Project Update 1: Real-Time Price Aggregator

Overview

This update summarizes my work on the **Real-Time Price Aggregator** project for the CS6650 Scalable Distributed Systems course. The project is a distributed system that aggregates real-time financial asset prices from multiple mock exchanges, caches them in Redis for low-latency access, and stores historical data in DynamoDB for persistence. I completed the core functionality, focusing on scalability, automation, and performance, with updates to the design based on feedback and testing.

Literature Review

To inform the design of my price aggregation system, I researched the following platforms:

- **CoinGecko (https://www.coingecko.com/)**: A leading cryptocurrency data aggregator that fetches prices from multiple exchanges. This inspired my initial design to aggregate prices, though I simplified the aggregation to a mock price for this phase.
- **TradingView (https://www.tradingview.com/)**: A real-time financial data platform that visualizes price trends. It highlighted the importance of low-latency data access, leading me to implement Redis caching with a 5-minute TTL to balance freshness and performance.

These platforms helped me understand key trade-offs in distributed systems, such as latency vs. data freshness and availability vs. consistency, which shaped my design decisions.

Project Backlog

Completed Tasks

- **System Design**: Architected a microservices-based system with three mock exchanges, a price aggregation server, Redis caching, and DynamoDB storage.
- **Mock Exchanges**: Implemented three mock exchanges in Go (`mocks/mock_server.go`) to simulate real-world trading platforms, supporting 10,000 assets defined in `symbols.csv`.
- **Price Fetching**: Built the main server (`cmd/main.go`) to fetch mock prices (random prices for simplicity) and manage caching/storage.
- **Redis Caching**: Integrated Redis (`internal/cache/redis.go`) to cache prices for low-latency access, with a 5-minute TTL to ensure data freshness.
- **DynamoDB Storage**: Integrated DynamoDB (`internal/storage/dynamodb.go`) for persistent storage, with automatic table creation on startup to simplify setup.
- **Asset Management**: Added `symbols.csv` with 10,000 assets, with validation to ensure only supported assets are queried or refreshed.
- **Automation**: Used Docker Compose (`docker-compose.yml`) to automate the deployment of Redis, three mock exchanges, and the main server, allowing all services to start with a single command.
- **API Design**: Defined a REST API with two endpoints:
  - `GET /prices/{asset}`: Retrieve the cached price of an asset.
  - `POST /refresh/{asset}`: Manually refresh the price of an asset.
- **Error Handling**: Updated error responses:
  - 400: Invalid asset symbol (not in `symbols.csv`).

    - 404: Asset not found in cache.
- **Documentation**: Documented the API using OpenAPI 3.0 (`openapi.yaml`) and provided detailed setup instructions in the README.
- **Testing**: Conducted manual tests with `curl` to verify API functionality, Redis caching, and DynamoDB storage, and planned load testing with JMeter.

Remaining Tasks

- **Load Testing**: Complete load testing with Apache JMeter to verify performance goals (1000+ queries/sec, <100ms response time, >90% Redis cache hit rate).
    - `POST /refresh/{asset}`: Simulate 10 thread groups (each with 1000 assets), 2-second delay.
    - `GET /prices/{asset}`: Test with 1000 concurrent users, random asset selection.
- **Screenshots**: Add screenshots of API responses, DynamoDB data, and JMeter results to this report for visual documentation.
- **AWS Deployment**: Deploy the system to AWS EC2 and test in a production-like environment.
- **Final Documentation**: Polish the documentation and prepare for submission.

Future Work

- **Kafka Integration**: Add Kafka for distributed data ingestion to decouple price fetching and processing, improving scalability for real-time price streams.
- **Monitoring**: Integrate Prometheus and Grafana for monitoring system performance and visualizing price trends.
- **AWS Deployment**: Deploy to AWS ECS/EKS with a load balancer to handle production-level traffic.
- **Advanced Features**: Support more assets and add historical price charts using DynamoDB data.

Code

The project is written in Go and consists of the following key components:

- **Mock Exchanges** (`mocks/mock_server.go`): Simulate three exchanges running on ports 8081, 8082, and 8083, providing mock price and timestamp data for 10,000 assets.
- **Main Server** (`cmd/main.go`): Loads assets from `symbols.csv`, fetches prices, and manages caching/storage.
- **Fetcher** (`internal/fetcher/fetcher.go`): Fetches mock prices (random prices for simplicity).
- **Redis Cache** (`internal/cache/redis.go`): Caches prices for low-latency access with a 5-minute TTL.
- **DynamoDB Storage** (`internal/storage/dynamodb.go`): Stores historical price data with automatic table creation on startup.
- **API Handler** (`internal/api/handler.go`): Implements the REST API endpoints with proper error handling and asset validation.

Code Snippet: Asset Validation

Here's how asset validation is implemented in `internal/api/handler.go`:

```
symbolLower := strings.ToLower(symbol)
// Check if asset is supported (in CSV)
if !h.supportedAssets[symbolLower] {
    respondWithError(w, http.StatusBadRequest, "Invalid asset symbol")
    return
}
```

Documentation

- **README**: Includes setup instructions, API design, data structure, testing steps, and EC2 deployment guide. GitHub link
- **OpenAPI Spec** (`openapi.yaml`): Documents the API endpoints with updated error responses:
  - `GET /prices/{asset}`: Returns the cached price or 404 if not found.
  - `POST /refresh/{asset}`: Forces a price refresh and updates cache/storage.
- **Data Structure**: Documented the DynamoDB table structure in the README:
  - Table Name: `prices`
  - Fields: `asset` (partition key), `timestamp` (sort key), `price`, `updated_at`.

Testing

Manual Testing

- **Mock Exchanges**:

  ```
  curl http://localhost:8081/mock/ticker/btcusdt
  ```

  Response:

  ```
  {
    "symbol": "btcusdt",
    "price": 79850.12,
    "timestamp": 1696118400
  }
  ```

- **API Endpoints**:
  - Refresh price:

    ```
    curl -X POST http://localhost:8080/refresh/btcusdt
    ```

    Response:

    ```
    
    ```

```
{
  "message": "Price for btcusdt refreshed"
}
```

- Get price:

```
curl http://localhost:8080/prices/btcusdt
```

Response:

```
{
  "asset": "btcusdt",
  "price": 79450.12,
  "last_updated": 1696118400
}
```

- Invalid asset:

```
curl http://localhost:8080/prices/bbb
```

Response:

```
{"msg": "Invalid asset symbol"}
```

- **DynamoDB Storage**:

```
aws dynamodb scan --table-name prices --region us-west-2
```

Output:

```
{
  "Items": [
    {
      "asset": {"S": "btcusdt"},
      "timestamp": {"N": "1696118400"},
      "price": {"N": "79450.12"},
      "updated_at": {"N": "1696118405"}
    }
  ]
}
```

Load Testing (Planned)

- Planned to use Apache JMeter to test the system with 10,000 assets:
  - `POST /refresh/{asset}`: Simulate 10 thread groups (each with 1000 assets), 2-second delay.
  - `GET /prices/{asset}`: Test with 1000 concurrent users, random asset selection from `symbols.csv`.
- Performance goals:
  - Throughput: 1000+ queries/sec
  - Response Time: <100ms for cached requests
  - Redis Cache Hit Rate: >90%
- Test plan is included in the README under "Testing the System."

Challenges Overcome

- **Design Simplification**: Simplified the design by removing automatic price updates and focusing on manual `POST`, with plans to add automation later.
- **Asset Management**: Initially supported only 10 assets; expanded to 10,000 assets using `symbols.csv` for scalability.
- **Error Handling**: Improved error responses (400, 404) to handle invalid assets and cache misses.
- **Learning Curve**: Quickly learned Redis, DynamoDB, and Docker Compose to build a functional system, leveraging Grok's guidance to accelerate the process.

Skills Gained

- **Distributed Systems**: Designed a microservices-based system with caching and persistent storage.
- **Docker and Automation**: Used Docker Compose to automate service deployment, making the system easy to run.
- **API Design**: Created a REST API with proper error handling, documented with OpenAPI 3.0.

Future Work

- **Kafka Integration**: Add Kafka for distributed data ingestion to decouple price fetching and processing, improving scalability for real-time price streams.
- **Monitoring**: Integrate Prometheus and Grafana for monitoring system performance and visualizing price trends.
- **AWS Deployment**: Deploy to AWS ECS/EKS with a load balancer to handle production-level traffic.
- **Advanced Features**: Support historical price charts using DynamoDB data.

Conclusion

This project showcases my ability to design, implement, and test a scalable distributed system under tight constraints. The updated design with 10,000 assets, Redis caching, and DynamoDB storage demonstrates my understanding of distributed systems principles. I learned valuable skills in Go programming, microservices architecture, Docker automation, and API design, which I believe will be highly applicable in a professional software engineering role. The automation with Docker Compose, integration of Redis and

DynamoDB, and focus on performance make this project a strong demonstration of my technical and problem-solving abilities.