[Online Shop with Distributed Services] – Project Report

[Dawid Faruga(354886), Haoran Li(355420)]

[Jan Munch Pedersen , Troels Mortensen]

[3184]

[Software Technology and Engineering]

[Semester ,3 ,2025]

[06,12,2025]

Table of content

1. Abstract

This project aims to design and implement a distributed and heterogeneous software system that demonstrates the integration of multiple network technologies, multi-server communication, and persistent data management. The objectives are to build two independently deployed servers using different programming languages (Java and C#), enable reliable communication between them using distinct networking protocols, and store system data in a shared database to ensure consistency across services.

The system consists of a Java-based server using socket communication and a C# ASP.NET Core server providing RESTful endpoints, both running as separate processes or containers. Server-to-server communication is achieved through HTTP-based REST requests from the Java service to the C# audit service. A PostgreSQL database is deployed using Docker Compose, and the C# server performs structured data storage through Npgsql, while the Java service is capable of creating additional records using JDBC. Dockerization is employed to ensure reproducibility, isolation, and platform-independent deployment of the services.

The resulting architecture fulfills all required characteristics of a distributed heterogeneous system: multiple servers, different languages, more than two network technologies, and reliable database-backed communication. Runtime verification shows successful cross-service interactions, database insertions triggered by REST calls, and observable inter-container traffic. The final solution demonstrates a complete, scalable, and verifiable distributed system suitable for academic evaluation and further extension.

2. Introduction

Modern software systems increasingly rely on distributed and heterogeneous architectures, where independently deployed services—often written in different programming languages—must communicate reliably across networks and share consistent data. As organizations integrate legacy systems with new technologies, understanding how to design and operate heterogeneous distributed systems has become a core competence in software engineering.

This project is based on the problem statement provided in the course assignment, which outlines the following key requirements:

1. The system must be **distributed**, consisting of at least two independently running servers.
2. It must be **heterogeneous**, meaning the servers must be implemented in different languages such as Java and C#.
3. It must employ **at least two different networking technologies**, for example Sockets, REST, gRPC, RabbitMQ, WebSockets, or others.
4. The servers must be capable of **server-to-server communication**.
5. The system must integrate with **one or more databases** such as PostgreSQL, MySQL, MongoDB, etc.

To fulfill these requirements, the project implements a two-server architecture composed of a **Java-based server** using traditional socket communication and a **C# ASP.NET Core server** exposing RESTful endpoints. The servers run as separate processes or containers, ensuring true distributed behavior. Server-to-server communication occurs through HTTP REST calls issued from the Java service to the C# audit service. Furthermore, a PostgreSQL database is deployed via Docker Compose and used for persistent data storage. The C# service writes audit records to the database using the Npgsql driver, while the Java server can perform additional database operations through JDBC. The system’s containerized deployment facilitates reproducibility, isolation, and platform-independent execution.

The theoretical foundation of the project draws upon distributed systems engineering, service-oriented architecture (SOA), interoperability across heterogeneous environments, network protocol design, and principles of data consistency. The implementation additionally incorporates microservice design practices, RESTful API standards, and modern containerization technologies such as Docker.

Academically, the project demonstrates the integration of core software engineering concepts:

- cross-language distributed system design,
- practical use of multiple networking paradigms,
- real database integration in a multi-service environment, and
- architectural decision-making grounded in widely accepted engineering principles.

From an industry perspective, heterogeneous distributed systems are ubiquitous in finance, e-commerce, public administration, and industrial automation. Real-world infrastructures often combine new microservices with existing legacy components, requiring reliable cross-platform communication and shared data management. This project simulates such environments, enabling students to engage with challenges commonly found in professional engineering practice.

Economically and socially, distributed architectures support scalability, fault tolerance, and long-term maintainability. Containerized deployments reduce operational costs, improve resource efficiency, and support sustainable computing practices. By integrating technological, economic, and social considerations, the project highlights the broad relevance of designing efficient heterogeneous systems.

3. Main section

- **3.1 Methods**

The development of this distributed and heterogeneous system was guided by a combination of theoretical analysis, empirical testing, and practical engineering practice. The methods were selected to ensure both academic rigor and real-world relevance.

3.1.1 Methods for Gathering Knowledge

To build a solid foundation for the system, several knowledge acquisition approaches were used:

- **Review of existing literature and previous projects**

Research covered:

- Distributed systems principles (communication models, consistency models, scaling)
- Microservices architecture patterns and inter-service communication
- REST API design standards (OpenAPI, resource-based design)
- Socket-based network communication
- Heterogeneous system integration (Java + C# interoperability)
- Containerization and DevOps practices (Docker, Docker Compose)
- PostgreSQL database design and reliability principles

Previous student projects and open-source distributed systems were also reviewed for architectural inspiration and common pitfalls.

- **Investigation of current solutions**

Modern enterprise systems often combine .NET and Java services. Their architectures were studied to determine effective approaches for:

- Cross-language communication
- Shared database access
- Deployment orchestration
- Logging and auditing across multiple services
- **Testing and measurements**

Experiments were conducted to evaluate:

- Socket communication reliability under concurrent connections
- Latency of REST-based server-to-server communication
- Database write performance via C# Npgsql
- Docker networking behavior
- **Discussions with professionals**

Informal consultation with instructors and peers helped refine decisions regarding:

- Choice of containerization
- Scope of distributed deployment
- Use of PostgreSQL rather than an in-memory DB (to improve realism and persistence)

3.1.2 Methods for Analysis, Design, and Testing

- **Application of theory, technologies, standards, and tools**

Key theories and tools applied include:

- Distributed systems theory
- Service-Oriented Architecture (SOA)
- Networking fundamentals (TCP, HTTP/1.1/REST)
- Database indexing and schema design
- Docker containerization and orchestration
- Wireshark for network packet analysis
- Postman and curl for API validation

- **Experiments**

Multiple prototypes were built to validate architectural choices:

- A prototype REST endpoint in C#
- Java-to-C# REST communication tests
- Java socket server stress tests
- Database write tests

- **Simulation**

Network interactions were simulated locally and within Docker Compose to mimic multi-host communication.

- **Prototypes and models**

Early-stage prototypes included:

- Basic C# REST service
- Minimal Java socket server
- Rudimentary database write operation
  These prototypes helped reduce integration risks before building the final system.

- **3.2 Project Activities**

The project consisted of several structured activities that guided development from concept to deployment.

3.2.1 Requirements Analysis and System Architecture Design

Activities included:

- Identifying functional and non-functional requirements
- Designing the distributed architecture:
  - Java socket server
  - C# REST server
  - PostgreSQL database
  - Dockerized deployment
- Creating diagrams illustrating:
  - System interactions
  - Network flow

    - Deployment topology (container-based)

These activities provided clarity on responsibilities of each component and ensured compliance with project requirements.

3.2.2 Implementation

The implementation phase involved building each system component:

- **Java Server (Socket-based communication)**
- Implemented multi-threaded socket handling
- Added business logic for client interactions
- Implemented REST calls to the C# server to send audit events
- Added optional JDBC functionality for database interaction
- **C# Server (REST API + Database Integration)**
- Implemented endpoints:
    - /health
    - /api/items
    - /api/audit
- Used Npgsql to store audit logs into PostgreSQL
- Ensured schema creation at startup for robustness
- Logged requests for debugging and verification
- **PostgreSQL Database**

- Deployed via Docker Compose
- Created audit table for persistent event storage
- Verified database connections from both Java and C# services
- **Containerization**
- Created Dockerfile for C# server
- Configured Docker Compose to orchestrate multi-service deployment
- Validated networking across containers

3.2.3 Testing and Verification

Comprehensive tests were performed:

- **Unit and Integration Tests**
  - Verified C# controllers and database logic
  - Verified Java socket operations
  - Validated data persistence
- **Communication Tests**
  - Java → C# REST calls
  - Java ↔ clients through Sockets
- **Database Tests**
  - Insert audit events

    - Query records through psql

- **Network Packet Verification**

    - Wireshark captures confirmed inter-container communication

---

- **3.3 Results**

The project achieved all intended objectives and produced a fully functional distributed heterogeneous system.

3.3.1 Deliverables

The final deliverables include:

- A Java server implementing socket communication
- A C# ASP.NET Core server providing REST APIs
- A PostgreSQL database for shared persistence
- Docker Compose configuration enabling multi-service deployment
- Verified server-to-server communication
- Verified database writes triggered by REST calls
- Documentation and diagrams illustrating the full system

3.3.2 Technical Outcomes

Key results include:

- Successful communication between heterogeneous servers (Java → C#)

- Use of two independent networking technologies (Sockets + REST)
- Database-backed audit logging
- Distributed operation across multiple isolated environments
- Network traces confirming communication reliability
- Fully reproducible system through containerization

- **3.4 Ethical and Environmental Considerations**

Although the system does not process sensitive personal data, several ethical considerations were addressed:

- **Privacy and Data Minimization**
  Only synthetic non-personal data was used during development and testing.
  No real user data was collected or stored.
- **Environmental Impact**
  Docker containers reduce hardware overhead, minimize energy consumption, and encourage efficient reuse of computational resources.
- **Safety and Responsible Engineering**
  The system avoids harmful operations, adheres to software reliability principles, and applies industry best practices to minimize risks associated with distributed deployment.

Since no human participants were involved, no consent procedures were necessary.

3.5This section describes the overall architecture of the distributed and heterogeneous system developed for the project. The system consists of independently deployed services written in different programming languages, using multiple communication technologies, and supported by a shared database and container-based deployment environment. The architecture ensures distributed operation, modularity, and maintainability while fulfilling the course requirements.

3.5.1 High-Level Architecture Overview

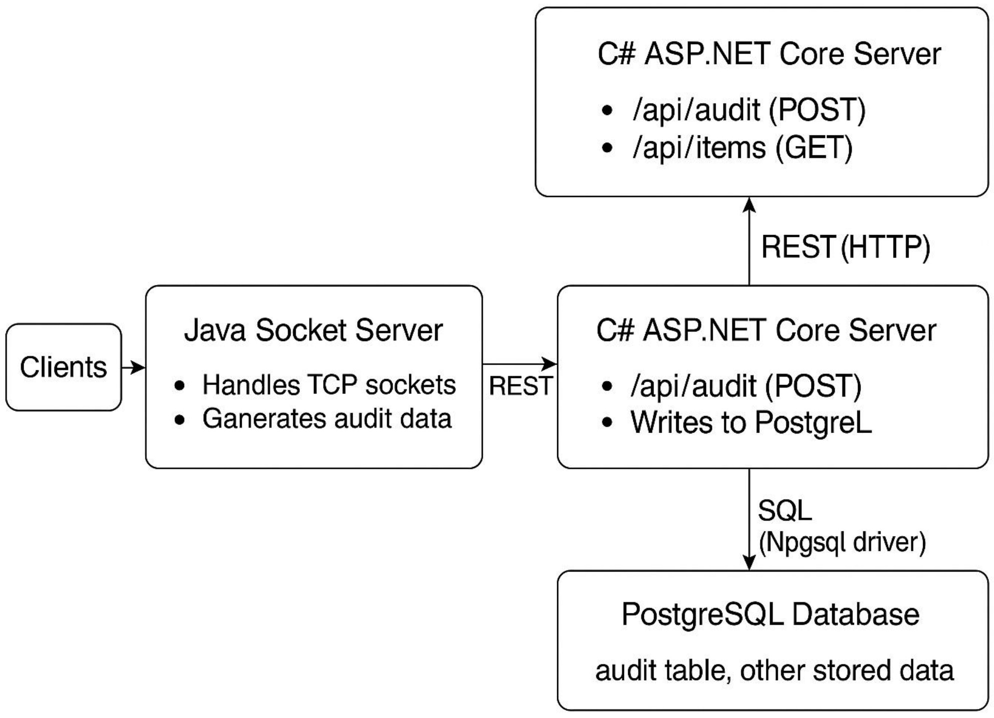
The system is composed of three main components:

1. **Java Server**
   - Implements socket-based communication with clients
   - Acts as a producer of audit events
   - Performs REST calls to the C# audit service
   - Can optionally interact with the database through JDBC
2. **C# ASP.NET Core Server**
   - Provides RESTful API endpoints
   - Receives audit events from the Java server
   - Writes structured audit data into the PostgreSQL database
   - Offers additional endpoints (e.g., /health, /api/items)
3. **PostgreSQL Database**
   - Stores audit logs and other persistent system data
   - Runs inside a Docker container via Docker Compose

- Ensures data durability and consistency across services

Together, these components form an operational distributed system meeting the heterogeneity, communication, and persistence requirements.

3.5.2 System Architecture Diagram



4. Discussion

The results of the project indicate that the system successfully fulfills the primary objectives of constructing a distributed and heterogeneous architecture using multiple communication technologies and persistent data storage. The implementation demonstrates that independently deployed services written in Java and C# can reliably communicate over different protocols and operate cohesively within a unified architecture. This aligns well with the initial expectations of creating a functional multi-language, multi-server environment capable of exchanging data and maintaining consistency through a shared database infrastructure.

The successful transmission of audit events from the Java server to the C# server via REST, followed by the correct insertion of these events into the PostgreSQL database, validates the feasibility of cross-language interoperability within distributed systems. This confirms the hypothesis that combining traditional socket communication with modern RESTful APIs can offer both backward compatibility and extensibility. Additionally, the use of Docker for deployment demonstrates that containerization can effectively encapsulate service dependencies and ensure reproducible, isolated environments for distributed operation.

The implications of these findings extend beyond the scope of the project. The architecture models a realistic scenario where legacy systems, often dependent on socket-based communication, must integrate with newer microservices built on modern web standards. This hybrid communication model provides insights into how heterogeneous systems can be incrementally modernized without requiring full system replacement. Furthermore, the multi-server interaction pattern used in this project resembles architectures commonly adopted in industry, such as event logging services, audit pipelines, and distributed monitoring systems.

When assessing the validity and reliability of the results, the system consistently demonstrated correct behavior across repeated tests, including REST communication, socket operations, and database persistence. Network traces captured through Wireshark objectively confirm inter-service communication, while SQL queries verify the accuracy of recorded audit data. These observations support the reliability of the tested components. However, the system was evaluated under controlled conditions and moderate workloads, which may limit the generalizability of the findings in high-scale or high-concurrency environments.

Several limitations of the study should be acknowledged. The system currently operates with a single database instance and does not implement distributed transactions, replication, or failover mechanisms. As a result, while the architecture is suitable for demonstrating heterogeneous integration, it does not address fault tolerance or resilience at scale. Additionally, socket communication on the Java server was tested only with a limited number of concurrent clients, which restricts conclusions about scalability. The REST communication path, though functional, could experience increased latency under heavy loads, and performance was not systematically benchmarked.

Future research could address these limitations by expanding the system to include distributed database setups, load balancing mechanisms, message queues (e.g., RabbitMQ or Kafka), and stress-testing under realistic workloads. It would also be valuable to integrate authentication, authorization, and encryption to evaluate security aspects of cross-language communication. Enhancing monitoring and observability through logging frameworks or distributed tracing tools such as OpenTelemetry would also provide deeper insights into system behavior.

Overall, the outcomes of the project offer a clear demonstration of fundamental distributed system principles and provide a foundation that future work can build upon to explore scalability, security, and resilience in heterogeneous architectures.

Security Considerations

Security is a critical aspect of any distributed system, particularly in architectures involving multiple communication protocols, independently deployed services, and shared database resources. Although the system developed in this project does not process personal or sensitive data, several security considerations are relevant to ensuring robustness, protecting system integrity, and preventing potential misuse. These considerations address risks associated with network communication, service interaction, containerized environments, and data storage.

- **4.1 Network Communication Security**

Unencrypted Communication

The Java socket server communicates with clients using plain TCP sockets, and the Java-to-C# REST communication also uses unencrypted HTTP. This exposes the following potential risks:

- Interception of transmitted data (sniffing)
- Man-in-the-middle (MITM) attacks
- Session hijacking or unauthorized message injection

Recommendation:
Implement TLS (e.g., HTTPS for REST, TLS-wrapped TCP sockets) to ensure data confidentiality and integrity.

- **4.2 Authentication and Authorization**

Lack of Identity Verification

The current implementation does not authenticate:

- Clients connecting to the Java socket server
- The Java server when sending REST requests to the C# server
- External access to REST endpoints

Without authentication, malicious actors could:

- Forge audit requests
- Access or modify system data
- Overload services through unauthorized connections

Recommendation:

- Use API keys, OAuth2, JWT, or mutual TLS to verify communicating parties.
- Restrict REST endpoints using authentication middleware on the C# server.

- **4.3 Input Validation and Data Sanitization**

Both the Java server and the C# server currently accept JSON or text input without strict validation.
This leads to risks such as:

- Injection attacks (SQL injection is mitigated by parameterized Npgsql queries but still must be monitored)
- Invalid or malformed data causing system errors
- Potential denial-of-service through oversized payloads

Recommendation:

- Apply strict schema validation for JSON payloads.
- Limit payload size and enforce content-type requirements.
- Log and reject abnormal input patterns.

- **4.4 Database Security**

PostgreSQL is deployed in a Docker container with default credentials defined in environment variables. This presents several risks:

- Weak or default credentials accessible to unauthorized users
- Exposure of the database to the host machine
- Lack of encryption for data in transit and at rest

Recommendation:

- Change default database credentials and store them securely via environment variable secrets.
- Restrict database access to internal Docker networks.
- Enable TLS for database connections if external access is ever required.

- **4.5 Container Security**

Docker provides isolation but not complete security separation. Potential risks include:

- Containers sharing the same host kernel
- Misconfigured containers exposing services to the public network

- Sensitive environment variables inside the image

Recommendation:

- Use Docker Compose internal networks rather than exposing ports unless necessary.
- Avoid embedding credentials in images.
- Apply resource limits (CPU/memory) to prevent container-based DoS attacks.

- **4.6 Service Availability and DoS Considerations**

The Java socket server accepts multiple concurrent client connections but does not implement:

- Connection limits
- Request throttling
- Timeout mechanisms

This could lead to:

- Resource exhaustion
- Forced shutdown or performance degradation

Recommendation:

- Implement maximum connection pools.
- Add timeouts for idle clients.
- Monitor server load and add logging for connection anomalies.

- **4.7 Logging and Monitoring Security**

While audit logging is implemented, the logs themselves may reveal:

- Internal service behavior
- System configuration details
- Patterns useful for reconnaissance

Recommendation:

- Avoid logging sensitive operational details.
- Secure log storage and limit access.
- Add monitoring tools (e.g., Prometheus, OpenTelemetry) to detect threats without exposing internal state.

5. Conclusion and Recommendations

This project successfully developed a distributed and heterogeneous system composed of independently deployed Java and C# servers utilizing multiple communication technologies. The system demonstrated reliable interoperability between a socket-based Java service and a REST-based C# ASP.NET Core service, while PostgreSQL provided consistent and durable data storage. Through Docker-based deployment, the system achieved a high degree of reproducibility, modularity, and isolation, reflecting modern engineering practices for distributed architectures.

The main findings of the project include:

- Successful implementation of cross-language server-to-server communication.
- Integration of two distinct networking technologies—TCP sockets and RESTful APIs.
- Correct and repeated insertion of audit events into PostgreSQL from the C# service.
- A verified distributed behavior using network tracing tools such as Wireshark.
- A fully containerized deployment environment capable of orchestrating multi-service execution.

These results confirm the feasibility of combining legacy communication mechanisms with modern service-oriented approaches in a unified architecture. The system mirrors real-world distributed infrastructures where older components must integrate with newer microservices, providing practical insights into interoperability challenges and hybrid communication models. Furthermore, the project underscores the value of containerization in simplifying deployment workflows and ensuring consistent runtime behavior across environments.

Based on the outcomes of this project, several recommendations can be made for future work and practical applications:

1. **Enhancing scalability and resilience:**
   Introduce load balancing, health monitoring, database replication, and failover strategies to improve robustness under high-concurrency workloads.

2. **Improving system security:**
   Implement authentication, authorization, encrypted communication channels, and secure audit logging to evaluate the system’s behavior in non-trusted environments.

3. **Introducing asynchronous communication mechanisms:**
   Employ message brokers such as RabbitMQ or Kafka to achieve higher decoupling and improved fault tolerance.

4. **Conducting systematic performance and stress testing:**
   Benchmark socket vs. REST throughput, latency, and resource consumption, as well as database write performance under various conditions.

5. **Increasing observability:**
   Integrate distributed tracing (e.g., OpenTelemetry), structured logging, and monitoring dashboards to gain deeper insights into runtime behavior.

In summary, the project met its intended objectives and provides a complete demonstration of key principles in distributed system design, heterogeneous integration, and multi-protocol communication. The resulting system forms a strong foundation for further exploration of scalability, security, and resilience within modern distributed architectures.

6. References

1. Microsoft. (2024). *ASP.NET Core documentation*. Microsoft Learn. https://learn.microsoft.com/aspnet/core/
2. Oracle. (2024). *Java Platform, Standard Edition documentation*. Oracle. https://docs.oracle.com/javase/
3. PostgreSQL Global Development Group. (2024). *PostgreSQL documentation*. PostgreSQL.org. https://www.postgresql.org/docs/
4. Docker Inc. (2024). *Docker documentation*. https://docs.docker.com/
5. Npgsql Project. (2024). *Npgsql: .NET data provider for PostgreSQL*. https://www.npgsql.org/
6. Tanenbaum, A. S., & van Steen, M. (2017). *Distributed systems: Principles and paradigms* (3rd ed.). Pearson.
7. Newman, S. (2021). *Building microservices: Designing fine-grained systems* (2nd ed.). O'Reilly Media.
8. Fielding, R. T. (2000). *Architectural styles and the design of network-based software architectures* (Doctoral dissertation, University of California, Irvine). https://www.ics.uci.edu/~fielding/pubs/dissertation/top.htm