

FUNDAMENTALS OF DISTRIBUTED SYSTEMS

Assignment 1

Anjitha Ravikumar

G24AI2043

PG Diploma in Data Engineering

IIT Jodhpur

Github Link: <https://github.com/anjitharavikumar/FDS-Assignment>

Table of Contents

1. Introduction	3
2. Tools & Technologies Used	3
3. Vector Clocks and Causal Ordering	3
3.1 Overview.....	3
3.2 Objective	3
3.3 Components.....	4
3.4 Architecture	4
3.5 Process.....	4
3.6 Screenshots	5
3.6.1 node.py	5
3.6.2 client.py	6
3.6.3 Dockerfile	6
3.6.4 Docker-compose.yml.....	7
3.6.5 import json.py.....	8
3.6.6 Docker Desktop	9
4. Dynamic Load Balancing for a Smart Grid	9
4.1 Overview.....	9
4.2 Objective	9
4.3 Components.....	9
4.4 Architecture	10
4.5 Process.....	10
4.6 Screenshots.....	11
4.6.1 substation_service/main.py	11
4.6.2 load_balancer/main.py	11
4.6.3 Docker-compose.yml.....	12
4.6.4 load_balancer/Dockerfile	12
4.6.5 load_tester/test.py.....	13
4.6.6 Docker Desktop	13

1. Introduction

This report documents the implementation of two distributed systems:

- **Vector Clocks and Causal Ordering** – used to track the partial order of events in distributed systems and ensure consistency across data replicas.
- **Dynamic Load Balancing in Smart Grids** – focused on simulating real-time distribution of EV charging loads across substations using monitoring and intelligent routing mechanisms.

Both systems were developed in **Python**, containerized with **Docker**, and orchestrated using **Docker Compose**.

2. Tools & Technologies Used

- **Docker** – Enables containerization of services to emulate distributed nodes
- **Docker Compose** – Manages and orchestrates multiple interdependent services
- **Python** – Used as the core programming language for implementing backend logic
- **Flask** – A lightweight web framework used to build RESTful APIs
- **Prometheus** – Collects and scrapes metrics from running services
- **Grafana** – Visualizes real-time data through interactive dashboards

3. Vector Clocks and Causal Ordering

3.1 Overview

Vector Clocks track event causality in distributed systems. Each node maintains counters representing events across all nodes, enabling conflict resolution and proper event ordering. This ensures causal consistency - if one event affects another, all nodes process them in the correct sequence. Essential for distributed databases and collaborative systems.

3.2 Objective

Implement a key-value store across 3 nodes that maintains causal consistency using vector clocks. The system should detect out-of-order messages and delay their processing until causal dependencies are satisfied.

3.3 Components

- **node.py** - Handles the distributed key-value store operations and implements vector clock synchronization logic
- **docker-compose.yml** - Configures and deploys the 3-node cluster with assigned port mappings and node identifiers
- **client.py** - Provides a testing interface for sending PUT/GET requests and inspecting vector clock states

3.4 Architecture

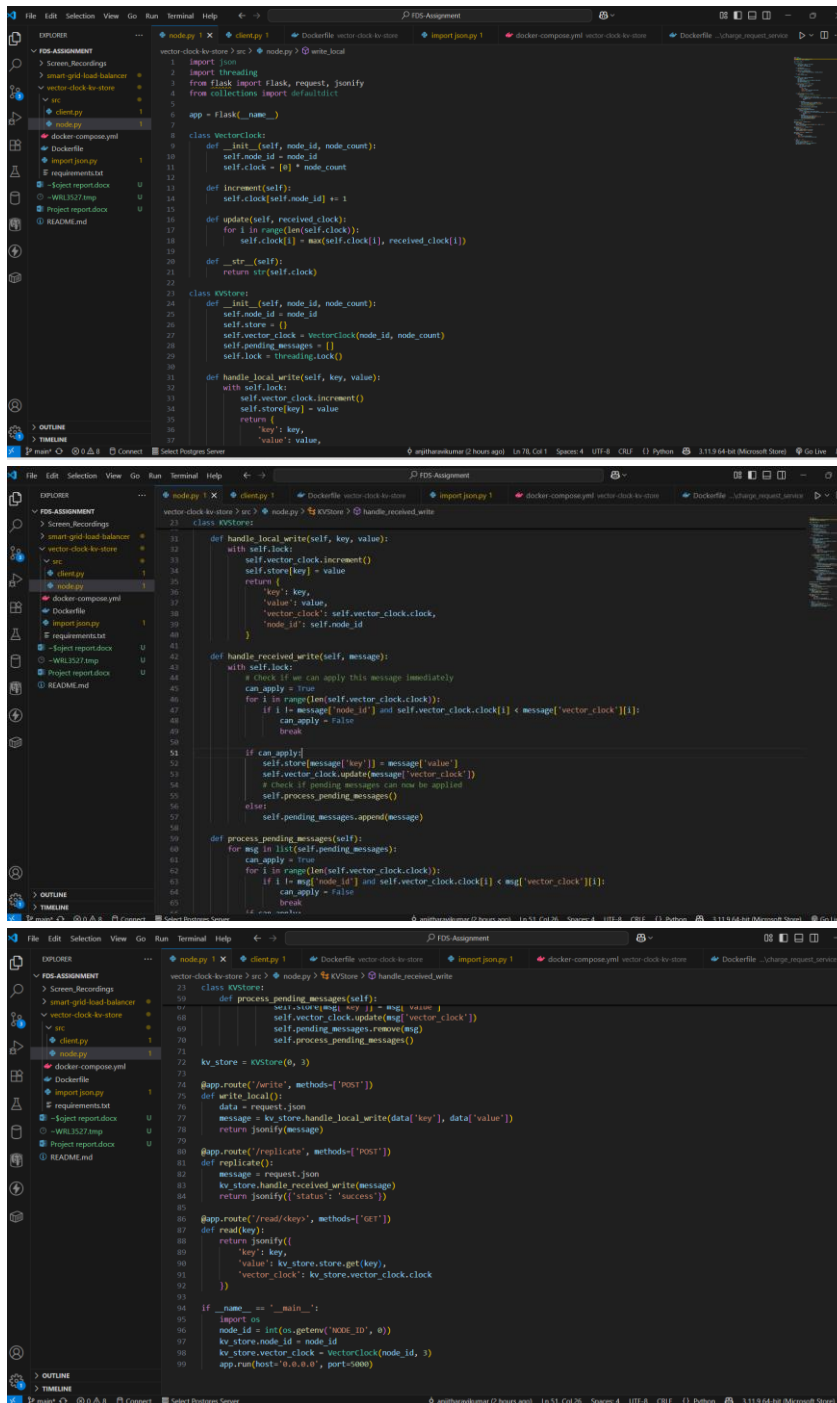
- The system has three distributed nodes, each running as a Python Flask service within a Docker container. Each node has its own local key-value store and a vector clock to track causality.
- When a node handles a local PUT request, it increments its vector clock and broadcasts the update to other nodes. Receiving nodes compare vector clocks to decide whether to apply the update immediately or buffer it for later.
- Buffered messages are periodically reviewed and applied as soon as their causal dependencies are satisfied.
- All nodes interact via REST APIs over a common Docker network.

3.5 Process

- node.py created using Flask to simulate each node in the distributed
- system.
- Each node maintains a local key-value store, a vector clock list $[i, j, k]$ and a buffer for delayed messages.
- When a local PUT occurs, the node updates its vector clock and sends the update along with the clock to other nodes.
- Upon receiving a replicated message, a node verifies the causal delivery condition; if unmet, the message is buffered.
- Buffered messages are regularly checked to determine if they can be delivered.
- Docker Compose is used to launch three separate containers, one for each node.

3.6 Screenshots

3.6.1 node.py



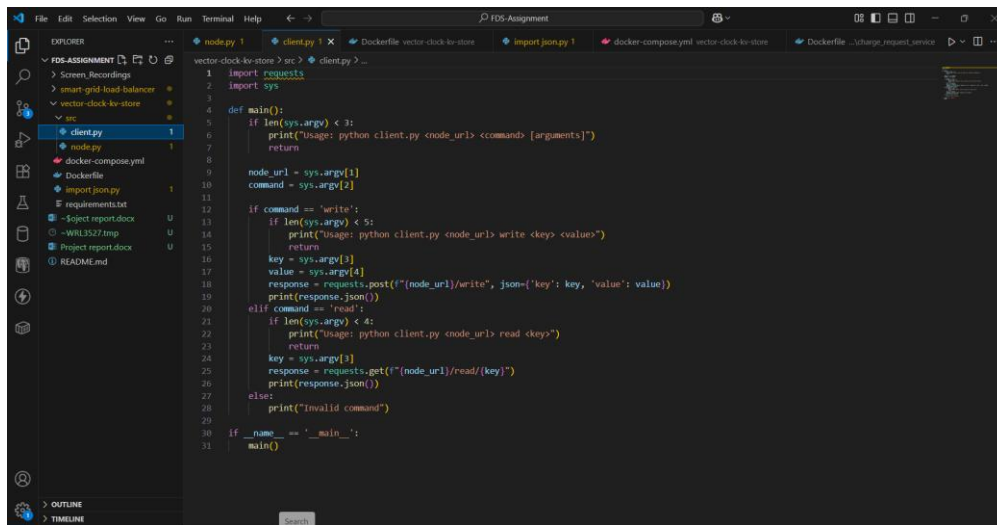
The image displays three sequential screenshots of a code editor, likely Visual Studio Code, showing the implementation of a distributed key-value store in Python. The code is organized into three main classes: `VectorClock`, `KVStore`, and `Node`.

First Screenshot: Shows the initial setup and the `VectorClock` class. The `VectorClock` class is defined with attributes `node_id`, `node_count`, and `clock`. It includes methods for `increment`, `update`, and `__str__`. The `KVStore` class is also defined with attributes `node_id`, `store`, `vector_clock`, and `pending_messages`. It includes methods for `handle_local_write`, `handle_received_write`, and `process_pending_messages`.

Second Screenshot: Shows the `Node` class and the `handle_received_write` method. The `Node` class is defined with attributes `node_id`, `node_count`, and `clock`. It includes methods for `handle_local_write`, `handle_received_write`, and `process_pending_messages`. The `handle_received_write` method is implemented to handle incoming write requests from other nodes.

Third Screenshot: Shows the `process_pending_messages` method and the `__main__` block. The `process_pending_messages` method is implemented to process pending messages from other nodes. The `__main__` block sets up the application, including the `app` object, the `node_id`, and the `kv_store` object. It also defines the `write` and `replicate` endpoints.

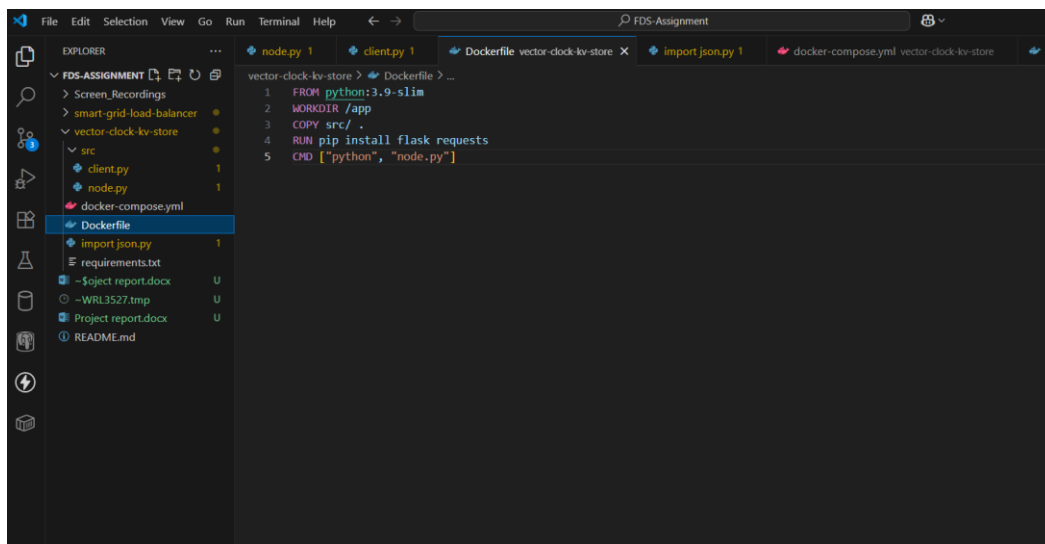
3.6.2 client.py



The screenshot shows the Visual Studio Code editor with the file explorer on the left and the code editor in the center. The file explorer shows a project named 'FDS-Assignment' with several files and folders. The code editor displays the contents of 'client.py', which is a Python script for interacting with a Docker-based key-value store. The script uses the 'requests' library to send HTTP requests to the store's API. It includes a 'main()' function that handles command-line arguments and performs operations like 'write' and 'read' based on the provided arguments. The script also includes a 'requirements.txt' file in the project root.

```
1 import requests
2 import sys
3
4 def main():
5     if len(sys.argv) < 3:
6         print("Usage: python client.py <node_url> <command> [arguments]")
7         return
8
9     node_url = sys.argv[1]
10    command = sys.argv[2]
11
12    if command == 'write':
13        if len(sys.argv) < 5:
14            print("Usage: python client.py <node_url> write <key> <value>")
15            return
16        key = sys.argv[3]
17        value = sys.argv[4]
18        response = requests.post(f"{node_url}/write", json={'key': key, 'value': value})
19        print(response.json())
20    elif command == 'read':
21        if len(sys.argv) < 4:
22            print("Usage: python client.py <node_url> read <key>")
23            return
24        key = sys.argv[3]
25        response = requests.get(f"{node_url}/read/{key}")
26        print(response.json())
27    else:
28        print("Invalid command")
29
30 if __name__ == '__main__':
31     main()
```

3.6.3 Dockerfile



The screenshot shows the Visual Studio Code editor with the file explorer on the left and the code editor in the center. The file explorer shows a project named 'FDS-Assignment' with several files and folders. The code editor displays the contents of 'Dockerfile', which is a Dockerfile for building a container image. The Dockerfile starts with the 'FROM' instruction to use the 'python:3.9-slim' base image. It then sets the working directory to '/app', copies the source code from the current directory, and runs 'pip install flask requests' to install the required dependencies. Finally, it sets the command to run the application using 'python' and 'node.py'.

```
1 FROM python:3.9-slim
2 WORKDIR /app
3 COPY src/ .
4 RUN pip install flask requests
5 CMD ["python", "node.py"]
```

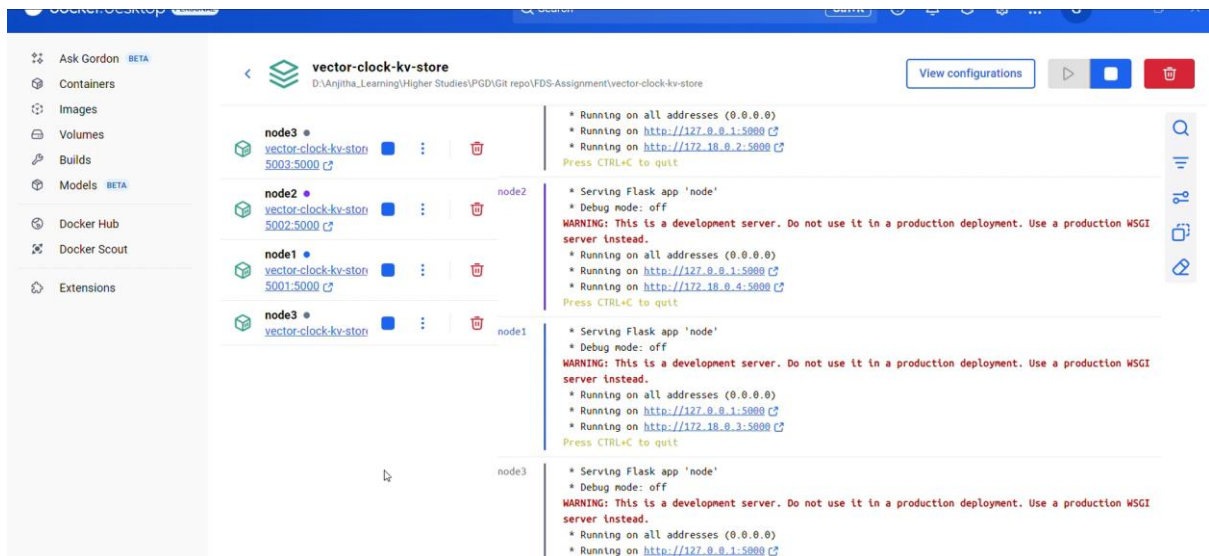
3.6.4 Docker-compose.yml

```
1 version: '3.8'
2
3 DRun All Services
4 services:
5   DRun Service
6   node1:
7     build: .
8     environment:
9       - NODE_ID=0
10    ports:
11      - "5001:5000"
12
13 DRun Service
14 node2:
15   build: .
16   environment:
17     - NODE_ID=1
18   ports:
19     - "5002:5000"
20
21 DRun Service
22 node3:
23   build: .
24   environment:
25     - NODE_ID=2
26   ports:
27     - "5003:5000"
```

3.6.5 import json.py

```
1 import json
2 import threading
3 from flask import Flask, request, jsonify
4 from collections import defaultdict
5
6 app = Flask(__name__)
7
8 class VectorClock:
9     def __init__(self, node_id, node_count):
10         self.node_id = node_id
11         self.clock = [0] * node_count
12
13     def increment(self):
14         self.clock[self.node_id] += 1
15
16     def update(self, received_clock):
17         for i in range(len(self.clock)):
18             self.clock[i] = max(self.clock[i], received_clock[i])
19
20     def __str__(self):
21         return str(self.clock)
22
23 class KVStore:
24     def __init__(self, node_id, node_count):
25         self.node_id = node_id
26         self.store = {}
27         self.vector_clock = VectorClock(node_id, node_count)
28         self.pending_messages = []
29         self.lock = threading.Lock()
30
31     def handle_local_write(self, key, value):
32         with self.lock:
33             self.vector_clock.increment()
34             self.store[key] = value
35         return {
36             'key': key,
37             'value': value,
38         }
39
40     def handle_received_write(self, message):
41         with self.lock:
42             can_apply = True
43             for i in range(len(self.vector_clock.clock)):
44                 if i != message['node_id'] and self.vector_clock.clock[i] < message['vector_clock'][i]:
45                     can_apply = False
46                     break
47             if can_apply:
48                 self.store[message['key']] = message['value']
49                 self.vector_clock.update(message['vector_clock'])
50                 self.process_pending_messages()
51             else:
52                 self.pending_messages.append(message)
53
54     def process_pending_messages(self):
55         for msg in list(self.pending_messages):
56             can_apply = True
57             for i in range(len(self.vector_clock.clock)):
58                 if i != msg['node_id'] and self.vector_clock.clock[i] < msg['vector_clock'][i]:
59                     can_apply = False
60                     break
61             if can_apply:
62                 self.store[msg['key']] = msg['value']
63                 self.vector_clock.update(msg['vector_clock'])
64                 self.pending_messages.remove(msg)
65                 self.process_pending_messages()
66
67 kv_store = KVStore(0, 3)
68
69 @app.route('/write', methods=['POST'])
70 def write_local():
71     data = request.json
72     message = kv_store.handle_local_write(data['key'], data['value'])
73     return jsonify(message)
74
75 @app.route('/replicate', methods=['POST'])
76 def replicate():
77     message = request.json
78     kv_store.handle_received_write(message)
79     return jsonify({'status': 'success'})
80
81 @app.route('/read/keys', methods=['GET'])
82 def read(key):
83     return jsonify({
84         'key': key,
85         'value': kv_store.store.get(key),
86         'vector_clock': kv_store.vector_clock.clock
87     })
88
89 if __name__ == '__main__':
90     import os
91     node_id = int(os.getenv('NODE_ID', 0))
92     kv_store.node_id = node_id
93     kv_store.vector_clock = VectorClock(node_id, 3)
94     app.run(host='0.0.0.0', port=5000)
```


3.6.6 Docker Desktop



Demo video is available in [github](#)

4. Dynamic Load Balancing for a Smart Grid

4.1 Overview

Dynamic load balancing intelligently distributes EV charging requests across substations based on real-time load. In a smart grid, where demand can spike suddenly, a centralized load balancer directs requests to the substation with the lowest load. This approach prevents overload, enhances efficiency, and improves overall system responsiveness. Tools like Prometheus are used to collect real-time data, while Grafana provides visualization of system performance.

4.2 Objective

To develop a dynamic, load-aware routing system that distributes electric vehicle charging requests across substations. The system should:

- Continuously monitor real-time load
- Route requests to the substation with the lowest load
- Visualize system metrics using Prometheus and Grafana

4.3 Components

- **substation_service/main.py**: Processes charging requests and exposes load metrics

- **load_balancer/main.py:** Contains the routing logic based on current load data
- **test.py:** Simulates EV charging requests to evaluate the load balancing system
- **prometheus.yml:** Sets up Prometheus to scrape metrics from substations
- **dashboard.json:** Configures the Grafana dashboard for real-time monitoring

4.4 Architecture

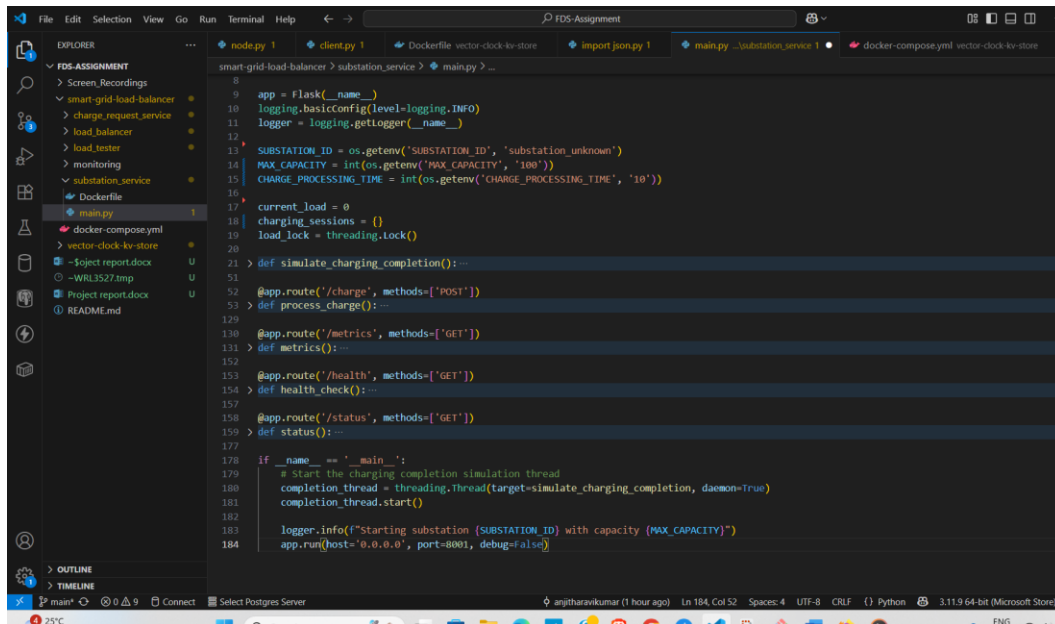
- The system includes two substation services, a centralized load balancer, and monitoring tools (Prometheus and Grafana).
- Substations handle EV charging requests and expose their current load through a /metrics endpoint.
- The load balancer regularly polls these metrics and directs new requests to the substation with the lowest load.
- Prometheus collects and stores time-series data from the substations.
- Grafana connects to Prometheus to display real-time load metrics on a custom dashboard.

4.5 Process

- Developed substation_service/main.py with /charge and /metrics endpoints.
- Built load_balancer/main.py to fetch substation metrics and route requests to the least-loaded node.
- Created test.py to simulate 50 EV charging requests.
- Set up Prometheus to scrape metrics from the substation services.
- Integrated Grafana with Prometheus and imported a custom dashboard for visualization.

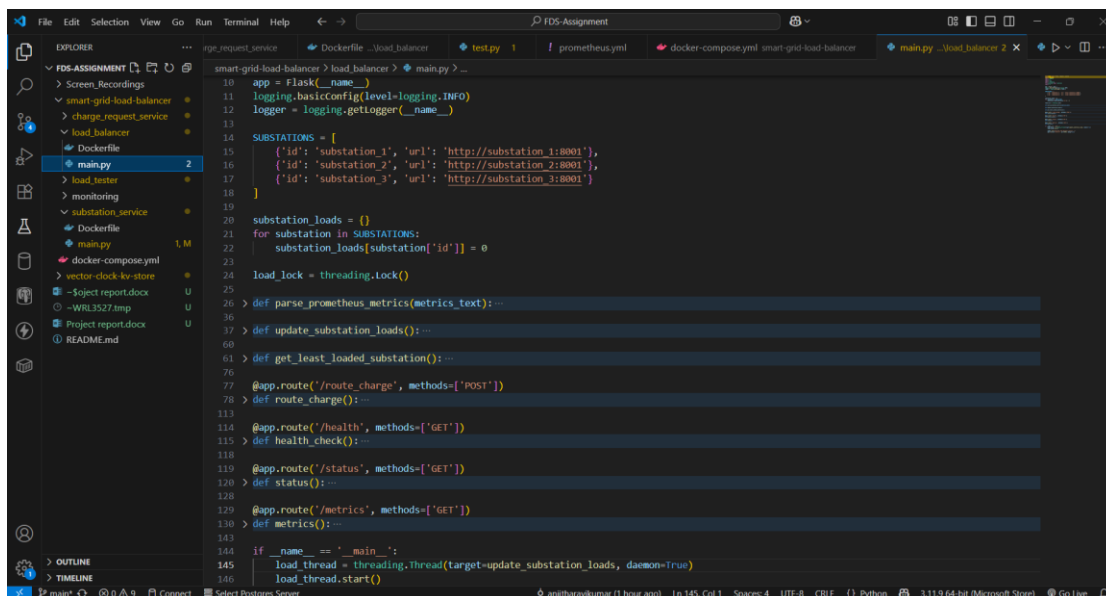
4.6 Screenshots

4.6.1 substation_service/main.py



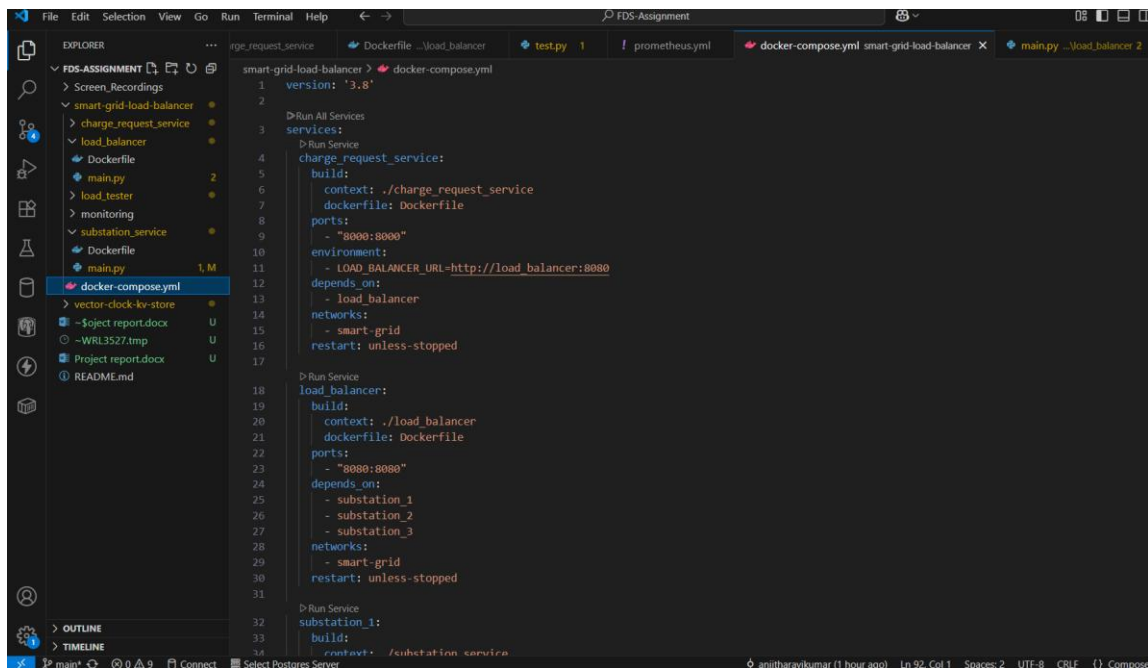
```
smart-grid-load-balancer > substation_service > main.py > ...
8
9 app = Flask(__name__)
10 logging.basicConfig(level=logging.INFO)
11 logger = logging.getLogger(__name__)
12
13 SUBSTATION_ID = os.getenv('SUBSTATION_ID', 'substation_unknown')
14 MAX_CAPACITY = int(os.getenv('MAX_CAPACITY', '100'))
15 CHARGE_PROCESSING_TIME = int(os.getenv('CHARGE_PROCESSING_TIME', '10'))
16
17 current_load = 0
18 charging_sessions = {}
19 load_lock = threading.Lock()
20
21 > def simulate_charging_completion(): ...
22
23 @app.route('/charge', methods=['POST'])
24 > def process_charge(): ...
25
26 @app.route('/metrics', methods=['GET'])
27 > def metrics(): ...
28
29 @app.route('/health', methods=['GET'])
30 > def health_check(): ...
31
32 @app.route('/status', methods=['GET'])
33 > def status(): ...
34
35 if __name__ == '__main__':
36     # Start the charging completion simulation thread
37     completion_thread = threading.Thread(target=simulate_charging_completion, daemon=True)
38     completion_thread.start()
39
40     logger.info(f"Starting substation {SUBSTATION_ID} with capacity {MAX_CAPACITY}")
41     app.run(host='0.0.0.0', port=8001, debug=False)
```

4.6.2 load_balancer/main.py



```
smart-grid-load-balancer > load_balancer > main.py > ...
10 app = Flask(__name__)
11 logging.basicConfig(level=logging.INFO)
12 logger = logging.getLogger(__name__)
13
14 SUBSTATIONS = [
15     {'id': 'substation_1', 'url': 'http://substation_1:8001'},
16     {'id': 'substation_2', 'url': 'http://substation_2:8001'},
17     {'id': 'substation_3', 'url': 'http://substation_3:8001'}
18 ]
19
20 substation_loads = {}
21 for substation in SUBSTATIONS:
22     substation_loads[substation['id']] = 0
23
24 load_lock = threading.Lock()
25
26 > def parse_prometheus_metrics(metrics_text): ...
27
28 > def update_substation_loads(): ...
29
30 > def get_least_loaded_substation(): ...
31
32 @app.route('/route_charge', methods=['POST'])
33 > def route_charge(): ...
34
35 @app.route('/health', methods=['GET'])
36 > def health_check(): ...
37
38 @app.route('/status', methods=['GET'])
39 > def status(): ...
40
41 @app.route('/metrics', methods=['GET'])
42 > def metrics(): ...
43
44 if __name__ == '__main__':
45     load_thread = threading.Thread(target=update_substation_loads, daemon=True)
46     load_thread.start()
```

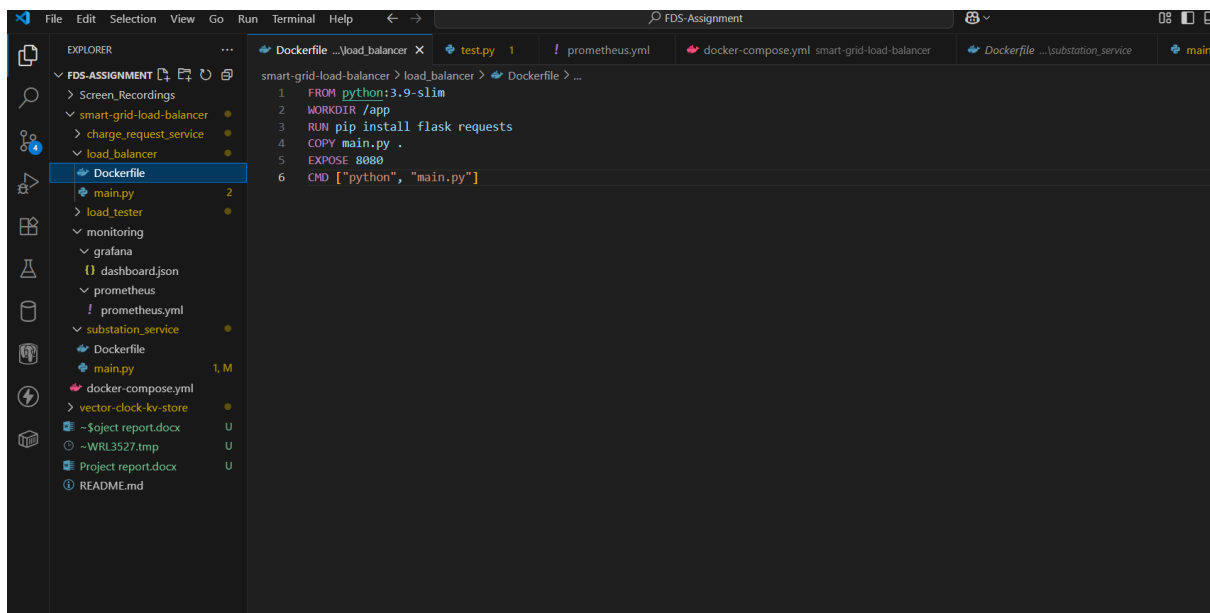
4.6.3 Docker-compose.yml



The screenshot shows the VS Code editor with the `docker-compose.yml` file open. The Explorer sidebar on the left shows the project structure, including `smart-grid-load-balancer`, `charge_request_service`, `load_balancer`, `monitoring`, `substation_service`, and `vector-clock-kv-store`. The `docker-compose.yml` file is selected and highlighted in blue. The main editor area displays the content of the `docker-compose.yml` file, which defines three services: `charge_request_service`, `load_balancer`, and `substation_1`. The `charge_request_service` service is built from the `./charge_request_service` context and depends on the `load_balancer` service. The `load_balancer` service is built from the `./load_balancer` context and depends on the `substation_1` service. The `substation_1` service is built from the `./substation_service` context. All services are part of the `smart-grid` network and are configured to restart unless stopped.

```
1 version: '3.8'
2
3 services:
4   charge_request_service:
5     build:
6       context: ../charge_request_service
7       dockerfile: Dockerfile
8     ports:
9       - "8080:8080"
10    environment:
11      - LOAD_BALANCER_URL=http://load_balancer:8080
12    depends_on:
13      - load_balancer
14    networks:
15      - smart-grid
16    restart: unless-stopped
17
18   load_balancer:
19     build:
20       context: ../load_balancer
21       dockerfile: Dockerfile
22     ports:
23       - "8080:8080"
24     depends_on:
25       - substation_1
26       - substation_2
27       - substation_3
28     networks:
29       - smart-grid
30     restart: unless-stopped
31
32   substation_1:
33     build:
34       context: ../substation_service
```

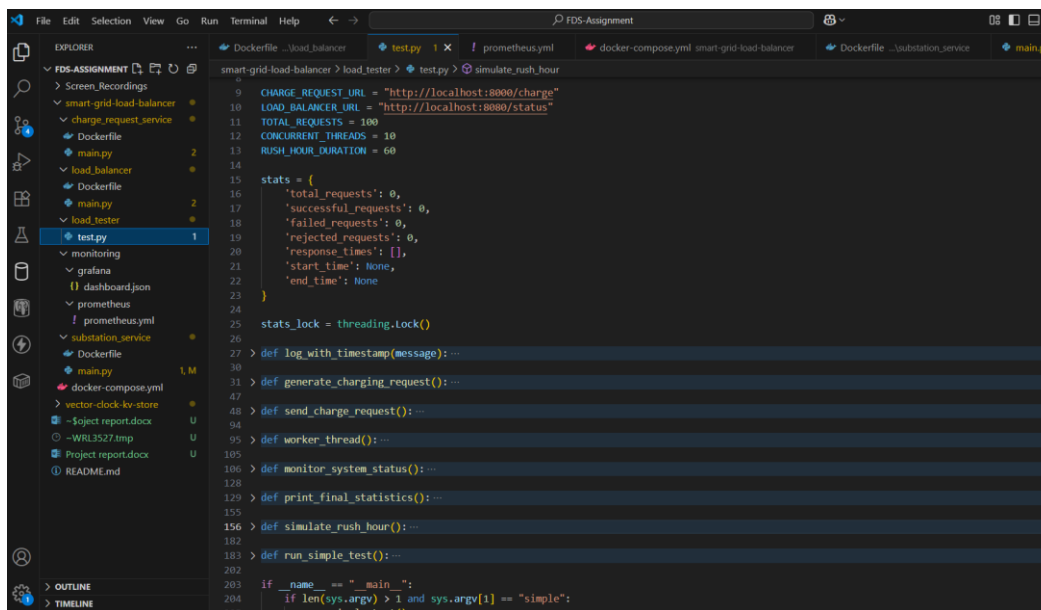
4.6.4 load_balancer/Dockerfile



The screenshot shows the VS Code editor with the `Dockerfile` file for the `load_balancer` service open. The Explorer sidebar on the left shows the project structure, including `smart-grid-load-balancer`, `charge_request_service`, `load_balancer`, `monitoring`, `substation_service`, and `vector-clock-kv-store`. The `Dockerfile` file is selected and highlighted in blue. The main editor area displays the content of the `Dockerfile` file, which defines the build process for the `load_balancer` service. The `Dockerfile` starts with the `FROM python:3.9-slim` base image, sets the `WORKDIR` to `/app`, installs `flask` and `requests` using `pip install flask requests`, copies the `main.py` file to the current directory, exposes port `8080`, and sets the `CMD` to `["python", "main.py"]`.

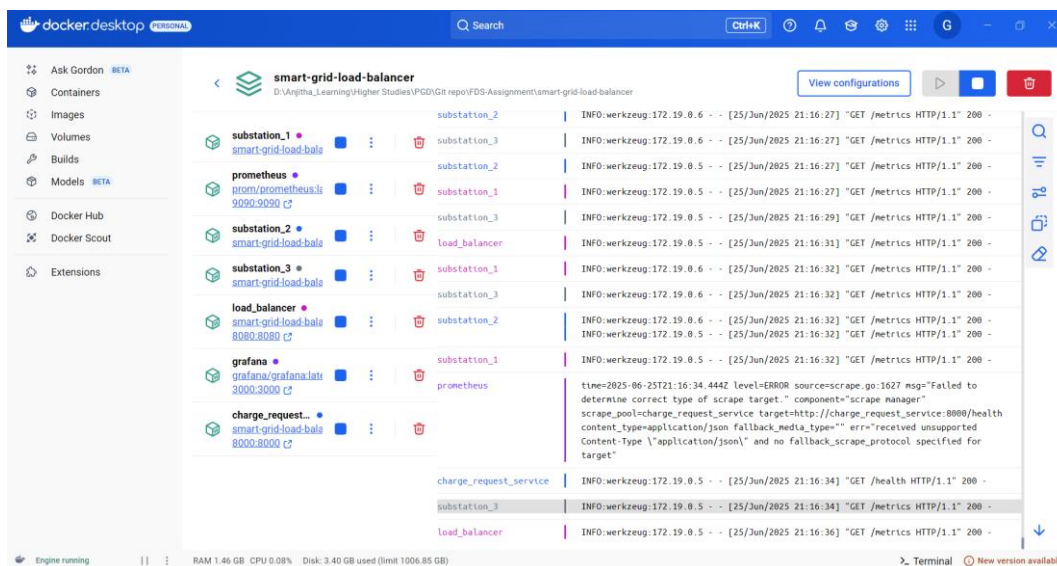
```
1 FROM python:3.9-slim
2 WORKDIR /app
3 RUN pip install flask requests
4 COPY main.py .
5 EXPOSE 8080
6 CMD ["python", "main.py"]
```

4.6.5 load_tester/test.py



```
smart-grid-load-balancer > load_tester > test.py > simulate_rush_hour
9
10 CHARGE_REQUEST_URL = "http://localhost:8080/charge"
11 LOAD_BALANCER_URL = "http://localhost:8080/status"
12 TOTAL_REQUESTS = 100
13 CONCURRENT_THREADS = 10
14 RUSH_HOUR_DURATION = 60
15
16 stats = {
17     'total_requests': 0,
18     'successful_requests': 0,
19     'failed_requests': 0,
20     'rejected_requests': 0,
21     'response_times': [],
22     'start_time': None,
23     'end_time': None
24 }
25
26 stats_lock = threading.Lock()
27
28 > def log_with_timestamp(message): ...
30
31 > def generate_charging_request(): ...
47
48 > def send_charge_request(): ...
94
95 > def worker_thread(): ...
105
106 > def monitor_system_status(): ...
128
129 > def print_final_statistics(): ...
155
156 > def simulate_rush_hour(): ...
182
183 > def run_simple_test(): ...
202
203 if __name__ == "__main__":
204     if len(sys.argv) > 1 and sys.argv[1] == "simple":
205         run_simple_test()
```

4.6.6 Docker Desktop



Demo video is available in github

