Telemetry Data Simulation and Packet Validation

*Multithreading Data Parser

1st Shivam Jadhav
dept. Computer Science (of Aff.)
University Of North Texas (of Aff.)
Denton, United States
Shivamjadhav@my.unt.edu

***Abstract*—This project implements a real-time telemetry system for fleet management, enabling secure communication between multiple IoT devices and a central server. Using socket-based TCP/IP communication, the system processes and stores vehicle data, including location, fuel levels, and speed, with CRC32 data integrity checks. Multithreading allows the server to handle data from over 100 devices simultaneously. The system supports live tracking and detailed analysis of fleet performance, route optimization, and driving behavior, offering valuable insights for logistics and fleet management.**

***Index Terms*—telemetry, fleet management, IoT (Internet of Things), Data Integrity, TCP/IP Communication, Real-Time Data Transmission, Data Parsing, Multithreading,Vehicle Tracking**

I. Introduction

This project develops a system to manage and analyze data from multiple IoT-enabled fleet devices. It utilizes secure TCP/IP socket communication to transmit real-time data from 100+ devices, including fuel levels, speed, and location. The server processes and stores this data for analysis, providing insights into fleet performance, fuel efficiency, and vehicle behavior. Using multithreading, the system ensures scalability and efficient data handling. This solution enhances fleet management by offering real-time tracking, route optimization, and performance monitoring, with a focus on data integrity and accurate packet parsing.

II. Related Work

The concept of real-time fleet management is explored in many research papers, such as the work by [1] which discusses Fleet Management and Control Systems (FMCS) and focuses on improving safety, efficiency, and productivity in public transport using Intelligent Transportation Systems (ITS). Another relevant work [2] discusses an IoT-based fleet management system that integrates real-time tracking, predictive maintenance, and fuel efficiency monitoring. This approach utilizes a combination of sensor networks and cloud-based platforms to provide accurate fleet data analysis and management. The system is designed to enhance decision-making in fleet management by offering better insights into vehicle health, driver behavior, and route optimization.

While these studies emphasize vehicle tracking and control for public transport in cities, this project extends the ability to provide bandwidth capable of handling many IoT devices, which communicate via 2G SIM cards and send data packets over the network to a server. This enables fleet owners to receive live data from entire fleets or multiple organizations, allowing for real-time analysis of fleet data. We utilize multi-threading, socket-based communication, and data verification techniques to improve scalability and performance for fleet management systems.

Identify applicable funding agency here. If none, delete this.

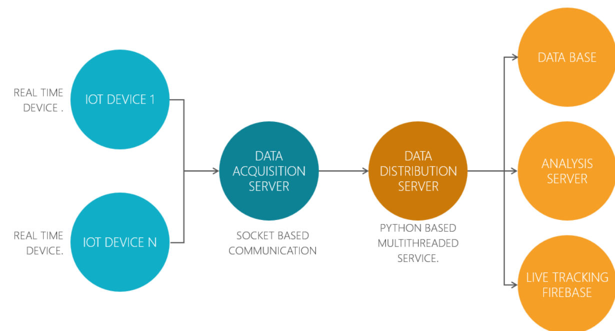

Fig. 1. workflow

III. Architecture Design

This project features three components: a device simulator, a data acquisition server, and a data distribution server, as shown in Fig. 1. Together, they offer a unified solution for managing various devices, focusing on device authentication, data integrity, and data parsing. Decoupling the data acquisition and distribution servers enhances overall system efficiency.

Devices are configured to send data every two seconds through a program that constructs a data packet using a custom "Packet" class. This packet simulates parameters like sensor readings and location. The device connects to the server via Socket.IO, calculates the CRC, and sends the completed packet.

Multiple devices can connect to a Data Acquisition (DAC), which verifies data integrity using a packet number. If the client receives a positive acknowledgment, it can send another packet; otherwise, it will resend the last one.

The data distribution server then separates the data packet into a header, body, and CRC, parsing the information to

extract sensor data and device metadata, which it stores in the local file system for future use.

IV. Data Packet Protocol V1.0

To send data effectively, we need to design a data packet protocol before implementation. This design should ensure that information is compressed into as few bytes as possible, as network bandwidth is limited and IoT devices have restricted memory. Compiling all sensor data and metadata into a packet must facilitate easy retrieval and minimize overhead during transmission, which is crucial for efficient network communication. Our data packet protocol version 1.0 consists of three components. Table I depicts each section's memory in bytes,

TABLE I
Data Packet Structure

Section	Description	Size (Bytes)
Header	Contains metadata for data identification	23
Body	Main content/data payload	24
CRC	Cyclic Redundancy Check for data integrity	4

with a total data packet size of 51 bytes. Further details of each field in each section for the data packet are described in Table II.

TABLE II
Detailed Data Packet Structure

Section	Field	Size (Bytes)
Header	Packet Start	1
	Packet Length	1
	Header Length	1
	Packet Number	4
	Protocol Version (ASCII)	4
	IMEI	7
	Timestamp	4
	Header End	1
Body	Start of Body	1
	Body Length	1
	Timestamp	4
	Latitude (Lat)	4
	Longitude (Lng)	4
	Speed	4
	Fuel	4
	Active Status	1
	End of Body	1
CRC	CRC Check	4

V. System Design and Architecture

Overview of Components

A. Device Simulator

here in this project report we shall make use of device simulator which is an essential component The device simulator mimics multiple vehicles, each sending periodic data packets to the server. Each packet consists of metadata and sensor data, including fuel levels, speed, and location. The data is structured in a predefined format, which includes a header (with IMEI, timestamp, etc.) and a body (with sensor data).

B. Data Acquisition Server

This server receives data from the devices using TCP/IP socket communication. It is capable of handling connections from over 100 devices simultaneously using multithreading. The server performs packet verification, parses the data, and stores it for further analysis.

C. Data Distribution/Parser

After receiving the data, the server validates the integrity using CRC32, then parses the packet to extract relevant data points like fuel usage, speed, and location. The parsed data is stored in a database for later analysis.

VI. Methods and Implementation

A. Device Simulator

```
import random
from datetime import datetime as dt

class DataPacket:
    def __init__(self, imei, protocolversion='v1',
        packetcount=0, packet_frequency=2, lat=0,
        lng=0, fuellevel=0, active=1, speed=0):
        self.imei = imei
        self.protocolversion = protocolversion
        self.packetcount = packetcount
        self.fuellevel = fuellevel if fuellevel else
             random.randint(0, 100)
        self.lat = lat if lat else random.uniform
            (20, 90)
        self.lng = lng if lng else random.uniform
            (-180, 180)
        self.active = active
        self.speed = speed if speed else random.
            randint(0, 100)
        self.packet_number = 0
        self.packet_frequency = packet_frequency

    def get_lat(self):
        self.lat += random.uniform(-0.00001,
            0.00001)  # Move a few meters
        print(f"Lat:␣{self.lat}")
        p = float_to_bytes(self.lat)
        print("len==␣lat=", len(p))
        return p

    def get_lng(self):
        self.lng += random.uniform(-0.00001,
            0.00001)
        print(f"Lng:␣{self.lng}")
        return float_to_bytes(self.lng)

    def get_speed(self):
        self.speed += random.uniform(-0.5, 0.5)
        print(f"Speed:␣{self.speed}")
        return float_to_bytes(self.speed)

    def get_fuel_level(self):
        self.fuellevel += random.uniform(-2.5, 0)
        print(f"Fuel␣level:␣{self.fuellevel}")
        return float_to_bytes(self.fuellevel)

    def get_packet_number(self):
        max_packet_number = 60 * 60 * 24
        max_packet_number = max_packet_number //
            self.packet_frequency
        if self.packet_number >= max_packet_number:
            self.packet_number = 0
        self.packet_number += 1
```

```python
        return int_to_bytes(self.packet_number, 4)

    def get_header(self):
        header = b'$'
        header += int_to_bytes(51, 1)
        header += int_to_bytes(23, 1)
        header += self.get_packet_number()
        header += b'v1.0'
        header += int_to_bytes(self.imei, 7)
        header += int_to_bytes(int(dt.now().
            timestamp()), 4)
        header += b'@'
        return header

    def get_body(self):
        packet = b'&'  # Start byte
        packet += int_to_bytes(24, 1)
        packet += int_to_bytes(int(dt.now().
            timestamp()), 4)
        packet += self.get_lat()
        packet += self.get_lng()
        packet += self.get_speed()
        packet += self.get_fuel_level()
        packet += self.active.to_bytes(1, byteorder=
            'big')
        packet += b'#'  # End byte
        return packet

    def generate_packet(self):
        header = self.get_header()
        body = self.get_body()
        print(f"Header: {header}")
        print(f"Body: {body}")
        crc = calculate_crc32(header + body)
        crc = int_to_bytes(crc, 4)
        return header + body + crc
```

This custom class is designed to create a simulated IoT device that forms fixed-size data packets for transfer over a socket connection to a server. It allows us to test the server's ability to parse information from these data packets. This setup can support an unlimited number of simulated devices. Once physical devices are available, we can test the server before deploying production IoT devices. Through this process, we can assess the server's stress tolerance and its ability to handle errors during data transmission, as well as evaluate its fault tolerance.

B. Data Aquisition Server

```python
import socket
import threading
import sys
from packet_handler import handle_client
from utils import monitor_input

# Global event to control server shutdown
server_running = threading.Event()
client_threads = []  # List to keep track of all
    threads

# Function to run the server
def run_server(host='0.0.0.0', port=12345):
    with socket.socket(socket.AF_INET, socket.
        SOCK_STREAM) as server_socket:
        server_socket.bind((host, port))
        server_socket.listen(5)  # Listen for
            incoming connections
        print(f"Server is listening on {host}:{port}
            ")

        while server_running.is_set():  # Keep
            accepting new clients if server is
            running
            conn, addr = server_socket.accept()
            client_thread = threading.Thread(target=
                handle_client, args=(conn, addr))
            client_thread.start()
            client_threads.append(client_thread)
            print(f"Started thread for client {addr}
                ")

# Function to stop the server gracefully
def stop_server():
    print("\nStopping server...")
    server_running.clear()  # Stop the server
    print("Server stopped.")

    # Wait for all client-handling threads to finish
    for thread in client_threads:
        thread.join()

if __name__ == "__main__":
    server_running.set()

    input_thread = threading.Thread(target=
        monitor_input, args=(stop_server,))
    input_thread.daemon = True  # Daemonize the
        input thread so it ends when the program
        ends
    input_thread.start()

    # Start the server
    run_server()

    # Once the server stops, exit the program
    print("Server has completely stopped. Exiting 
        program.")
    sys.exit(0)
```

This server acts as the first contact point for IoT devices, and our primary goal is to test the simulator. Using Socket.IO, we can generate unlimited devices that send data to the DAC server. The server tests the data packets and, if errors are found, it sends an error code. If the data is error-free, it sends an acknowledgment with the packet number. If the device receives this acknowledgment with the same packet number, it proceeds to send another packet from memory.

C. Data Parser

```python
from utils import bytes_to_int,bytes_to_float
def parse_header(header):
    packet_number=bytes_to_int(header[3:7])
    protocol_version=header[7:11].decode()
    imei=bytes_to_int(header[11:18])
    timestamp=bytes_to_int(header[18:22])
    return {'packet_number':packet_number,
            'protocol_version':protocol_version,
            'imei':imei,
            'timestamp':timestamp
            }
def parse_body(body):
    start_byte=body[0:1].decode()
    if start_byte!='&':
        return None
    body_length=bytes_to_int(body[1:2])
    timestamp=bytes_to_int(body[2:6])
    latitude=bytes_to_float(body[6:10])
    longitude=bytes_to_float(body[10:14])
    speed=bytes_to_float(body[14:18])
    fuel_level=bytes_to_float(body[18:22])
```

```
22      return {'start_byte':start_byte,
23              'body_length':body_length,
24              'timestamp':timestamp,
25              'latitude':latitude,
26              'longitude':longitude,
27              'speed':speed,
28              'fuel_level':fuel_level
29              }
30  # Function to parse the received packet and return
        structured data
31  def parse_packet(packet):
32      try:
33          if packet[0:1].decode()=='$':
34              packet_length=bytes_to_int(packet[1:2])
35              header_length=bytes_to_int(packet[2:3])
36              header=packet[0:header_length]
37              body=packet[header_length:]
38              header_data=parse_header(header)
39              body_data=parse_body(body)
40              packet_data={**header_data,**body_data}
41              return packet_data,header_data['
                    packet_number']
42
43
44          return None,None
45      except Exception as e:
46          print(f"Unexpected error: {e}")
47          return None,None
```

This data parser service will extract information from data packets and distribute it to various services where it can be utilized. For instance, the data can be stored in a NoSQL database for analysis purposes such as fuel monitoring, fuel consumption analysis, vehicle performance analysis, and cost evaluation of vehicle routes. Additionally, by integrating with Firebase, this service can provide a live tracking solution for fleet owners in the logistics sector.

VII. Results and Evaluation

A. System Performance

The system successfully handled connections from 100+ devices simultaneously without noticeable latency. Multithreading ensured that the server was responsive even under heavy load. The CRC32 verification mechanism efficiently detected and discarded corrupted packets. The parsing mechanism was able to handle large data sets with minimal overhead, storing the parsed data into the database for real-time analysis.

B. Fleet Performance Analysis

The system tracked vehicle locations in real-time, displaying a map showing the movement of each vehicle. Fuel utilization and vehicle speed data were analyzed to identify performance trends and optimize fleet operations.

VIII. Conclusion

This project successfully addressed the challenges of managing real-time data from a large fleet of vehicles. By utilizing TCP/IP socket communication, multithreading, and data integrity mechanisms, the system ensures reliable communication, data verification, and real-time performance analysis. The system provides significant benefits for fleet management and logistics optimization.

IX. Future Work

A. Future improvements could include

- Implementing predictive maintenance algorithms
- Real-time anomaly detection in fleet performance
- Integration with advanced route optimization tools
- Expanding the database to support historical data analysis

References

[1] A. of the Paper, "Fleet management and control systems (fmcs)," in *2019 2nd Latin American Conference on Intelligent Transportation Systems (ITS LATAM)*. IEEE, 2019, pp. 1–6. [Online]. Available: https://ieeexplore.ieee.org/document/8721347

[2] P. Singh, M. S. Suryawanshi, and D. Tak, "Smart fleet management system using iot, computer vision, cloud computing and machine learning technologies," in *2019 IEEE 5th International Conference for Convergence in Technology (I2CT)*, 2019, pp. 1–8.