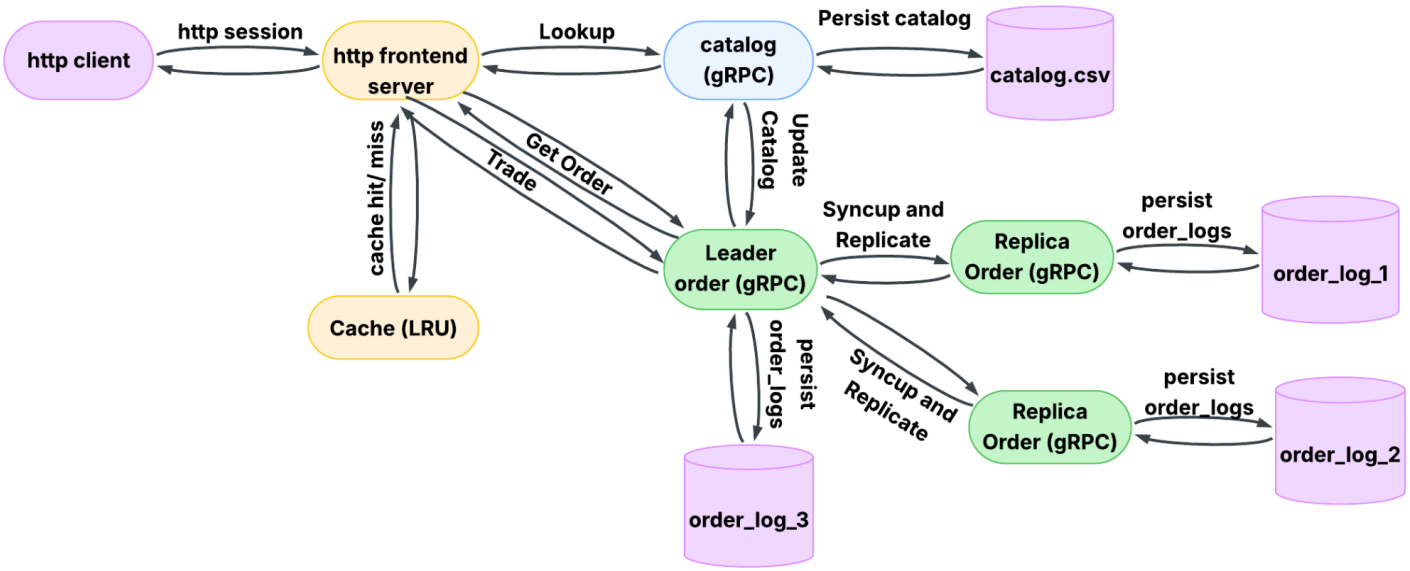
Introduction

In this lab, we design a distributed, multi-tier stock trading server capable of handling high volumes of concurrent lookup and trade operations. The system is decomposed into three microservices: front-end, catalog, and order. We enhance our system by introducing **caching**, **replication**, and **fault tolerance**. The goal is to achieve improved latency, availability, and robustness under failures.

- An LRU cache at the front-end.
- Replicated order services with a leader-follower protocol.
- Automatic failover and state synchronization for crash recovery.

Solutions Overview



Our system consists of the following microservices, all launched on a t2.medium EC2 instance using Bash scripts. Services are started using **native_build.sh** and failure scenarios are simulated using **simulate_crashes.sh.**

- **Front-end Service**
  Exposes REST APIs and manages session persistence, caching, and failover logic of Leader Order Service triggering leader re-election.
- **Catalog Service**
  Maintains an in-memory dictionary of stock data initialized from catalog.csv. Responds to lookup and trade requests received via gRPC. Upon trade, updates the stock state and sends cache invalidation requests to the front-end. Periodically writes the updated stock data to disk using a background thread.

- **Order Service (Replicated)**

Runs 3 replicas, each on a unique port. The replica with the highest available ID is selected as leader. The leader processes all trade and order requests, generates a transaction number, and propagates the transaction to follower replicas. Followers store the replicated data and synchronize any missed transactions on restart using the SyncUp mechanism. Each replica writes its order log to disk asynchronously.

APIs/Interface

Frontend (REST APIs)

GET /stocks/<stock_name>: Returns stock info (from cache or from catalog)

- Input: GET /stocks/GameStart
- Output (Success): { "data": { "name": "GameStart", "price": 15.99, "quantity": 100 } }
- Output (Error): { "error": { "code": 404, "message": "error message" } }

GET /orders/<transaction_number>: Retrieves transaction details from the Order leader.

- Input: GET /orders/42
- Output (Success): { "data": { "order_num": 42, "name": "GameStart", "type": "buy", "quantity": 1 } }
- Output (Error): { "error": { "code": 404, "message": "Transaction number not found" } }

POST /orders: Handles trade requests sent by clients and forwards it to the Order leader replica.

- Input: POST /orders, Body: { "name": "GameStart", "quantity": 1, "type": "sell" }
- Output (Success): { "data": { "transaction_number": 42 } }
- Output (Error): { "error": { "code": 404, "message": "error message" } }

DELETE /delete/ : Used by Catalog service to remove a stock from cache.

- Input: DELETE /delete/GameStart
- Output: { "code": 200, "message": "Cache invalidated" }

Implementation Details

- Thread-per-session model with persistent HTTP/1.1 sockets
- Uses OrderedDict to implement LRU caching
- Automatically retries trade or order lookup in case of replica failure after re-electing leader
- Leader elected via gRPC Heartbeat check
- Libraries Used: http.server, socketserver, json, grpc, grpcio-tools, readerwriterlock, os

Catalog (gRPC APIs)

Lookup: Handles stock lookup requests from the frontend and returns price and quantity details.

- Input**:** LookupRequest { string stock_name = 1; }
- Output (Success): LookupResponse { code: 200, name: "GameStart", price: 15.99, quantity: 100 }
- Output (Error): LookupResponse { code: 404, message: "stock not found" }

Trade: Handles buy/sell trade requests from order and updates stock quantity and volume.

- Input: TradeRequest { name: "GameStart", number_of_items: 1, type: "sell" }
- Output (Success): TradeResponse { code: 200 }
- Output (Error); TradeResponse { code: 404 }

Implementation Details

- In-memory stock dictionary initialized from catalog.csv

- Background thread writes updated stock data to disk every second
- After successful trade, sends HTTP DELETE to frontend to invalidate cached stock
- Thread safety enforced using fair read-write locks (RWLockFair)
- Libraries Used: grpc, concurrent.futures, csv, threading, readerwriterlock, os, requests

Order (gRPC APIs)

Order: Validates and processes trade requests. Leader propagates transactions to follower replicas.
- Input: OrderRequest { name: "GameStart", number_of_items: 1, type: "buy" }
- Output (Success): OrderResponse { code: 200, transaction_num: 42, message: "order placed successfully" }
- Output (Error): OrderResponse { code: 404, message: "invalid transaction type/invalid stock name/num stocks traded should be non negative/not enough stocks left to buy" }

GetOrderDetails: Returns details for a specific transaction number.
- Input: GetOrderDetailsRequest { transaction_num: 42 }
- Output (Success): GetOrderDetailsResponse { code: 200, transaction_num: 42, name: "GameStart", type: "buy", volume_traded: 1 }
- Output (Error): GetOrderDetailsResponse { code: 404, message: "Transaction number not found" }

Heartbeat: Used by the frontend to detect if an Order replica is alive.
- Input: Empty {}
- Output: HeartbeatResponse { code: 200 }

NotifyReplica: Notifies a replica about the new elected leader.
- Input: NotifyReplicaRequest { leader_id: 3 }

ReplicateOrder: Used by the leader to replicate the trade to followers.
- Input: ReplicateOrderRequest { transaction_num: 42, name: "GameStart", number_of_items: 1, type: "buy", leader_id: 3 }

SyncUp: Used by crashed replicas to synchronize missed transactions after restart.
- Input: SyncUpRequest { transaction_num: 37, service_id: 2 }
- Output: SyncUpResponse { orders: [OrderDetails { ... }, ...] }

Implementation Details
- Three replicas run independently; leader processes and propagates trades
- Transaction numbers initialized from local CSV logs
- Followers synchronize missed logs after restart using SyncUp
- Background thread periodically writes in-memory logs to disk
- Fair read-write locks protect shared state (RWLockFair)
- Libraries Used**:** grpc, concurrent.futures, csv, threading, readerwriterlock, os, dotenv

Design Choices

- Re-using Lab2 code and building upon it for this lab.
- Used gRPC for efficient communication between services.
- Used thread-per-session model in front-end to handle each client separately.
- Used fair read-write locks to avoid starvation and handle concurrency.

- Explicitly removed cache entries after trades for consistency.
- Used a leader-follower approach to keep order logs consistent.
- Restarted replicas catch up using a SyncUp mechanism.
- The frontend uses an OrderedDict to implement an LRU cache for stock lookups.
- A separate simulate_crashes.sh script allows deterministic testing of fault scenarios by killing and restarting specific replicas.
- When an Order replica restarts. It assumes that the remaining replicas are consistent and simply pings each of them. The first responsive peer is used as the source for missing transactions during SyncUp, minimizing coordination overhead while ensuring eventual consistency.
- Threadpool size of 3 in both order and catalog microservices (keeping less than num of clients for realistic evaluation)

Testing

Functional Testing

Tested individual microservice and the entire application by developing a comprehensive suite of test cases as follows:

- test_lookup_valid and test_lookup_invalid check correctness of stock lookup requests.
- test_trade_buy_valid and test_trade_sell_valid validate updates to quantity and volume for valid trades.
- test_trade_buy_invalid ensures proper error handling when buying more stock than available.
- test_frontend_lookup_valid and test_frontend_lookup_invalid validate end-to-end stock lookup via HTTP.
- test_frontend_cache_hit verifies that repeated lookups hit the LRU cache.
- test_frontend_cache_eviction confirms eviction behavior once cache exceeds capacity.
- test_post, test_get_order_valid test valid trade and order retrieval.
- Multiple error scenarios (test_post_invalid_url, test_post_wrong_stockname, test_post_wrong_type, test_post_negative_quantity, test_post_large_quantity) validate robustness against malformed requests.
- test_post_transaction_increments ensures transaction numbers are correctly incremented and persisted.
- test_leader_logs_trade_replicates_followers verifies that trades handled by the leader are replicated across all replicas.
- test_order_csvs_same ensures consistency of order_log_<id>.csv files across replicas.
- test_replica_sync_after_crash tests that a restarted replica performs SyncUp and correctly catches up with missed transactions.
- test_leader_crash_and_recovery simulates a leader crash and confirms that the system re-elects a new leader and maintains transaction continuity.

Load Testing

- Ran up to 5 clients at once and tested latency for different types of requests (get stocks, get order details and trade order)
- Measured response time for different trade probabilities (p from 0% to 80%).
- Compared the system with and without caching.
- Verified cache eviction with LRU policy.

Crash Testing

- Randomly stopped replicas during operation.
- Verified leader re-election worked correctly.
- Checked that restarted replicas synchronized with others.
- Ensured all replicas had consistent logs at the end.

References

- https://docs.python.org/3/library/collections.html#collections.OrderedDict
- https://docs.aws.amazon.com/ec2/
- https://www.youtube.com/watch?v=hFNZ6kdBgO0&ab_channel=howCode
- https://obinexuscomputing.medium.com/python-is-a-great-language-for-building-servers-especially-for-quick-projects-and-prototyping-71e0769bf738
- https://jasonstitt.com/perfect-python-live-test-coverage
- https://stackoverflow.com/questions/10415028/how-can-i-get-the-return-value-of-a-function-passed-to-multiprocessing-process
- https://en.wikipedia.org/wiki/Hypertext_Transfer_Protocol
- https://docs.docker.com/engine/reference/builder/
- https://docs.docker.com/compose/compose-file/
- https://docs.docker.com/storage/volumes/