order status.

- Order Verification:
  For each successful order, the script later retrieves the order status using the order number via a GET request to /orders/{order_number}. This step confirms that the order has been processed correctly and logs the time taken to receive the response.
- Probability Calculation
  The decision to place an order is determined by comparing a random float from 0 to 1 against the probability_order. If the random number is less than the probability (set to 0.4 or 40% by default), an order is placed. This stochastic approach introduces realistic unpredictability into the simulation, mimicking real-world user behavior where not every product query results in a purchase.
- Performance Metrics
  The script calculates average response times for product queries, order placements, and order verifications by aggregating individual timings over the session. These metrics provide insights into the system's performance, helping identify potential delays or inefficiencies.

2. **front_end_service.py:**
   The front_end_service.py script is the gateway through which client requests are processed in the Online Toy Store system. This script is crucial for interfacing with both the Catalog and Order services, and for maintaining a consistent and responsive user experience. This script utilizes Python's http.server module to handle HTTP requests with a multithreaded server, ensuring that multiple client requests can be handled simultaneously without blocking. The service uses a Least Recently Used (LRU) cache to improve response times and reduce load on the backend services.
   Key Components and Logic:
   - Configuration and Setup:
     Hostnames and ports for the front-end, catalog, and order services are configured through environment variables, allowing flexible deployment configurations. Order service replicas are mapped with their respective host and port, facilitating leader election and request routing.
   - LRU Cache Implementation (LRUCache class):
     A thread-safe cache implemented using OrderedDict from Python's collections module. The cache has a specified capacity, and it evicts the least recently used item when the capacity is exceeded. Operations include get (retrieve and mark as

recently used), put (add or update and mark as recently used), and invalidate (remove an item based on external events, such as product restocking).

- API Endpoints and Request Handling:
  - Product Queries (/products/{product_name}):
    Queries are first checked against the cache. If a cache miss occurs, the request is forwarded to the Catalog Service. Responses from the Catalog Service are cached for future requests, and successful responses are returned to the client.
  - Order Operations (/orders/ and /orders/{order_number}):
    For new orders, requests are forwarded to the leader of the Order Service replicas. The leader is determined by a predefined election mechanism which checks the availability of replicas starting from the highest predefined ID. Order status queries are routed directly to the leader to fetch the status of a specific order.
  - Cache Invalidation (/invalidate/{product_name}):
    Allows external systems (like the Catalog Service) to invalidate cache entries, ensuring data consistency across the system.
- Leader Election and Notification:
  The system checks each Order Service replica in descending order of their IDs to determine the leader. Once a leader is confirmed available, it notifies all other replicas of the leader's status, ensuring that they direct write operations appropriately.
- Threading Model:
  The server operates on a multi-threaded model using ThreadingHTTPServer, which spawns a new thread for each incoming request, allowing it to handle high concurrency without service degradation.
- Exception Handling and Error Reporting
  Comprehensive error handling is implemented to manage exceptions such as connection errors, timeouts, and unexpected response codes from downstream services. Errors are logged with appropriate HTTP status codes and messages, providing clear feedback to clients and aiding in troubleshooting.

3. **catalog.py:**
   The Catalog Service in the Online Toy Store system plays a pivotal role in managing product inventory and serving product information. It operates as a standalone microservice, interfacing with other services to ensure consistency and reliability in inventory data.
   Detailed Functionality and Flow:
   - Initialization and Configuration:

    - Environment Setup: The service initializes using environment variables to define its operational parameters, such as the host and port it listens on and the location of its catalog file.
    - Catalog Initialization (load_catalog()): This function is responsible for loading the product catalog from a CSV file into a dictionary. If the file doesn't exist, it creates a default catalog with pre-defined products and writes this back to a new CSV file, ensuring the service can operate even if the initial file is missing.
- Catalog Data Management:
    - Product Data Handling: The catalog data is stored in a dictionary where each key is a product name, and the value is another dictionary containing the product's price and quantity.
    - Restocking Mechanism (restock_catalog()): This function runs in a separate thread, continuously monitoring the inventory levels. When a product's quantity drops to zero, the function replenishes the stock and triggers a cache invalidation request to ensure that the front-end service provides up-to-date information.
- HTTP Request Handlers:
    - GET Handler (do_GET() in CatalogRequestHandler): This method handles incoming GET requests for product information. It parses the requested product name from the URL, retrieves the product details from the catalog, and sends the information back to the client. If the product is not found, it returns a 404 error.
    - Caching Strategy: Before serving a product query, it checks an in-memory cache (implemented using an LRU cache mechanism). If the product is in the cache, it serves the cached data; otherwise, it fetches the data from the main catalog.
    - POST Handler (do_POST() in CatalogRequestHandler): Handles purchase requests from clients. It reads the product name and quantity from the request body, checks inventory levels, and updates the catalog if the product is available. If the update is successful, it also writes the new inventory levels back to the CSV file and sends a cache invalidation request.
- Cache Invalidation (send_invalidation_request()):
  After any operation that modifies the product data (like a purchase or restocking), this function is called to notify the front-end service to invalidate its cache for the affected product. This ensures that subsequent queries for this product reflect the updated data.
- Concurrency and Thread Safety:

All operations that modify the catalog (like handling purchases or restocking) are protected with a threading lock (LOCK). This prevents race conditions and ensures that multiple operations do not interfere with each other, maintaining data integrity across the service.

- Error Handling and Logging:
  The service is designed to catch and handle various exceptions such as file I/O errors, network failures, and parsing errors. Each error type is logged appropriately, providing clear diagnostics for maintenance and troubleshooting.

4. **order.py:**
   The order.py script is crucial for the robust and efficient operation of the Online Toy Store, particularly in managing order processing and ensuring system resilience through advanced caching, replication, and failure recovery mechanisms.
   System Initialization and Configuration:
   - Service Setup: Initializes using environment variables that define host, port, and paths for order data. Each replica has a unique configuration to support a distributed architecture.
   - Order Data Initialization: Sets up a CSV file for logging orders per replica, facilitating persistence and recovery of order data.

   Core Functionalities and Advanced Features:
   - Order Processing and Caching:
     - Order Logging and Number Generation:
       generate_order_number(): Secures a unique identifier for each order, synchronizing access via a global lock to prevent concurrency issues.
       log_order(): Records orders to a CSV file. If executed by the leader replica, it triggers replication across follower replicas.
     - Caching Mechanism: Not directly handled in order.py but interacts closely with the front-end's caching strategy. Ensures the front-end cache remains consistent with the latest data by sending invalidation requests after each update.
   - Replication and Consistency:
     - Data Propagation (propagate_order_to_followers()): Ensures all replicas maintain up-to-date data by replicating new orders to follower nodes. Uses threading to manage simultaneous data transmissions efficiently.
     - Follower Management (get_followers()): Identifies and manages follower nodes, excluding the leader from receiving its own updates.
   - Handling Failures and Recovery:
     - Leader Election and Notification: Implements a method to determine the leader based on the highest available replica ID, ensuring there's always a designated leader to handle write operations.

    - Recovery and Data Synchronization:
      request_missed_orders(): Checks for any orders missed during outages by consulting higher ID replicas upon startup or recovery.
      fetch_missed_orders(): Retrieves orders that were not logged by a recovering replica, allowing it to synchronize its state with the system.

Detailed Scenario Handling:

- Cache Consistency: The order service communicates with the front-end to invalidate cache entries whenever products are bought or restocked. This ensures that the front-end's cache reflects the most current data, improving query response times and system efficiency.
- Replication Strategy: Utilizes a replication strategy where one node acts as the leader to handle all write operations, while other nodes act as followers. The leader node is determined by its ID, with the front-end service selecting the highest available ID as the leader. Upon receiving a new order, the leader logs the order and propagates the details to followers to ensure data consistency across the system.
- Leader Selection and Health Checks: The front-end service periodically checks the health of the leader. If the leader is unresponsive, the front-end initiates a leader reselection process, starting from the replica with the highest ID and moving to lower IDs until an active leader is found.
- Fault Tolerance and Crash Recovery: The system is designed to handle crash failures by ensuring that other replicas can continue processing requests even if the leader fails. Upon recovery, a crashed replica retrieves the latest order number from its log file and queries other replicas to synchronize any missed data, ensuring no orders are lost and maintaining the integrity of the order log.
- Error Handling and System Robustness: Implements comprehensive error handling for network failures, data synchronization errors, and file access issues, ensuring the system remains robust under various failure scenarios. Logs detailed error messages and system states to facilitate troubleshooting and maintenance.

TESTING:

The unitTests.py file is structured to rigorously test the microservices of the Online Toy Store system through a series of unit tests that cover various scenarios. Each test is crafted to simulate specific interactions with the front-end, catalog, and order services, verifying the correctness and reliability of their responses. Successful passing logs can be found in ../Documents/unitTestResults.JPG

FrontEndServiceTests:

- test_front_end_query_existing_product: Ensures that querying an existing product via the front-end returns the correct product details.

- test_front_end_query_non_existing_product: Tests the response to a query for a non-existent product, expecting a 404 status code.
- test_front_end_place_order_successfully: Checks that a valid order can be placed through the front-end and verifies the presence of an order number in the response.
- test_front_end_quantity_more_than_available: Verifies that the system correctly handles an order request exceeding available stock with an appropriate error status.
- test_front_end_place_order_for_non_existing_product: Tests ordering a product that does not exist and expects a 404 error.
- test_front_end_query_existing_order_number: Confirms that querying an existing order number retrieves the correct order details.
- test_front_end_query_nonexisting_order_number: Ensures the system correctly handles queries for non-existent order numbers by returning a 404 status.
- test_health_check: Checks the health endpoint of a replica, expecting a healthy status.
- test_notify_leader_to_replica: Verifies that a replica can be notified of leader information successfully.

CatalogServiceTests:

- test_retrieve_product_info_directly: Confirms that product information can be directly retrieved from the catalog service.
- test_buy_product_successfully: Checks if a product can be purchased successfully through the catalog service.
- test_buy_non_existent_product: Ensures that an attempt to buy a non-existent product returns an appropriate error.

OrderServiceTests:

- test_place_order_successfully: Validates that an order can be placed successfully with the order service and checks for an order number in the response.
- test_query_existing_order_number: Tests the retrieval of existing order details from the order service.
- test_query_nonexisting_order_number: Ensures that queries for non-existent order numbers are handled correctly with a 404 status.
- test_quantity_more_than_available: Verifies that an order exceeding the available quantity is handled correctly with an error status.
- test_place_order_for_non_existing_product: Confirms that orders for non-existent products are correctly rejected with a 404 status.
- test_propagate_order_details_to_follower: Tests the propagation of order details to a follower node, expecting successful replication.
- test_missed_orders: Checks how the system handles potentially missed orders when a replica comes back online, verifying that it can request and retrieve missed orders appropriately.

LATENCY EVALUATION:
See the latency evaluation docs: ../Documents/latency/latencyEval.pdf

Part 5:

1. **Front end raft service performs health checks of order replica with highest id whenever an order query or purhase query comes in. When replica responds, term variable is updated and replica_id, and stores in front end log. The leader is elected, and notified about the term**
2. **When purchase request comes in, leader updates its raft log and replicates to followers as well, and asks them if they are willing to write. It majority do not reply, it invalidates the raft log and sends not enough replicas to process order error to client. It then checks if requested item is in stock. If yes, it places order else it sends corresponding error to client and invalidates that raft log and asks replicas to do the same. When order is placed, it replicates the successful order details in its order log and sends the same to replicas**
3. **Whenever an order replica starts, it synchronizes its raft and order logs with the leader.**