UEEN3123/UEEN3433 TCP/IP Network Application Development

ASSIGNMENT

June 2025 Trimester

Course Outcomes

In this assessment, you are assessed based on the following **course outcomes**:

- **CO2**: Build client-server network applications using TCP and UDP transport-layer protocols.
- **CO3**: Develop multicast and broadcast network applications.
- **CO4**: Utilize a scripting language to automate networking tasks.

Assessment Contribution

This assignment contributes **20%** to the overall assessment for this course with equal contribution for each CO.

Instruction

1. This is a **group project**. The minimum group size is 2 and the maximum group size is 3. Members of the group can be from any program (SE, ET) and year of study (2, 3 or 4).
2. When completed, archive all Python source files in ZIP file format. Each filename has 4 parts, group number, Programme (SE/ET), Course Code, Group Assignment (GA) and then .zip/rar. For example, if group number is **G9**, your **ZIP** archive shall be named **G9_SE_XXX_GA.zip**.
3. Upload the abovementioned ZIP archive using the link provided (in EWBLE).
4. Please ensure that you are logged into your UTAR student portal on your browser before attempting to access this link to s pecify your assignment group:
   - https://tinyurl.com/bddh7snm
5. To monitor all projects effectively, each group must meet with the lecturer at least three times during the project development process.

Deadline

The deadline for this assignment is **Tuesday, September 09, 2025**.

Learning Objectives

1. **Real-time Communication**: Understand and implement bidirectional, event-driven data streams using WebSockets or MQTT in a Flask application
2. **Database Integration**: Load CSV data into SQLite and perform CRUD operations within Flask request contexts.
3. **Application Architecture**: Design a modular Flask project structure that cleanly separates data generation, API endpoints, real-time messaging, and frontend rendering.
4. **Kiosk Interface Simulation**: Build a simple HTML/JavaScript client that visualizes station maps, active train positions, and dynamic route suggestions.

Prerequisites & Tools

- **Flask** (v2.x)
- **Flask-SocketIO** or **Flask-Sock** for WebSockets
- **Flask-MQTT** extension for MQTT integration
- **SQLite** (via sqlite3 or Flask's pattern)
- **Python 3.8+** and **pip**
- **CSV file** containing Malaysian station metadata and fare tables
- Basic **HTML/JavaScript** for the kiosk frontend

Summary

In this group assignment, students will build a real-time metro tracking and routing application using Flask, SQLite, and either WebSockets or MQTT for instant data updates. The backend will serve live position data generated by a Python script and stored in an SQLite database populated from provided CSV files of Malaysian station info and route fares. The frontend (simulated kiosk interface) will subscribe to real-time updates via WebSockets (e.g., Flask-SocketIO or Flask-Sock) or MQTT (using the Flask-MQTT extension) to display train movements and calculate optimal passenger routes.

The system architecture is designed to be modular and extensible: a Python-based data generator produces simulated train location updates every few seconds, which are persisted in the SQLite database and additionally broadcast via the chosen real-time protocol. The Flask application exposes RESTful HTTP endpoints for station metadata, fare lookup, and route computation, while also managing a persistent messaging layer—either WebSocket or MQTT—to transmit live updates to connected clients in the kiosk UI. Using SQLite ensures simple setup and portability; the CSV-based ingestion populates station and fare tables on startup, making deployment and testing straightforward.

Educationally, this assignment offers hands-on exposure to core concepts in real-time web systems, network-based messaging, frontend-backend interaction, and algorithmic routing. Students will learn to implement **WebSocket** via Flask-SocketIO or **MQTT** via Flask-MQTT (a lightweight Paho-MQTT wrapper)—both widely used in IoT and web applications. The project also reinforces database schema design and CSV ingestion, graph pathfinding (e.g. BFS or Dijkstra) for route selection, and building responsive visual interfaces that animate train movement in sync with backend events. These skills are highly transferable to real-world transportation, delivery, and tracking systems.

Each group is required to choose a project title from the provided list accessible through the following link: http://tinyurl.com/mu7xa6zh

Additionally, groups have the option to propose their preferred title. Upon approval from the lecturer, they may proceed with their chosen project. It is important to note that each group must select a unique title.

Assignment Requirements

1. Project Setup

- Initialize a repository and organize the Flask project into modules:
  - app.py (entry point)
  - database.py (SQLite connection and schema)
  - data_generator.py (script publishing live position updates)
  - realtime.py (WebSocket/MQTT setup)
  - routes.py (HTTP API endpoints)
  - static/ & templates/ (frontend files)

2. Data Ingestion

- Read station definitions (station_id, name, latitude, longitude) and fare matrix (origin_id, destination_id, price) from two CSV files.

  https://www.kaggle.com/datasets/niknmarjan/myrapidkl-train-dataset

- On application start, load these into an SQLite database with appropriate tables and indexes.

3. Real-Time Data Generation

- Implement data_generator.py that simulates train movements by updating each train's station_id or GPS coordinates every 2–5 seconds.
- The script should publish these updates via:
  - **WebSockets**: Emit events with socketio.emit('train_update', data)
  - or **MQTT**: mqtt.publish('metro/train_update', payload=json.dumps(data))

4. Real-Time Messaging Integration

- **WebSocket Option**:
  - Install and configure Flask-SocketIO or Flask-Sock in realtime.py.
  - Handle client connections and broadcast train updates.
- **MQTT Option**:
  - Configure Flask-MQTT with broker URL and topic subscriptions.
  - In the Flask app context, subscribe to metro/train_update and forward messages to connected HTTP/WebSocket clients if needed.

5. HTTP API Endpoints

- **GET** /stations: Return all station metadata.

- **GET** /fare?from=<id>&to=<id>: Return fare between two stations.
- **GET** /route?from=<id>&to=<id>: Compute and return the shortest path (by hops or price) using in-memory graph traversal.

6. Frontend (Kiosk) Interface

- Create a single-page application under templates/index.html that:
  - Shows a map (e.g., Leaflet.js) with station markers.
  - Subscribes to real-time updates via WebSocket or MQTT over WebSocket bridge.
  - Animates train markers moving between stations.
  - Provides a route-planning widget: select origin & destination, call /route, display path and fare.

Deliverables

1. **Source Code**: Complete Flask application in a public GitHub repo.
2. **README**:
   - Setup instructions (pip install -r requirements.txt, SQLite initialization).
   - How to run the data generator and Flask server.
   - How to access the kiosk interface.
3. **ER Diagram** and **System Architecture Diagram** (e.g., as PNG/PDF).
4. **Demo Video** (max 5 minutes) showing:
   - Real-time train movement.
   - Route selection and fare display.
5. **Report** (PDF, max 15 pages): Challenges faced, decisions (WebSocket vs. MQTT), and performance observations.

Timeline & Milestones

Week	Tasks
1	Project planning, repo setup, CSV ingestion, database schema design.
2	Implement data generator and real-time messaging (choose WebSocket or MQTT).
3	Build HTTP API endpoints and basic frontend map.

Week	Tasks
4	Integrate real-time updates into frontend; implement route planner.
5	Testing, documentation, diagrams, and demo video creation.

Evaluation Criteria

- **Correctness** (30%): All features work per specification (real-time updates, routing, fares).
- **Code Quality** (20%): Clear structure, comments, PEP8 compliance.
- **Architecture & Design** (15%): Modular code, well-defined interfaces, diagrams.
- **UX & Visualization** (15%): Smooth animations, intuitive route selection, responsive layout.
- **Documentation & Presentation** (20%): Clarity of README, report insight, and demo quality.