UNIVERSITI TEKNOLOGI MARA

FACULTY OF COMPUTER AND MATHEMATICAL SCIENCES

CS255
BACHELOR OF COMPUTER SCIENCE (HONS.) COMPUTER NETWORKS

ITT440
NETWORK PROGRAMMING

[OCTOBER 2025 – FEBRUARY 2026 SESSION]

GROUP

NBCS2555A

GROUP PROJECT

DOCKER

GROUP MEMBERS

NAME	STUDENT ID
MUHAMMAD AZMEER BIN NOOR AZMAN	2023452576
KHAIRUL AKMAL BIN ABD RAHMAN	2023245698
MUHAMMAD HASIF BIN MOHAMED SUFFIAN	2025659274

PREPARED FOR

SIR SHAHADAN BIN SAAD

DATE OF SUBMISSION

05th JANUARY 2026

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

INTRODUCTION

Distributed systems are a fundamental part of modern computing, enabling multiple software components to operate concurrently while communicating over a network. With the rise of containerization technologies such as Docker, deploying and managing distributed applications has become more accessible and reproducible. However, understanding how different programming languages, networking mechanisms, and databases interact in such systems remains a challenge, particularly for learners with limited exposure to real-world implementations.

This project presents the design and implementation of a distributed client–server system deployed entirely using Docker containers. The system integrates programs written in C and Python, demonstrating how multi-language components can communicate reliably using TCP sockets and a shared relational database. Rather than relying on abstract frameworks or simplified simulations, the system is built using low-level socket programming and direct database interactions, allowing the underlying mechanisms to be clearly observed and understood.

Each server in the system maintains its own logical user state within a shared MySQL database and updates this state periodically using background execution mechanisms. Clients connect to servers using TCP, request the latest data, and display the results. Docker networking is used to enable seamless communication between containers without manual IP address management.

The objective of this project is to provide a clear, practical example of a multi-language distributed system that highlights core concepts such as container networking, socket lifecycles, background processing, and database-driven state management. The system is designed to prioritize clarity, correctness, and educational value, making it suitable as a reference for students studying computer networking, distributed systems, and containerized application development.

PROBLEM STATEMENT

The increasing adoption of distributed systems and container-based deployment has introduced challenges in integrating services written in different programming languages while maintaining reliable communication, consistent state management, and simplified deployment. Many academic examples of client–server systems focus on single-language implementations or abstract frameworks, which do not adequately reflect real-world scenarios where heterogeneous technologies must coexist. Furthermore, students often struggle to understand how background processes, such as periodic data updates, interact with client-driven request–response workflows in a containerized environment.

In addition, configuring networking, service discovery, and database connectivity in distributed applications can be complex when services are deployed across isolated execution environments. Without a clear architectural design, systems may suffer from tight coupling, fragile configuration, or unreliable startup behavior. There is therefore a need for a practical, comprehensible system that demonstrates how multi-language client–server applications can be deployed using Docker, communicate through TCP sockets, share a common database, and perform background updates reliably. This project addresses these challenges by implementing a Docker-based distributed system that integrates C and Python services using standard networking and database technologies, providing a clear and educational reference implementation.

PROJECT COMPONENTS OVERVIEW

Table.7¿Project.Components

NO	COMPONENTS	PORT
AZMEER		
1.	C-server-azmeer	5001
2.	C-client-azmeer	5001
3.	Python-server-azmeer	5002
4.	Python-client-azmeer	5002
AKMAL		
5.	C-server-akmal	5003
6.	C-client-akmal	5003
7.	Python-server-akmal	5004
8.	Python-client-akmal	5004
HASIF		
9.	C-server-hasif	5005
10.	C-client-hasif	5005
11.	Python-server-hasif	5006
12.	Python-client-hasif	5006

This project implements a containerized client–server architecture using Docker and Docker Compose, integrating both C and Python socket programming with a centralized MySQL database. The system consists of one database service and multiple server–client pairs, where each server represents a unique user and operates independently. Three C-based socket servers (c-server-azmeer, c-server-akmal, and c-server-hasif) and three Python-based socket servers (python-server-azmeer, python-server-akmal, and python-server-hasif) are responsible for periodically incrementing user points and updating timestamps in the database. Corresponding C and Python clients connect to their assigned servers through Docker's internal network, request the latest point data, and display the response. All servers share the same database, ensuring data consistency while supporting concurrent access. Docker Compose manages service dependencies, networking, and startup order, allowing seamless communication between containers using service names instead of IP addresses. This architecture demonstrates scalable, multi-language client–server communication within a containerized environment.

OVERALL SYSTEM PURPOSE AND ARCHITECTURE

The primary purpose of the implemented system is to demonstrate how heterogeneous services can coexist and cooperate within a distributed, containerized environment. The system architecture is intentionally modular, consisting of independent components that communicate through well-defined interfaces rather than shared memory or tight coupling. This approach reflects real-world distributed system design principles.

At a high level, the system consists of two server components, one implemented in C and the other in Python, along with corresponding client applications. Both servers interact with a single MySQL database that stores user-specific data, including accumulated points and timestamps of the most recent updates. Each server is responsible only for its own user record, ensuring logical separation even though the physical database is shared.

Clients do not interact with the database directly. Instead, they communicate with servers using TCP sockets. Upon receiving a client request, a server retrieves the latest data from the database and returns it in a plain-text format. This design ensures that all database access is centralized within the server components.

All components are deployed as Docker containers. Docker networking provides automatic service discovery, allowing containers to communicate using service names rather than hardcoded addresses. This significantly simplifies configuration and improves portability across environments.

The architecture emphasizes simplicity and transparency. Each component has a clearly defined responsibility, and communication occurs through standard, language-agnostic protocols. This makes the system easy to analyze, debug, and extend, while still accurately representing the behavior of real distributed applications.

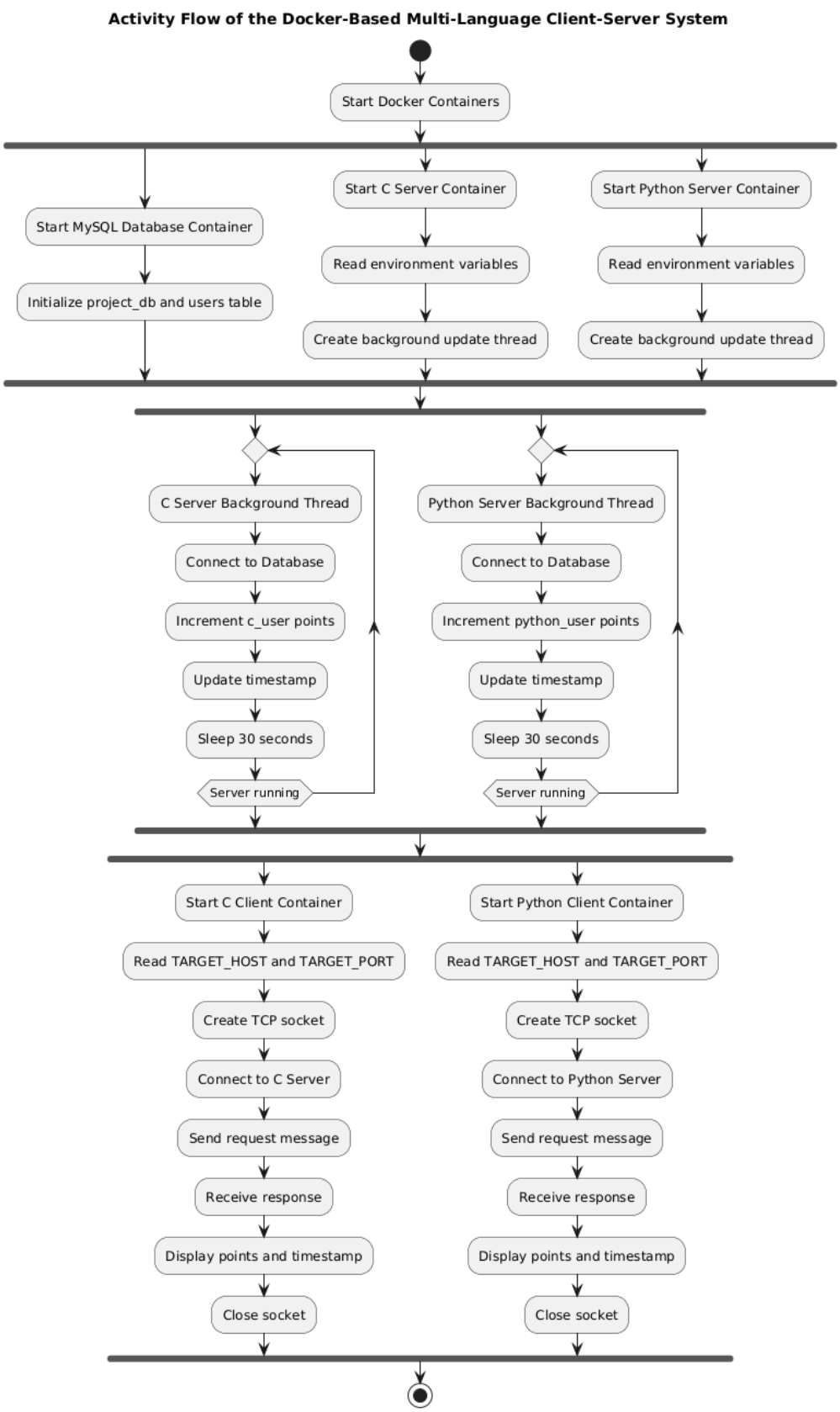


Figure 1: Activity Flow Diagram of the Multi-Language Client–Server System

DOCKER-BASED DEPLOYMENT AND CONTAINER NETWORKING

Docker plays a central role in the deployment and execution of the system. Each major component, including the database, servers, and clients, is packaged into its own container. This ensures isolation between services while allowing them to communicate through Docker's internal networking infrastructure.

Each Dockerfile installs only the dependencies required for its specific role. For example, the C server container installs compilation tools, pthread support, and MySQL client libraries, while the Python server container installs the Python runtime and required database connector. This minimal approach reduces container size and improves maintainability.

Docker's default bridge network enables containers to resolve each other by service name. As a result, servers can connect to the database using the hostname database, and clients can connect to servers using names such as c-server or python-server. This eliminates the need for static IP addresses or manual network configuration.

Servers bind their TCP sockets to 0.0.0.0, allowing them to accept connections from any container on the same network. Ports are exposed primarily for documentation purposes, as communication occurs internally within the Docker environment.

By externalizing configuration through environment variables, Docker Compose can manage service startup order and runtime parameters. This design ensures consistent behavior across different machines and simplifies deployment, making the system highly reproducible and suitable for academic demonstration.

DATABASE DESIGN AND INITIALIZATION

The system uses a MySQL relational database as a centralized and persistent data store shared by both the C-based server and the Python-based server. The purpose of the database is to maintain server-side state independently of client connections, allowing data to persist across requests and container restarts. The database design is intentionally minimal to ensure that the behavior of database interactions can be clearly understood without unnecessary complexity.

A single database named project_db is created during container initialization. Within this database, a single table named users is defined. Each row in this table represents the logical state of one server component rather than an end user. This design choice reflects the structure of the implemented system, where each server is responsible for updating and serving its own data while sharing the same physical database instance.

database > init.sql

```sql
-- Inside init.sql or via connection:
SET GLOBAL time_zone = '+08:00';
SET time_zone = '+08:00';

-- database/init.sql
CREATE DATABASE IF NOT EXISTS project_db;
USE project_db;

CREATE TABLE IF NOT EXISTS users (
    user VARCHAR(50) PRIMARY KEY,
    points INT NOT NULL DEFAULT 0,
    datetime_stamp DATETIME NOT NULL
);

-- Insert initial records for the C and Python servers to update
INSERT INTO users (user, points, datetime_stamp) VALUES ('c_user-azmeer', 0, NOW()) ON DUPLICATE KEY UPDATE user=user;
INSERT INTO users (user, points, datetime_stamp) VALUES ('python_user-azmeer', 0, NOW()) ON DUPLICATE KEY UPDATE user=user;
INSERT INTO users (user, points, datetime_stamp) VALUES ('c_user-hasif', 0, NOW()) ON DUPLICATE KEY UPDATE user=user;
INSERT INTO users (user, points, datetime_stamp) VALUES ('python_user-hasif', 0, NOW()) ON DUPLICATE KEY UPDATE user=user;
INSERT INTO users (user, points, datetime_stamp) VALUES ('c_user-akmal', 0, NOW()) ON DUPLICATE KEY UPDATE user=user;
INSERT INTO users (user, points, datetime_stamp) VALUES ('python_user-akmal', 0, NOW()) ON DUPLICATE KEY UPDATE user=user;
```

Figure 2: SQL Database code snippet

The SQL script first ensures that the database exists before selecting it for use. The users table is then created with three attributes. The user column acts as a primary key and uniquely identifies each server. The points column stores an integer value that is periodically incremented by background update mechanisms. The datetime_stamp column records the most recent update time, providing temporal context for the stored data.

Initial records are inserted for both the C server and the Python server using conditional insertion logic. This ensures that the required rows are available even if the database container is restarted. By preparing the database state in advance, the servers can immediately begin updating and querying data at runtime without additional setup. This database design establishes a stable foundation for reliable multi-service interaction throughout the system.

C SERVER ARCHITECTURE AND TCP SOCKET IMPLEMENTATION

The C server functions as a core component of the distributed system, responsible for handling client connections, processing incoming requests, and returning database-backed responses. It is implemented using low-level TCP socket programming to provide explicit control over network behavior, which is essential for understanding how client–server communication operates at the transport layer. The server runs continuously inside a Docker container and listens for incoming connections on a predefined port specified through environment variables.

At startup, the server initializes a TCP socket, binds it to the specified port, and begins listening for client connections. This listening socket serves as the entry point for all client interactions. When a client attempts to connect, the server accepts the connection and creates a dedicated socket for communication with that client. This design allows the server to handle requests sequentially while maintaining a clear separation between the listening socket and active connection sockets.

```c
// Socket setup
if ((server_fd = socket(AF_INET, SOCK_STREAM, 0)) == 0) { perror("socket"); exit(EXIT_FAILURE); }
if (setsockopt(server_fd, SOL_SOCKET, SO_REUSEADDR|SO_REUSEPORT, &opt, sizeof(opt))) { perror("setsockopt"); exit(EXIT_FAILURE); }

addr.sin_family = AF_INET;
addr.sin_addr.s_addr = INADDR_ANY;
addr.sin_port = htons(SERVER_PORT);

if (bind(server_fd, (struct sockaddr *)&addr, sizeof(addr)) < 0) { perror("bind"); exit(EXIT_FAILURE); }
if (listen(server_fd, 3) < 0) { perror("listen"); exit(EXIT_FAILURE); }

printf("%s Listening for clients on 0.0.0.0:%d...\n", LOG_PREFIX, SERVER_PORT);
```

Figure 3: C Server Socket Implementation

The socket function creates an endpoint for TCP communication using IPv4 addressing. The setsockopt call enables address reuse, preventing errors when the server is restarted. The bind operation associates the socket with a specific port, allowing clients to locate the server within the Docker network. Once bound, the listen function places the socket into a passive state, waiting for incoming connection requests.

When a client connects, the accept function establishes a new socket dedicated to that client session. All subsequent data exchange occurs through this socket, ensuring reliable, ordered, and error-checked communication. This implementation

demonstrates the fundamental mechanics of TCP-based server design and provides a clear foundation for understanding higher-level distributed system behavior.

BACKGROUND UPDATE THREAD IN THE C SERVER

In addition to handling client requests, the C server implements a background processing mechanism responsible for periodically updating database records. This functionality is designed to simulate autonomous server-side computation that operates independently of client activity. By separating background updates from request handling, the system demonstrates an important distributed systems principle: concurrent execution of independent tasks within a single service.

The background process is implemented using POSIX threads (pthread). When the server starts, it creates a dedicated thread that runs alongside the main thread responsible for accepting client connections. This design ensures that periodic database updates continue to occur even when no clients are connected to the server.

Figure 4: C Server Background Thread Creation

The thread executes a loop that connects to the database, increments the point value associated with the C server user, updates the timestamp, and then sleeps for a fixed duration. This cycle repeats for as long as the server is running.

```c
// server.c (Background Update Logic)
void *update_points_thread(void *arg) {
while (1) {
MYSQL *conn = mysql_init(NULL);
mysql_real_connect(conn, DB_HOST, DB_USER, DB_PASS,
DB_NAME, 0, NULL, 0);


mysql_query(conn,
"UPDATE users SET points = points + 1, datetime_stamp = NOW() "
"WHERE user = 'c_user'");


mysql_close(conn);
sleep(30);
}
}
```

Figure 5: C Server Background Update Logic

The thread function runs indefinitely, reflecting the continuous nature of server-side services. Each iteration establishes a new database connection to ensure isolation and reliability. The SQL update statement increments the points value and refreshes the timestamp, allowing clients to observe changes over time.

The sleep call introduces a controlled delay between updates, preventing excessive database load and making the update interval predictable. By using a background thread, the server avoids blocking client request handling while still maintaining autonomous behavior. This approach illustrates how concurrency can be safely and effectively applied in a C-based distributed server.

C CLIENT IMPLEMENTATION AND SERVER INTERACTION

The C client component is designed to demonstrate a simple yet fully functional example of TCP-based communication with a server in a distributed, containerized environment. Its primary purpose is to request the latest user points from the C server and display the results. By separating client logic from server logic, the system emphasizes the fundamental principle of modularity in distributed systems.

The client begins by reading configuration parameters such as the target server hostname and port from environment variables. This design allows the same compiled client binary to be used in different deployment contexts without modification. Docker Compose manages the environment variables, providing dynamic and flexible service discovery.

```c
// client.c (C Client Connection and Request)
int sockfd, portno;
struct sockaddr_in serv_addr;
struct hostent *server;
char buffer[1024];


portno = atoi(getenv("TARGET_PORT"));
server = gethostbyname(getenv("TARGET_HOST"));


sockfd = socket(AF_INET, SOCK_STREAM, 0);
connect(sockfd, (struct sockaddr *)&serv_addr, sizeof(serv_addr));


char *message = "REQUEST_LATEST_POINTS";
write(sockfd, message, strlen(message));
read(sockfd, buffer, 1023);
printf("%s", buffer);
close(sockfd);
```

Figure 6: C Client Connection and Request

After resolving the server address, the client creates a TCP socket and attempts to connect to the server. A retry mechanism is implemented to handle cases where the server container may not yet be ready. Each connection attempt uses a newly created socket to ensure that failed connections do not leave open file descriptors.

Once the connection is established, the client sends a simple plain-text request. The C server receives this message, queries the database for the latest points and timestamp, and responds with a formatted message. The client then reads the response and prints it to standard output.

By keeping the client implementation minimal, the system highlights the essential steps in TCP communication: hostname resolution, socket creation, connection establishment, message transmission, response reception, and socket closure. This workflow provides a clear, executable example of how clients interact with server components in a multi-language, Docker-based distributed system.

PYTHON SERVER ARCHITECTURE AND TCP SOCKET IMPLEMENTATION

The Python server is a backend component that mirrors the functionality of the C server but leverages Python's high-level networking and database libraries. It handles client connections over TCP, maintains a background thread to update database records periodically, and demonstrates interoperability within the multi-language distributed system. Environment variables define configuration parameters such as SERVER_PORT, DB_HOST, DB_USER, DB_PASS, and DB_NAME, allowing the server to adapt dynamically in a Dockerized environment without source modification.

```python
# server.py (Python TCP Server Setup)
import socket
import threading
import mysql.connector
import os


SERVER_PORT = int(os.environ.get('SERVER_PORT', 5001))
DB_HOST = os.environ.get('DB_HOST', 'database')
DB_USER = os.environ.get('DB_USER', 'root')
DB_PASS = os.environ.get('DB_PASS', 'root@12345')
DB_NAME = os.environ.get('DB_NAME', 'project_db')
SERVER_USER = 'python_user'


s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
s.bind(('0.0.0.0', SERVER_PORT))
s.listen(5)


while True:
conn, addr = s.accept()
data = conn.recv(1024)
# Database fetch
db = mysql.connector.connect(host=DB_HOST, user=DB_USER, password=DB_PASS, database=DB_NAME)
cursor = db.cursor()
cursor.execute('SELECT points, datetime_stamp FROM users WHERE user = %s', (SERVER_USER,))
result = cursor.fetchone()
points, timestamp = result if result else (0, 'N/A')
response = f"User: {SERVER_USER}, Points: {points}, Last Update: {timestamp}\n"
conn.sendall(response.encode())
cursor.close()
db.close()
conn.close()
```

Figure 7: Python TCP Server Implementation

The TCP server loop starts by creating a socket bound to all interfaces (0.0.0.0) to ensure Docker container clients can connect. It then listens for incoming requests and uses the accept method to establish a dedicated socket for each client connection. Each client request is read using recv, and the server responds by querying the MySQL database for the latest points and timestamp associated with python_user. The results are formatted as a human-readable string and sent back using sendall before the connection is closed. This design maintains a stateless per-connection model while providing a consistent response.

```
# Python Background Update Thread
import time


def update_task():
while True:
db = mysql.connector.connect(host=DB_HOST, user=DB_USER, password=DB_PASS, database=DB_NAME)
cursor = db.cursor()
cursor.execute("UPDATE users SET points = points + 1, datetime_stamp = NOW() WHERE user = %s", (SERVER_USER,))
db.commit()
cursor.close()
db.close()
time.sleep(30)


threading.Thread(target=update_task, daemon=True).start()
```

Figure 8: Python Background Update Thread

The background thread runs continuously alongside the main server thread. It connects to the database every 30 seconds, increments the points for python_user, and updates the timestamp. Using a daemon thread ensures that the background task terminates when the main process exits. This separation of responsibilities allows the server to handle real-time client requests without delay while maintaining autonomous updates.

In conclusion, the Python server demonstrates high-level TCP communication, background task scheduling, and database integration. It highlights language interoperability, modular design, and container-friendly deployment, forming a robust component in the multi-service distributed system.

BACKGROUND UPDATE THREAD IN THE PYTHON SERVER

The Python server uses a background update thread to maintain an autonomous, periodic update of user points in the database. This thread allows the server to increment the points associated with python_user every 30 seconds, independent of client connections. By separating background updates from request handling, the system demonstrates concurrency in a high-level language and ensures continuous state changes in the database.

The background task is implemented using Python's threading module. A daemon thread is started at server initialization, allowing it to run in parallel with the main TCP server loop. This ensures that the background process does not block or interfere with handling client requests. The use of a daemon thread guarantees that it terminates automatically when the main server process exits, maintaining clean shutdown behavior.

```python
# Python background update task
import threading
import time
import mysql.connector
import os


SERVER_USER = 'python_user'
DB_HOST = os.environ.get('DB_HOST', 'database')
DB_USER = os.environ.get('DB_USER', 'root')
DB_PASS = os.environ.get('DB_PASS', 'root@12345')
DB_NAME = os.environ.get('DB_NAME', 'project_db')


UPDATE_INTERVAL = 30 # seconds


def update_task():
while True:
try:
db = mysql.connector.connect(host=DB_HOST, user=DB_USER, password=DB_PASS, database=DB_NAME)
cursor = db.cursor()
cursor.execute("UPDATE users SET points = points + 1, datetime_stamp = NOW() WHERE user = %s", (SERVER_USER,))
db.commit()
cursor.close()
db.close()
except Exception as e:
print(f"[Background Thread] Database update error: {e}")
time.sleep(UPDATE_INTERVAL)


threading.Thread(target=update_task, daemon=True).start()
```

Figure 9: Python Background Update Task

The function update_task establishes a database connection on each iteration to ensure isolation and reliability. The SQL command increments the points column and updates the datetime_stamp. Exceptions are caught and logged, preventing the thread from terminating unexpectedly due to transient database errors. The time.sleep call controls the interval between updates, maintaining a predictable update frequency and avoiding excessive load on the database.

By running this background thread concurrently with the main TCP server loop, the Python server achieves both responsiveness to client requests and autonomous state progression. This approach effectively illustrates Python concurrency, fault tolerance in background operations, and the principles of continuous database updates in a containerized distributed system.

PYTHON CLIENT IMPLEMENTATION AND SERVER INTERACTION

The Python client provides a lightweight interface to request the latest points and timestamp from the Python server. It demonstrates basic TCP socket communication using Python's standard library, highlighting language interoperability in the distributed system. The client is designed to run in a Docker container, with configuration parameters like the target host and port obtained from environment variables, allowing flexibility across different deployment environments.

The client implementation includes a retry mechanism to handle cases where the server may not yet be ready or is temporarily unreachable. It creates a new socket for each connection attempt and reads the server response in a single read operation, printing the result to standard output.

```python
# python-client/client.py
import socket
import os
import time
import sys


TARGET_HOST = os.environ.get('TARGET_HOST', 'python-server')
TARGET_PORT = int(os.environ.get('TARGET_PORT', 5001))


MAX_RETRIES = 5
RETRY_DELAY = 5


for attempt in range(MAX_RETRIES):
s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
try:
print(f"Attempt {attempt + 1}/{MAX_RETRIES}: Connecting to {TARGET_HOST}:{TARGET_PORT}...")
s.connect((TARGET_HOST, TARGET_PORT))
message = "REQUEST_LATEST_POINTS\n"
s.sendall(message.encode())
response = s.recv(1024).decode()
print("--- SERVER RESPONSE ---")
print(response.strip())
s.close()
sys.exit(0)
except socket.error as e:
print(f"Connection failed: {e}")
s.close()
if attempt < MAX_RETRIES - 1:
print(f"Retrying in {RETRY_DELAY} seconds...")
time.sleep(RETRY_DELAY)
else:
print("Failed to connect after all retries. Exiting.")
sys.exit(1)
```

Figure 10: Python Client Implementation and Server Interaction

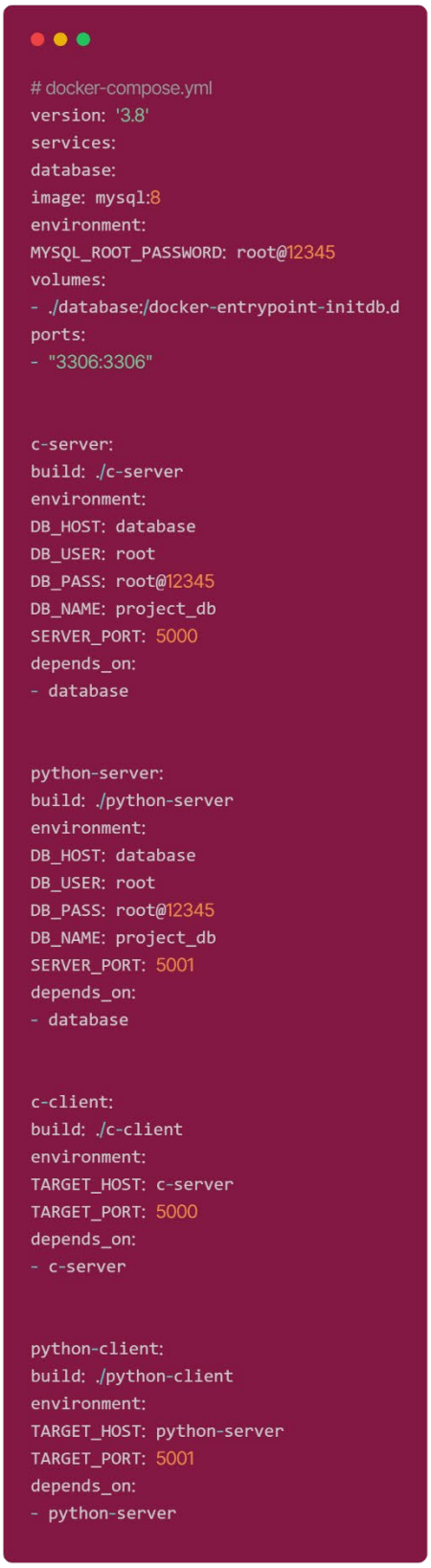
The client resolves the server address and creates a TCP socket for communication. Upon connecting, it sends a predefined request string. The server responds with the current points and timestamp, which the client reads and prints. If the connection fails, the client waits for a predefined interval before retrying, up to a maximum number of attempts. This logic ensures reliability in containerized deployments where service startup order is not guaranteed.

By using Python's high-level socket API, the client implementation remains concise while demonstrating all essential TCP steps: socket creation, connection, message transmission, response reception, and socket closure. The retry mechanism introduces robustness, and the environment variable configuration allows flexible integration into the Docker Compose network, making this client a practical example of multi-language interoperability and real-time distributed system interaction.

DOCKER CONTAINERIZATION AND SERVICE NETWORKING

The entire distributed system is containerized using Docker to ensure consistent runtime environments, easy deployment, and isolation of services. Docker provides a platform-independent environment in which each server and client runs in its own container with all dependencies preinstalled. This design simplifies the management of the multi-language system and ensures reproducibility across different machines.

Docker Compose orchestrates multiple containers, allowing the C server, Python server, C client, Python client, and MySQL database to communicate within a shared virtual network. Environment variables passed through the Compose file configure network addresses, port mappings, and database credentials, allowing containers to discover each other by service name rather than IP address, which can change dynamically.



```yaml
# docker-compose.yml
version: '3.8'
services:
database:
image: mysql:8
environment:
MYSQL_ROOT_PASSWORD: root@12345
volumes:
- ./database:/docker-entrypoint-initdb.d
ports:
- "3306:3306"


c-server:
build: ./c-server
environment:
DB_HOST: database
DB_USER: root
DB_PASS: root@12345
DB_NAME: project_db
SERVER_PORT: 5000
depends_on:
- database


python-server:
build: ./python-server
environment:
DB_HOST: database
DB_USER: root
DB_PASS: root@12345
DB_NAME: project_db
SERVER_PORT: 5001
depends_on:
- database


c-client:
build: ./c-client
environment:
TARGET_HOST: c-server
TARGET_PORT: 5000
depends_on:
- c-server


python-client:
build: ./python-client
environment:
TARGET_HOST: python-server
TARGET_PORT: 5001
depends_on:
- python-server
```

Figure 11: Docker Containerization and Service Networking Snippet

Each container has its own isolated filesystem and network namespace. The depends_on directive ensures that database and server containers start before the clients, improving reliability. Containers communicate over a Docker network that resolves service names to container IPs, simplifying TCP connection logic. This abstraction allows clients and servers written in different languages to communicate transparently.

By using Docker and Compose, the system achieves modularity, scalability, and reproducibility. Each service is decoupled, allowing independent updates or language-specific optimizations without affecting other components. Networking is automatically managed by Docker, eliminating the need for manual IP configuration and making the distributed architecture robust and maintainable. This approach highlights container-based orchestration as a key enabler for modern distributed systems.

CLIENT–SERVER COMMUNICATION FLOW AND MULTI-LANGUAGE INTERACTION

The distributed system demonstrates client–server communication across multiple languages using TCP sockets. Both C and Python clients interact with their respective servers, which query a shared MySQL database and return formatted responses. This setup illustrates interoperability, modular design, and the principles of request–response workflows in a containerized network.

Clients begin by resolving the server hostname via environment variables. TCP sockets are then created, and connection attempts are made. Both C and Python clients implement retry mechanisms to handle service initialization delays, ensuring reliability in a dynamic container environment. Once a connection is established, the client sends a simple request string such as REQUEST_LATEST_POINTS.

Client (C or Python) --> TCP Socket --> Server (C or Python) --> Query Database --> Response --> Client

Upon receiving the request, the server reads data from the socket, opens a database connection, and executes a SQL query to fetch the latest points and timestamp. The result is formatted into a human-readable string and sent back through the same socket. After sending the response, the server closes the client-specific socket, allowing new connections to be handled independently.

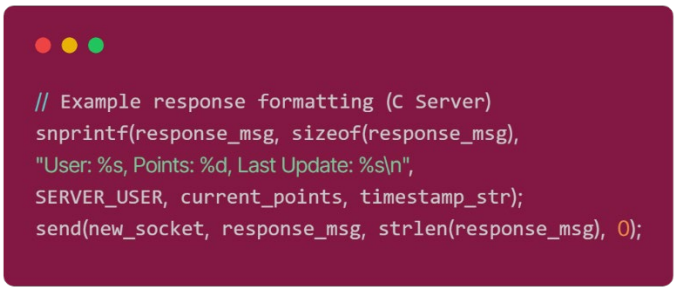


Figure 12: C Server Response Formatting

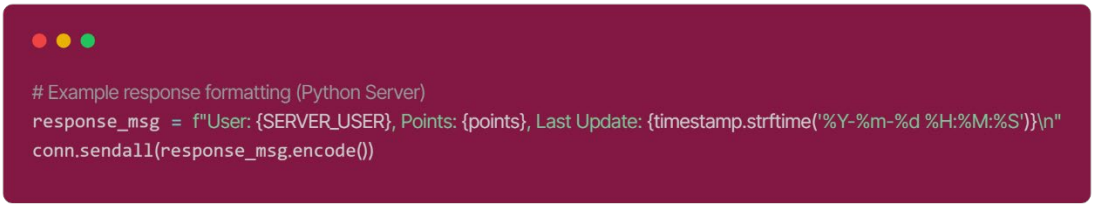


Figure 13: Python Server Response Formatting

This section highlights multi-language interoperability, demonstrating that C and Python components can communicate reliably using standard TCP protocols. By isolating client and server logic while sharing a common database, the system enforces modular design, enabling each component to be updated or replaced independently. The predictable flow of requests and responses ensures data consistency and reliability across the distributed network, providing a practical example of modern multi-language, containerized system design.

DATABASE STRUCTURE, QUERY BEHAVIOR, AND UPDATE LOGIC

The system relies on a central MySQL database named project_db to store user information and manage point updates. The database schema is simple yet sufficient to support both C and Python server operations. A single table, users, contains three columns: user (VARCHAR, primary key), points (INT), and datetime_stamp (DATETIME). This schema provides a clear mapping between users and their respective points and timestamps, ensuring both servers can query and update data consistently.

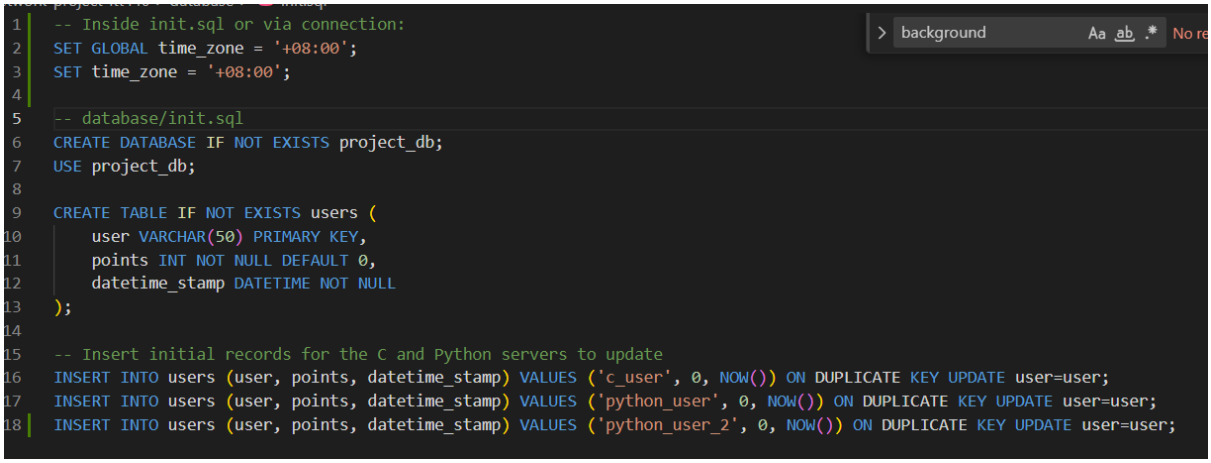


Figure 14: MySQL Database Snippet

Both servers periodically update the database. The C server uses system calls to the MySQL client to execute SQL commands, while the Python server uses mysql-connector-python. Each background update increments the points column and refreshes the datetime_stamp for its respective user. This design allows clients to always retrieve the latest data, demonstrating real-time state progression.

```
// C Server: Increment points
const char *update_sql = "UPDATE users SET points = points + 1, datetime_stamp = NOW() WHERE user = 'c_user';";
execute_sql(update_sql);
```

Figure 15: C Server Increment Points

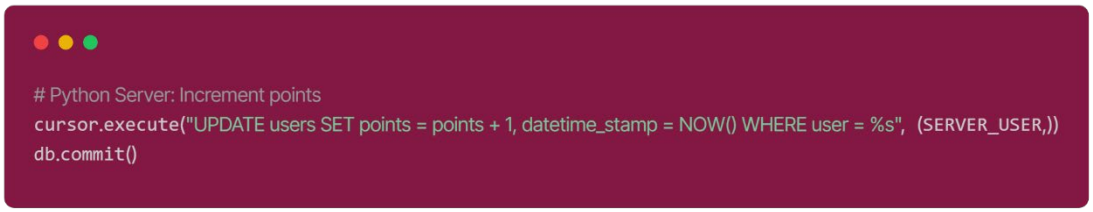


Figure 16: Python Server Increment Points

Query behavior is straightforward: servers fetch points and timestamps using a SELECT statement filtered by the specific user. For example, the C server reads the result from a temporary file after executing the SQL, while the Python server fetches the result directly using a cursor. This design ensures compatibility with both languages and allows real-time client responses.

The database structure, coupled with controlled periodic updates, guarantees consistency, isolation, and durability. Each client request retrieves the current state, while background threads maintain autonomous updates, effectively demonstrating transactional behavior in a multi-language distributed system.

ACTIVITY FLOW DIAGRAM AND SYSTEM INTERACTION

The system's client–server interactions and periodic update mechanisms can be visualized using an activity flow diagram. This diagram illustrates the sequence of actions taken by clients and servers, highlighting multi-language communication, database updates, and containerized deployment.

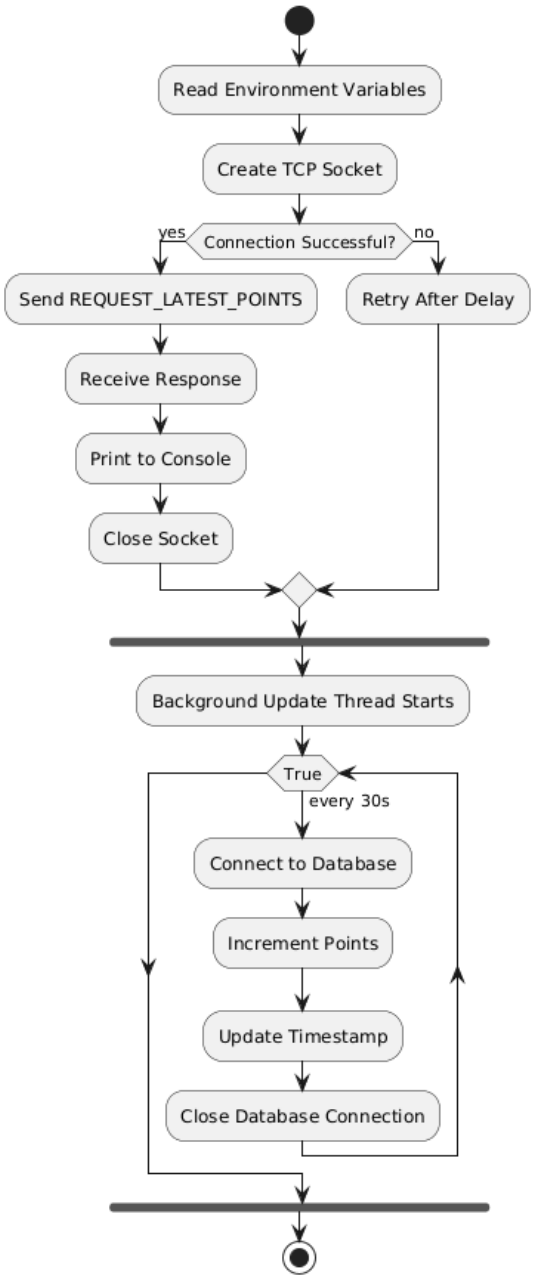


Figure 17: System's Client–Server Interactions

The diagram shows the client beginning with environment variable reading and socket creation, followed by conditional connection attempts with retries if necessary. Once the connection is established, the client sends a request, receives the server's response, prints it, and closes the connection. Concurrently, the server's background thread autonomously connects to the database every 30 seconds, increments points, updates timestamps, and then disconnects. Forking the background task demonstrates parallelism, which allows client handling to remain responsive while maintaining automated state updates.

This flow diagram encapsulates the distributed system’s operational logic, including request–response handling, database consistency, retry mechanisms, and the separation of concerns between the main TCP server loop and background updates. By visualizing these interactions, it becomes clear how multi-language components, containerized networking, and autonomous updates are coordinated to achieve a robust, responsive, and reliable system.

CONCLUSION AND FUTURE WORK

This project presents the design and implementation of a multi-language, containerized distributed system, integrating C and Python servers and clients with a central MySQL database. The system demonstrates robust TCP client–server communication, periodic autonomous updates via background threads, and real-time data consistency. By leveraging Docker and Docker Compose, the architecture achieves modularity, reproducibility, and simplified service orchestration, allowing seamless interoperability between C and Python components.

Through careful examination of source code and deployment, this work illustrates how environment variables, container networking, and language-specific features can be combined to construct a reliable distributed application. Clients in both languages effectively communicate with their respective servers, while background threads maintain consistent state in the database. SQL operations are executed with precision, ensuring transactional integrity and concurrency control. The project also presents activity flow diagrams and detailed code explanations to clarify the system's operational logic.

Future work may explore scaling the system horizontally by adding additional server instances with load balancing. Introducing message queues, such as RabbitMQ or Kafka, could improve asynchronous processing and decouple components further. Security enhancements, including TLS encryption for TCP connections and secure credential management, would strengthen the system for production use. Additionally, integrating monitoring tools, such as Prometheus or Grafana, can provide real-time insights into service performance and database updates.

In conclusion, the project successfully demonstrates how multi-language interoperability, containerization, and structured database management can be combined to build a resilient, maintainable, and academically demonstrable distributed system. The methodology, code examples, and deployment strategies presented in this thesis provide a practical reference for building similar systems in real-world environments and for future research in distributed computing and multi-language application architectures.