Distributed Marketplace

Agenda

1. Executive Summary

This project is a self contained demonstration of a distributed marketplace. It models multiple **Seller** instances, a redundant **Marketplace** service acting as broker and a load–generating **Client**. Every process can run either directly on the host or inside a Docker container. The code serves as a playground for studying reliable distributed transactions using the SAGA pattern and ZeroMQ based messaging.

A SAGA Pattern is a design pattern used to manage complex, long-running transactions in distributed systems, which breaks down a large transaction into a series of smaller, local transactions that can be independently rolled back if needed.
This is where ZeroMQ supports as a high-performance asynchronous messaging library, aimed at use in distributed or concurrent applications.

2. Setup

The code inside this repository can either be run through directly through scripts or through docker. Both ways are explained in the following.

2.1 Prerequisites

- Java 24 (tested); requires at least Java 19 because the services rely on virtual threads
- Maven (for building from source)
- Docker and Docker Compose (optional for container based runs)

Maven is a build automation tool used primarily for Java projects. It provides a standard way to build, test, and deploy Java applications.

2.2 Local Execution

The project includes three pre-built JAR files in the repository root, each representing a component of the distributed marketplace simulation:

1. `marketplace.jar`:
    - Main class: com.example.marketplace.Marketplace
    - Role: Coordinates the SAGA workflow across sellers
    - Execution: `java -jar marketplace.jar [properties]`
    - Example-Properties: `marketplace.properties`
2. `seller.jar`:
    - Main class: com.example.seller.Seller
    - Role: Manages product inventory and handles requests
    - Execution: `java -jar seller.jar [properties]`
    - Example-Properties: `seller.properties`
3. `client.jar`:
    - Main class: com.example.client.Client
    - Role: Generates orders and receives notifications
    - Execution: `java -jar client.jar [arguments]`
    - Example-Properties: `client.properties`

Each JAR is self-contained, including all necessary dependencies such as ZeroMQ libraries. Despite containing similar classes, each JAR's behavior is determined by its main class, allowing it to fulfill its specific role in the simulation.

To run a component, use the scripts provided below. The JARs will automatically set up their environment and start their specific simulation role based on the provided arguments.

```
java -jar seller.jar SELLER-1 'tcp://*:5561' seller.properties | tee -a
logs/SELLER-1.log
```

Creates the seller 'SELLER-1' on port 5561 with `seller.properties`. Logs are being saved to the file `SELLER-1.log`.

```
java -jar marketplace.jar MARKETPLACE-1 marketplace.properties | tee -a
logs/MARKETPLACE-1.log
```

Creates the marketplace 'MARKETPLACE-1' with `seller.properties`. Logs are being saved to the file `MARKETPLACE-1.log`.

```
java -jar client.jar CLIENT-1 client.properties
```

Creates the client 'CLIENT-1' with `client.properties`. Logs are being saved to the file `CLIENT-1.log`.

2.3 Running via Scripts

The repository contains helper scripts for starting all components. A short excerpt from `start-system-new.sh` shows how the sellers and marketplaces are launched:

```
start_component() {
    local name="$1"
    local jar="$2"
    shift 2
    local args=("$@")
    java -jar "$jar" "${args[@]}" &
    echo $! > "./pids/${name}.pid"
}
# Sellers
for i in {1..5}; do
    start_component "SELLER-$i" seller.jar "SELLER-$i" "tcp://*:556$i"
"seller.properties"
done
# Marketplaces
for i in {1..2}; do
    start_component "MARKETPLACE-$i" marketplace.jar "MARKETPLACE-$i"
"marketplace.properties"
done
```

To start this script, run the following commands inside your terminal:

```
./start-system-new.sh
```

Starts all components in foreground (logs shown in the terminal) and tracks their PIDs.

```
./stop-system-new.sh
```

Stops previously started local components by reading stored PIDs and terminating them.

2.4 Running with Docker

Docker images and a compose file are provided. The repository provides helper scripts which first must be made executable.

```
chmod +x docker-run.sh
```

The provided `docker-run.sh` script offers commands for build, start, stop, restart, logs, and clean operations to manage the Docker environment. Logs for each component are stored under `logs/`. You can start the whole environment using:

```
./docker-run.sh start
```

The compose file defines one client, five sellers and two marketplace instances that share a dedicated Docker network. The configuration includes:

```
services:
  client-1:
    image: marketplace-client:latest
    container_name: client-1
    environment:
      - INSTANCE_ID=CLIENT-1
    ports:
      - "5555:5555"
      - "5556:5556"
    networks:
      - marketplace-network
    volumes:
      - ./logs:/app/logs
      - ./client-docker.properties:/app/client.properties:ro

  # Seller Services (5 instances, additional instances with similar
```

```
properties)
  seller-1:
    image: marketplace-seller:latest
    container_name: seller-1
    environment:
      - INSTANCE_ID=SELLER-1
      - ENDPOINT=tcp://*:5561
    ports:
      - "5561:5561"
    networks:
      - marketplace-network
    volumes:
      - ./logs:/app/logs
      - ./seller-docker.properties:/app/seller.properties:ro

  # Marketplace Services (2 instances, additional instances with similar
properties)
  marketplace-1:
    image: marketplace:latest
    container_name: marketplace-1
    environment:
      - SERVICE_TYPE=marketplace
      - INSTANCE_ID=MARKETPLACE-1
    depends_on:
      - seller-1
      - seller-2
      - seller-3
      - seller-4
      - seller-5
    networks:
      - marketplace-network
    volumes:
      - ./logs:/app/logs
      - ./marketplace-docker.properties:/app/marketplace.properties:ro

networks:
  marketplace-network:
    driver: bridge
```

Each component uses environment variables for configuration and mounts volumes for logs and properties files. The marketplace instances depend on all seller services to ensure proper startup order. All services connect via a shared bridge network named `marketplace-network`.

Additional seller instances can be added by editing `docker-compose.yml`. Containers can be restarted one at a time to perform rolling upgrades. After a crash, restart the affected container and it will rejoin via ZeroMQ. Each service could expose a simple `/health` HTTP endpoint to act as a liveness probe for container orchestration.

For a manual docker setup, the scripts below can be used as an alternative.

Create shared network:

```
docker network create marketplace-network
```

Run individual containers (example for SELLER-1):

```
docker run --rm --name seller-1 \
  -e INSTANCE_ID=SELLER-1 \
  -e ENDPOINT=tcp://*:5561 \
  -p 5561:5561 \
  --network marketplace-network \
  -v "$(pwd)/logs:/app/logs" \
  -v "$(pwd)/seller-docker.properties:/app/seller.properties:ro" \
  marketplace-seller:latest
```

Repeat for other sellers, marketplaces, and the client with their respective environment variables and properties mounts.

To explicitly stop individual docker containers, the following command can be used:

```
docker stop client-1 marketplace-1 marketplace-2 seller-1 seller-2 seller-3 seller-4 seller-5
```

3. Architecture

3.1 File Structure

The repository is organized so that each component resides in its own package. Important paths include:

Path	Description
`src/main/java/com/example/marketplace`	Marketplace coordinator (`Marketplace.java`, `OrderSaga.java`)
`src/main/java/com/example/seller`	Seller service handling inventory and failure simulation (`Seller.java`)
`src/main/java/com/example/client`	Load-generating client (`Client.java`)
`src/main/java/com/example/common`	Shared message and domain classes (`Message`, `Order`, etc.)
`src/main/java/com/example/config`	Configuration loaders for all components
`*.properties`	Runtime configuration files used by the services
`start-system-new.sh`, `stop-system-new.sh`, `docker-run.sh`	Scripts for starting locally or in Docker

3.2 Main Components

The core of the system consists of:

Component	Port Range	Responsibilities
Seller	5561-5565	Manage stock, simulate failures, respond to reserve/confirm/cancel
Marketplace	5555-5556 (orders) 6555-6556 (notifications)	Coordinate the SAGA workflow across sellers
Client	dynamic	Generate orders and display statistics

The Java classes under `src/main/java/com/example` implement these roles:

- `Seller.java` keeps an in-memory inventory and processes each request in its own virtual thread while simulating failures defined in `SellerConfig`.
- `Marketplace.java` coordinates the SAGA, tracking per-order state in `OrderSaga` and scheduling retries and timeouts.
- `Client.java` generates random orders and distributes them across marketplaces, gathering statistics from notifications.

Failure behaviour such as crash probability or network delay is configured via the `*.properties` files. For example the seller configuration contains probabilities for crashes and lost responses as shown below:

```
failure.crash.probability=0.05
failure.no.response.probability=0.05
failure.process.no.confirm.probability=0.05
```

These properties files use a simple key-value pair format to configure various aspects of the system, including simulated failure scenarios.

3.3 Communication Architecture

All services communicate over **ZeroMQ**. The marketplace uses `DEALER` sockets to talk to sellers and a `REP` socket to receive orders from clients. Notifications are delivered via `PUSH`/`PULL` sockets. Messages are simple JSON strings described by the `Message` class and enumerated in `MessageType`.

ZeroMQ provides different types of sockets for various communication patterns:

- DEALER: Used for asynchronous request/reply patterns
- REP: Used for synchronous request/reply patterns
- PUSH/PULL: Used for pipeline patterns, where data is pushed from one end and pulled from the other

```java
// simplified excerpt from MessageType
public enum MessageType {
    RESERVE_REQUEST, CONFIRM_REQUEST, CANCEL_REQUEST, ORDER_REQUEST,
    RESERVE_RESPONSE, CONFIRM_RESPONSE, CANCEL_RESPONSE,
    SAGA_COMPLETED, SAGA_FAILED, SAGA_TIMED_OUT
}
```

3.2.1 Message Schema

Type	Required Fields	Example
RESERVE_REQUEST	sagaId, productId, quantity	`{ "type":"RESERVE_REQUEST", "sagaId":"123", "productId":"PROD-1", "quantity":1 }`
RESERVE_RESPONSE	sagaId, success, errorMessage?	`{ "type":"RESERVE_RESPONSE", "sagaId":"123", "success":true }`
CONFIRM_REQUEST	sagaId, productId, quantity	similar to RESERVE_REQUEST
CONFIRM_RESPONSE	sagaId, success	`{ "type":"CONFIRM_RESPONSE", "sagaId":"123", "success":true }`
CANCEL_REQUEST	sagaId, productId, quantity	cancellation request

Type	Required Fields	Example
CANCEL_RESPONSE	sagaId, success	confirmation of cancel
SAGA_COMPLETED	sagaId, sagaStatus	`{ "type":"SAGA_COMPLETED", "sagaId":"123", "sagaStatus":"COMPLETED" }`

3.4 Configuration

Each component loads its configuration from a dedicated `.properties` file which is parsed by its `*Config` class:

- **MarketplaceConfig** (`marketplace.properties`) – defines the seller endpoints, base and per-seller SAGA timeouts (`saga.timeout.base.ms`, `saga.timeout.per.seller.ms`), retry limits (`max.retries`, `response.timeout.ms`), and optional network or failure simulation flags.
- **SellerConfig** (`seller.properties`) – specifies initial inventory and thresholds for each product, average and variance of processing time, probabilities for crash/no-response/processing failures, and restocking parameters such as `restocking.proactive.enabled` and `restocking.delay.ms`.
- **ClientConfig** (`client.properties`) – configures how often orders are generated (`order.interval.ms`), maximum items and quantities per order, target marketplaces, and client-side failure simulation.

Changing these values alters runtime behaviour without recompilation. For example, increasing `failure.crash.probability` in the seller configuration makes crashes more frequent, while reducing `order.interval.ms` increases client load.

The following key properties can be changed:

- `saga.timeout.base.ms` (30_000)
- `saga.timeout.per.seller.ms` (10_000)
- `max.retries` (3)
- `response.timeout.ms` (5_000)
- `failure.crash.probability` (0.05)
- `failure.no.response.probability` (0.05)
- `failure.process.no.confirm.probability` (0.05)
- `restocking.proactive.enabled` (true)
- `order.interval.ms` (2_000)
  These values balance realism with testability and can be tuned for experiments or production-like reliability.

3.5 Threading Model

Each service relies heavily on virtual threads introduced in modern Java. The `Marketplace` starts dedicated threads for handling client requests, seller responses and transaction timeouts. Sellers process every incoming request in its own virtual thread, allowing thousands of parallel operations without heavy OS threads. The client generates orders and handles responses in separate virtual threads as well. The following snippet shows how the marketplace launches the response handler:

```
responseHandlerThread = Thread.ofVirtual()
    .name("ResponseHandler-" + marketplaceId)
    .start(() -> { /* poll and dispatch seller responses */ });
```

Virtual threads are managed by the Java runtime rather than the operating system, allowing for much higher concurrency with less overhead.

4. Fault Tolerance

4.1 Error Scenarios

The code accounts for a variety of failures. The table summarizes how each situation is detected and handled.

Error Type	Detection Trigger	Recovery Strategy
Seller crash or no response	No `RESERVE_RESPONSE` within `response.timeout.ms`	Marketplace retries up to `max.retries`; if still missing, it rolls back all prior reservations
Seller processes but does not reply	Reservation applied but no response (simulated by `failure.no.response.probability`)	Retries are issued; if the saga times out the marketplace cancels the reservation to release stock
Insufficient inventory	Seller responds with `success=false` to `RESERVE_REQUEST`	Marketplace immediately cancels any successful reservations and marks the saga failed
Timeout during confirm phase	Missing `CONFIRM_RESPONSE`	Marketplace sends `CANCEL_REQUEST` to all sellers and retries confirmation until `max.retries` is reached
Network delay or dropped response	Response arrives after saga finished	`OrderSaga` ignores late messages, preventing inconsistent state

Error Type	Detection Trigger	Recovery Strategy
Client disconnects before completion	Client stops polling for notifications	Marketplace continues processing and stores the final status; the client will miss the notification but consistency is preserved
Marketplace crash	Marketplace process stops before saga completion	Another marketplace instance can take new orders; sagas on the failed node do not complete and may require manual cleanup

4.2 Fault Tolerance Techniques

The marketplace coordinates orders using the SAGA pattern. Every outbound request is tracked in `OrderSaga`, allowing the coordinator to determine which sellers need compensation. When a response does not arrive within `response.timeout.ms` the scheduler in `sendReserveRequestWithRetry` or `sendConfirmRequestWithRetry` issues another attempt. The delay between attempts grows exponentially to reduce pressure on unhealthy sellers and the network.

If retries are exhausted or a timeout occurs, `rollbackOrder` sends `CANCEL_REQUEST` messages to all sellers that reserved stock. Messages are idempotent so a retry does not duplicate work on the seller side. Late responses are ignored because `OrderSaga.canProcessResponse` checks that the saga is still active before applying changes.

A typical failure is a seller not confirming a reservation. The marketplace then rolls back as shown in `handleReserveResponse`:

```
if (response.isSuccess()) {
    // when all reserved proceed to confirmation
} else {
    // ANY failure triggers complete rollback
    rollbackOrder(saga);
}
```

4.5 Consistency & Compensation

Idempotent message handling ensures retries do not duplicate actions. `OrderSaga` tracks which seller has responded to avoid double processing. Rollbacks occur in reverse order of reservations to restore inventory accurately. Retry logic uses exponential backoff starting at 1s as seen in `sendReserveRequestWithRetry`. Saga timeouts are calculated from `saga.timeout.base.ms` plus `saga.timeout.per.seller.ms` so larger orders automatically allow more time.

5. Observability & Logging

The services log all events using Java util logging. Each log entry includes the order ID as a correlation identifier so that the entire saga can be reconstructed from logs. Log levels range from INFO for business events to FINE for detailed debugging messages.

Metrics such as successful order rate, average latency and number of retries are written to the logs for later analysis.

6. Results and Conclusion

6.1 Current Standpoint

The implementation demonstrates that the SAGA pattern combined with ZeroMQ and virtual threads can handle a variety of failure scenarios. All components start via scripts or Docker, and orders are either fully processed or correctly rolled back.

6.2 Key Learnings

Implementing custom JSON handling and manual retry logic proved to be the most complex part of the code. The interplay between asynchronous messaging and timeouts required careful state management which is encapsulated in the `OrderSaga` class.

JSON (JavaScript Object Notation) is a lightweight data interchange format that is easy for humans to read and write and easy for machines to parse and generate.

6.3 Future Improvements

Several enhancements could be made to further improve the system's robustness and usability. We plan to develop automated integration tests that would bring up containers and systematically assert outcomes across various failure scenarios, which would strengthen our confidence in the system's resilience. Additionally, extending the documentation with detailed sequence diagrams for the SAGA workflow would provide better clarity on transaction flows, especially for new team members trying to understand the system. Lastly, the configuration handling could be improved by implementing a web-based dashboard for real-time statistics and system monitoring, which would make it easier to observe the system's behavior during high-load situations and failure events.