Toll Way

NSAPDEV Final Project

Members:

Mata, Maria Sarah Althea A.

Ranada, Arianne M.

**Table of Contents**

[**1. Project Description 2**](#_18bzdq95q82z)

[**2. Project Capability and Scope 3**](#_c1n7owuel7bc)

[2.1. Capabilities 3](#_1g8boyc9wllo)

[2.2. Scope 3](#_a71e35ormp4h)

[**3. System Architecture 3**](#_sftx2zj84oig)

[3.1. Description of the overall logical (network, hardware and software) architecture 3](#_hej4j12vex7n)

[3.2. Description of the server software architecture (include modules, library, probable classes or functions) 4](#_r4ffrp584bdr)

[3.3. Description of the client hardware and software architecture (include sensors, library, probable classes or functions 7](#_muk7lp3n6c52)

[**4. Project Performance 8**](#_8qdkln1hnalm)

[4.1. Server Application Performance 8](#_r0xh5gu46o8i)

[4.2. Client Application Performance 8](#_d50whm8sxwym)

[**5. Server application 9**](#_pq0g2cmj271z)

[5.1. Multiple Concurrent Connections: 9](#_xahdh1yx4hiz)

[5.2. Threading and Data Integrity: 9](#_1md6w0qnygb0)

[5.3. Flexibility in Implementation: 9](#_qrjgokv0nvfn)

[5.4. Usage of Allowed APIs: 9](#_2yexx1ncd3dr)

[**6. Client application 10**](#_8ocl5p89nmhi)

[6.1. Simulated Sensor Inputs: 10](#_hc20xbnmwnsb)

[6.2. Real-Time Transaction Simulation: 10](#_u1fcj5nhi61n)

[6.3. Programming Techniques: 10](#_aenq4ld5t8jh)

[6.4. API Usage and Low-Level Control: 10](#_dsqa15e6wae2)

# **Project Description**

This project is a Transaction Server for Road Toll Services designed to monitor vehicle movements along a 100‑kilometer highway. The system tracks vehicles by their plate numbers (following the Philippine format) and calculates toll fees upon exit. Instead of using a traditional database, all transaction data is managed in memory through custom data structures.

The server is built to handle multiple concurrent connections from simulated toll booth clients. In our simulation, 76 toll booths (representing 16 regular entry/exit points with 4 booths each and 2 toll plazas with 6 booths each) connect to the server. Each booth sends an ENTRY message when a vehicle enters and an EXIT message after a simulated delay that mimics travel time; the server then computes the fee and updates live statistics.

# **Project Capability and Scope**

## **Capabilities**

* + 1. **Concurrent Real-Time Tracking**

The server handles up to 76 simultaneous connections, processing messages as soon as they are received from each toll booth.

* + 1. **Fee Computation**

Toll fees are calculated using a predefined formula that considers a base fee, distance (booth difference), and the travel time between ENTRY and EXIT events.

* + 1. **In-Memory Data Management**

Instead of a database, the system uses Python dictionaries to track active vehicles and aggregate statistics.

* + 1. **Statistical Aggregation**

The server continuously updates and displays real-time statistics such as the current number of vehicles on the highway, total vehicles that have completed their transaction, and total fees collected.

## **Scope**

* + 1. **Low-Level Implementation**

The solution is built with fundamental programming constructs (sockets, threads, and basic data structures) without reliance on high-level frameworks or database engines.

* + 1. **Custom Communication Protocol**

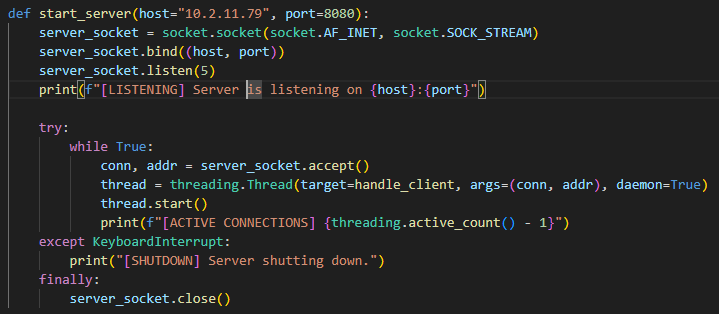
A custom message format (TYPE;PLATE\_NUMBER;TIMESTAMP;BOOTH\_ID) is used for all transactions between clients and the server.

* + 1. **Simulation Environment**
    2. Toll booths are fully simulated software clients that mimic sensor inputs (e.g., the generation of vehicle plate numbers) and operate on standard PCs or embedded systems.

# **System Architecture**

## **Description of the overall logical (network, hardware and software) architecture**

* + 1. **Network Layer** The server listens on a designated port using TCP/IP; each simulated toll booth client establishes its own socket connection to the server.



*Figure 1: start\_server() function from server.py*

* + 1. **Hardware** Both the server and client applications are designed to run on standard desktop PCs or embedded devices. No specialized sensor hardware is required as input is simulated by the client software.
    2. **Software** The entire system is written in Python. The server application runs continuously to accept connections, process incoming messages, and update statistics in real time. Simulated client applications (toll booths) generate entry and exit transactions using threads and socket communication.

## **Description of the server software architecture (include modules, library, probable classes or functions)**

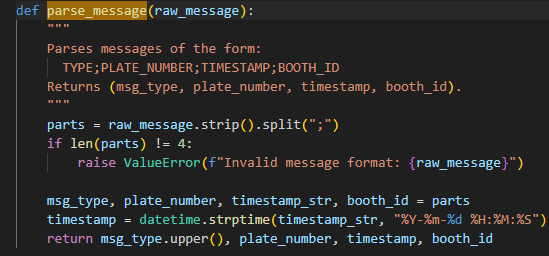
The server application is modularized into several key components:

* + 1. **Network Module**

Responsible for establishing the TCP server socket, accepting incoming connections, and launching a new thread (via the handle\_client function) for each client connection.

* + 1. **Message Parser Module**

Uses the parse\_message() function to decompose the incoming semicolon-separated messages into structured data, extracting the type (ENTRY/EXIT), vehicle plate number, timestamp, and booth ID.

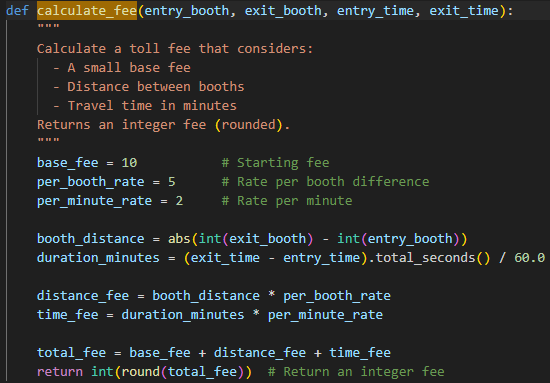
****

*Figure 2: parse\_message() function from server.py*

* + 1. **Transaction Processor Module**

Contains:

* + - * process\_entry(): Records the vehicle’s entry by adding an entry in the active vehicles dictionary.
      * process\_exit(): Retrieves the corresponding entry record, computes the fee using calculate\_fee(), updates the global statistics (current count, total vehicles, total fees), and removes the vehicle from the active list.
      * calculate\_fee(): Implements the fee calculation based on a base fee, booth difference, and elapsed time.



*Figure 3: calculate\_fee() function from server.py*

* + 1. **Data Storage Module**

Maintains all current and historical transaction data in memory via dictionaries (active\_vehicles and statistics).

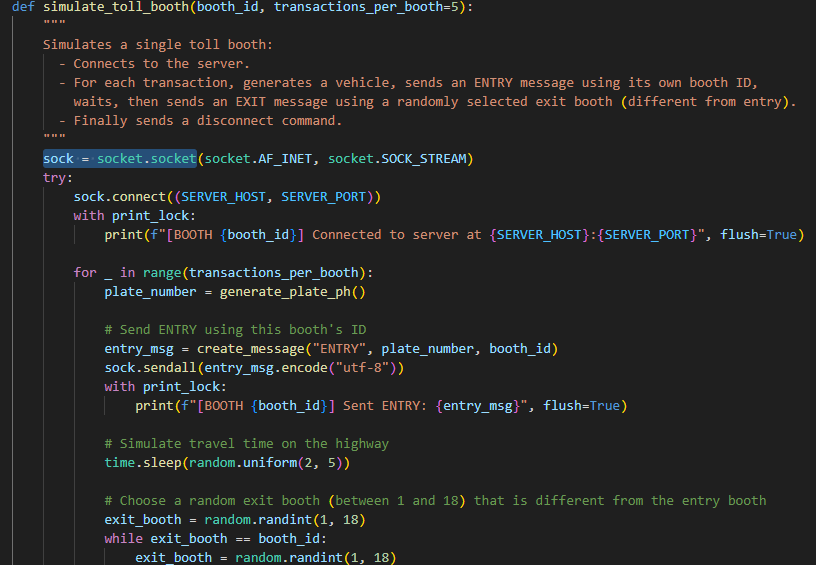
* + 1. **Concurrency Management Module**

Uses a global lock (data\_lock) to ensure thread safety when multiple threads update shared data structures.



*Figure 4: global lock declaration of data\_lock from server.py*

## **Description of the client hardware and software architecture (include sensors, library, probable classes or functions**

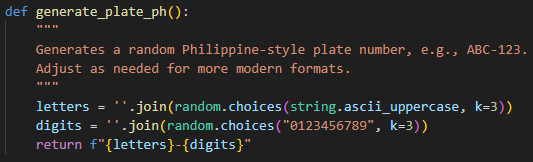


*Figure 5: simulate\_toll\_booth() function from client.py*

The client side simulates toll booth operations and comprises the following modules.

* + 1. **Plate Number Input Module**

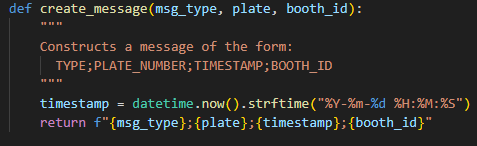
Generates vehicle plate numbers following the Philippine format using the generate\_plate\_ph() function.



*Figure 6: generate\_plate\_ph() function from client.py*

* + 1. **Message Generation Module**

Forms properly formatted messages (using create\_message()) that include the transaction type, generated plate number, timestamp, and booth ID.



*Figure 7: create\_message() function from client.py*

* + 1. **TCP Communication Module**

Each toll booth (simulated by the simulate\_toll\_booth() function) opens a socket connection to the server, sends ENTRY and EXIT messages, and finally sends a disconnect command.

* + 1. **Simulated Delay Module**

Uses time.sleep() with random intervals to simulate realistic travel times between an ENTRY and an EXIT event.

* + 1. **Real-Time Display Module**

Outputs transaction logs, including the sending of messages and connection status, to the console for monitoring.

# **Project Performance**

## **Server Application Performance**

The server application is designed to handle multiple simultaneous connections efficiently.



*Figure 8: server.py listening for new connections*

* + 1. **Concurrency**

Each client connection is processed on its own thread, ensuring that messages from up to 76 toll booths can be managed concurrently.

* + 1. **Real-Time Statistics**

The server continuously updates and prints statistics (current vehicle count, total processed vehicles, and fees collected) after each transaction.

* + 1. **Resource Management**

The use of a global lock (data\_lock) ensures data consistency, though it may become a bottleneck at very high loads. In the context of our project simulation, this model is sufficient.

## **Client Application Performance**

The client application simulates toll booth operations with the following performance characteristics.

* + 1. **Threaded Operations**

Each of the 76 toll booths runs on its own thread. A slight stagger in thread startup avoids connection flooding.

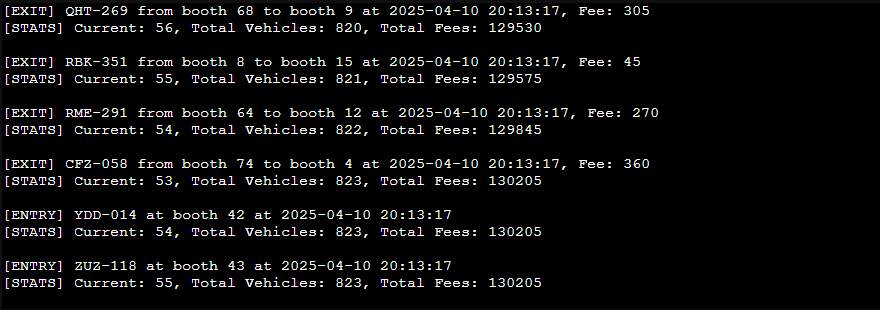
* + 1. **Simulated Delays**

Randomized delays are built into the booth simulation to create realistic travel times and to moderate the rate of transactions sent to the server.

* + 1. **Efficient Communication**

The use of socket connections ensures that transactions (ENTRY, EXIT, DISCONNECT) are reliably transmitted to the server.

# **Server application**



*Figure 9: Sample run of server.py inside ccscloud.dlsu.edu.ph*

The server application is designed as a low-level, multi-threaded solution that aligns with our project specifications. Key highlights include:

## **Multiple Concurrent Connections:**

Each client connection is handled in its own thread, showcasing parallel processing.

## **Threading and Data Integrity:**

The use of locks ensures safe concurrent updates to shared in-memory data structures.

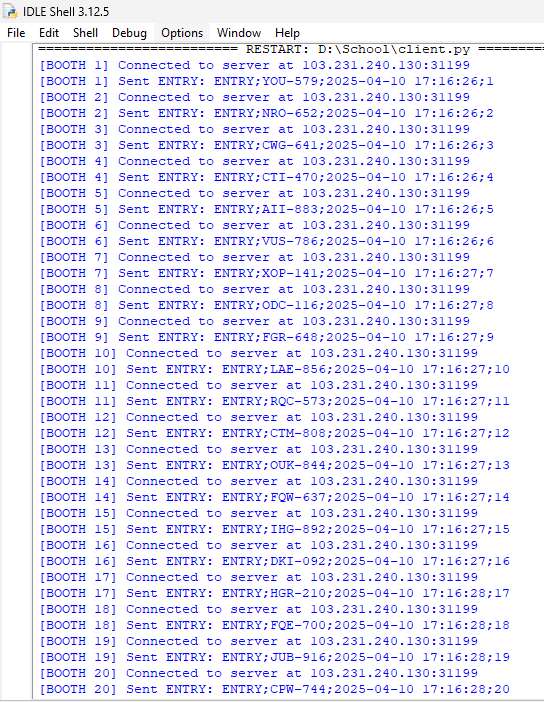
## **Flexibility in Implementation:**

Although implemented in Python, the techniques used are applicable across languages that support threading and socket programming.

## **Usage of Allowed APIs:**

The solution leverages the standard socket and threading libraries, adhering to class-learned programming techniques.

# **Client application**



*Figure 10: Sample run of client.py*

The client application simulates the real-world behavior of toll booths with:

## **Simulated Sensor Inputs:**

Vehicle plate numbers are randomly generated, and a custom message format is used to simulate sensor-based inputs.

## **Real-Time Transaction Simulation:**

Each toll booth runs in its own thread, connecting to the server, sending ENTRY/EXIT messages with delays, and then disconnecting.

## **Programming Techniques:**

Built using Python’s threading and socket libraries, it reflects the practical application of asynchronous programming techniques as taught in class.

## **API Usage and Low-Level Control:**

The client strictly uses low-level APIs (i.e., no high-level frameworks) to ensure all underlying processes are transparent and controllable.