# Indian Institute of Technology - Jodhpur

## Fundamentals of Distributed Systems

## Assignment– 1

## Part 1 &2

Student Name: Amruj Sai M M

Roll Number: G24AI2072

Github Link:

Part 1→ https://github.com/Ullivada/Amruj-vector-clock-kv-store/tree/main/vector-clock-kv-store

Part 2 → https://github.com/Ullivada/smart-grid-load-balancer.git

# Tech Stack

- Python 3.11 – Primary programming language for backend logic
- Docker – For containerizing services and simulating distributed nodes
- Docker Compose – For orchestrating multiple services
- Flask – Micro web framework to expose REST APIs
- Requests – To send HTTP messages between services
- Prometheus – For scraping metrics from services
- Grafana – For dashboard visualization of real-time data

**Docker** helps to package and run applications in self-contained units called containers. Think of a container as a complete, isolated package that has everything your application needs: its code, the software it runs on, and all its supporting files. This setup ensures that the application works the same way everywhere, no matter where it deploys, making it consitent and easy to scale.

**Docker Compose** is a tool that simplifies managing applications made of multiple Docker containers. It is using a YAML file to define and run all the services that make up your application together.

# Part 1: Vector Clocks and Causal Consistency

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.

### Vector Clocks and Causal Ordering

Vector clocks are data structures that track the causal relationships between events in distributed systems. Each node maintains a vector of counters representing the number of events observed at each node. These used to detect the order of events and resolve conflicts. Causal ordering ensures that if one event causally affects another, all nodes observe the events in that order. This technique is essential for achievingg eventual consistency and ensuring correctness in distributed applications.

**System Architecture**

• The system consists of three distributed nodes, each implemented as a Python Flask service running in a Docker container.

• Each node maintains its own local key-value store and a vector clock to track causality.

• When a node receives a local PUT request, it updates its clock and broadcasts the change (with its vector clock) to the other nodes.

• Nodes receiving updates compare vector clocks to determine if the message can be applied or should be buffered.

• Buffered messages are checked periodically and applied once their causal dependencies are fulfilled.

• All nodes communicate over a shared Docker network using REST APIs

**Steps**

1. Created node.py in python. Used Flask to simulate each node in the distributed system.

2. Each node contains:

- A local key-value store

- A buffer for delayed messages

- A vector clock list [i, j, k]

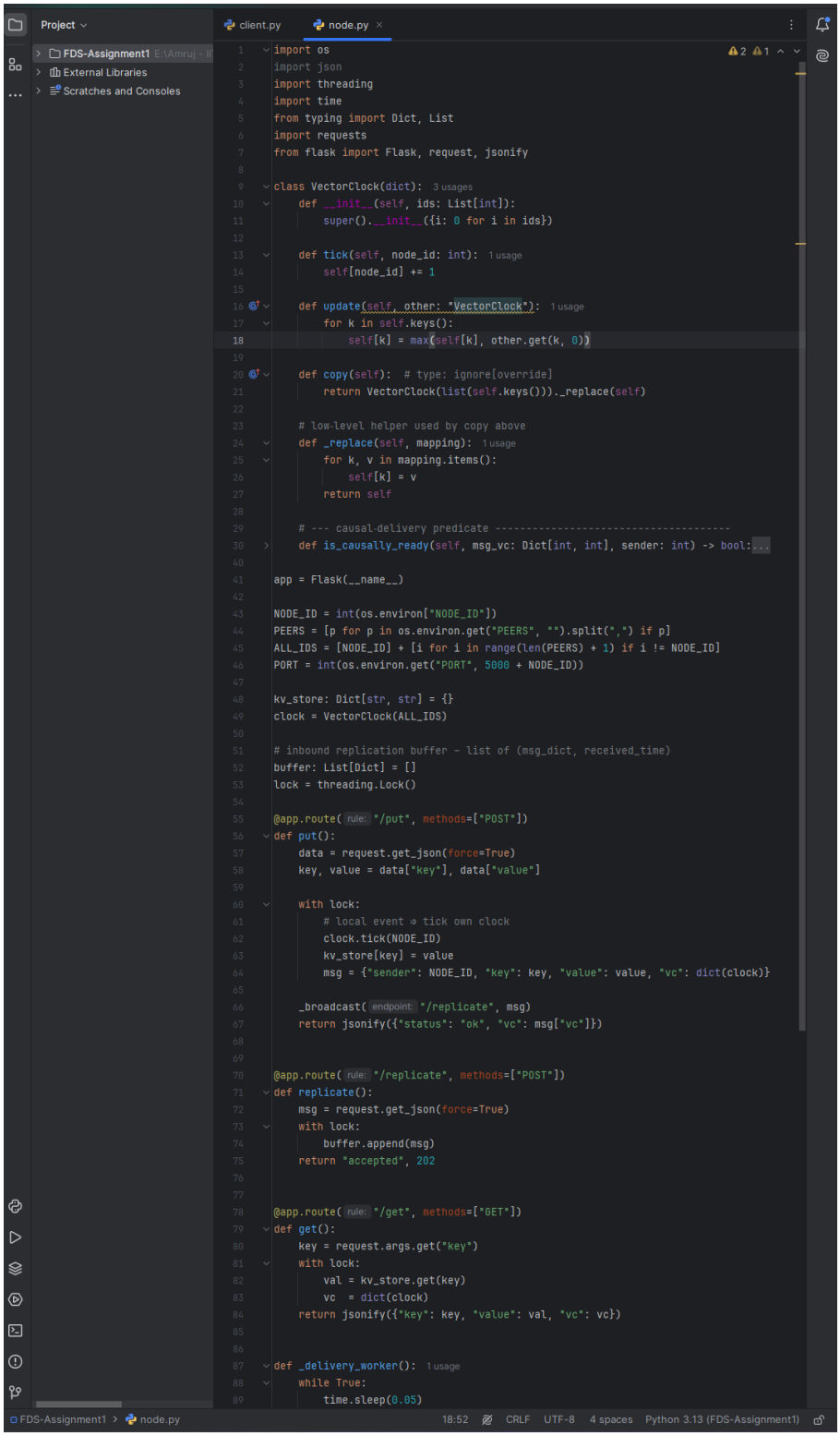
3. The local PUT triggers the node to increment its clock and sends the update + clock to other nodes.

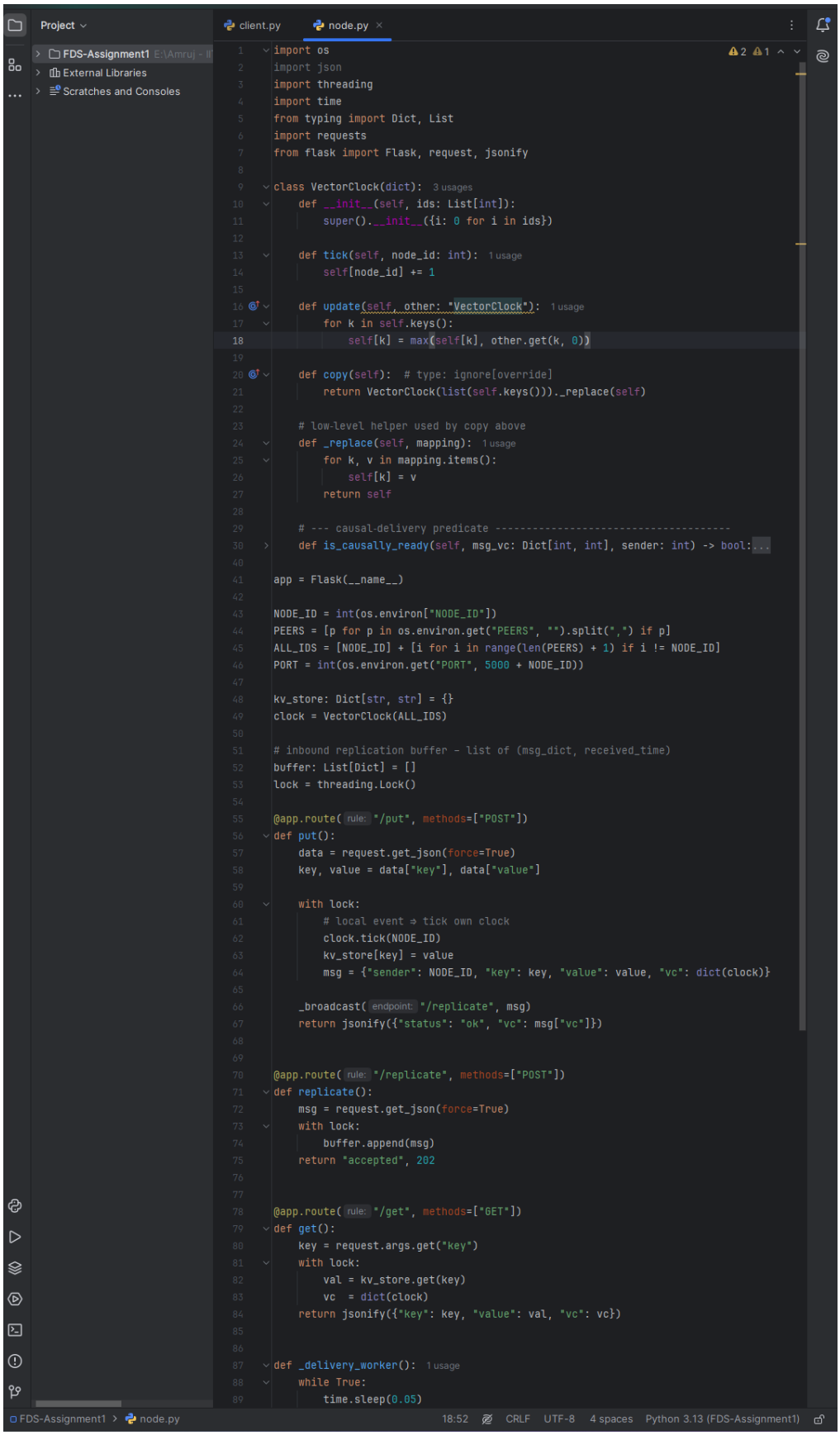
4. Upon receiving a replicated message, the node checks the causal delivery condition. If not met, it buffers the message.

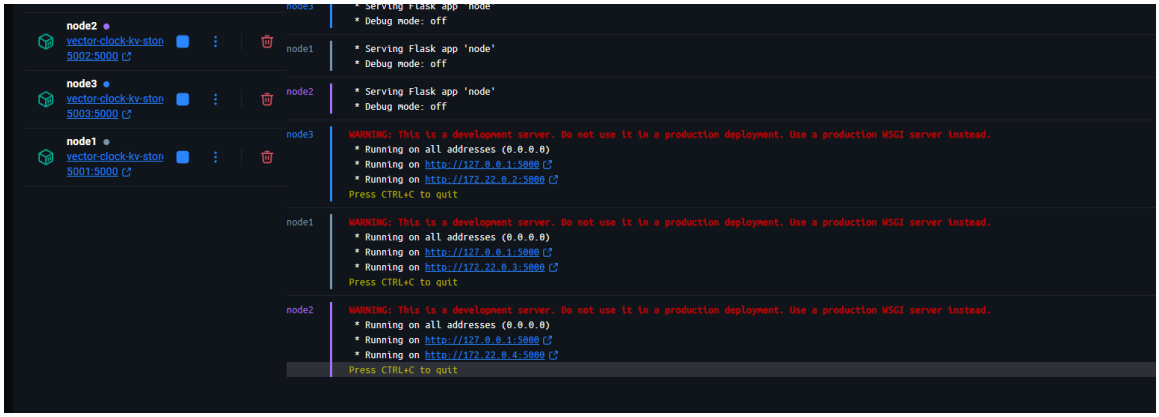
5. Buffered messages are periodically checked for deliverability.

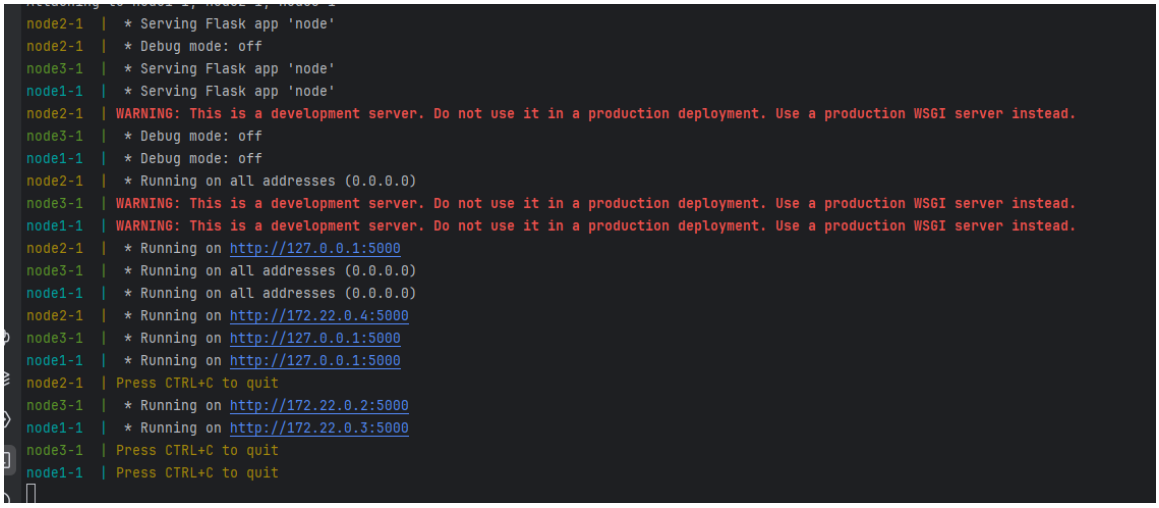
6. Docker Compose launches 3 containers for each node.

# Node.py

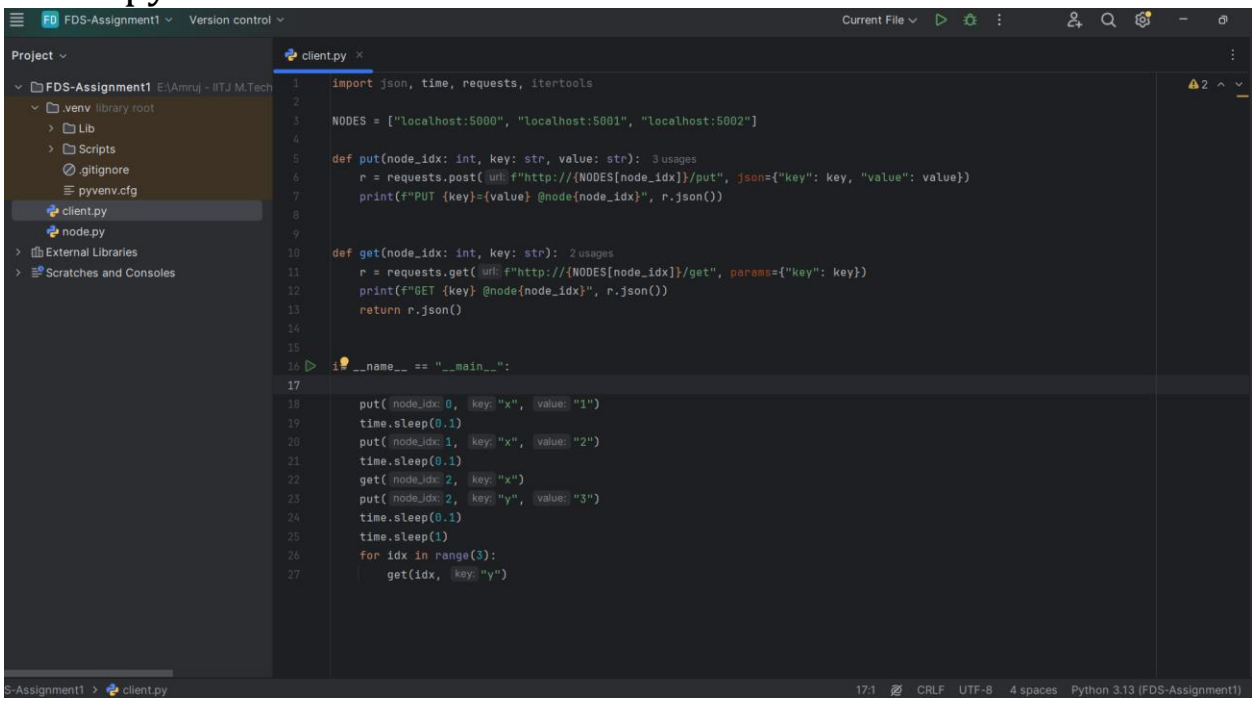

Project
FDS-Assignment1 E:\Amruj - I
External Libraries
Scratches and Consoles
client.py
node.py
import os
import json
import threading
import time
from typing import Dict, List
import requests
from flask import Flask, request, jsonify
class VectorClock(dict): 3 usages
def __init__(self, ids: List[int]):
super().__init__({i: 0 for i in ids})
def tick(self, node_id: int): 1 usage
self[node_id] += 1
def update(self, other: "VectorClock"): 1 usage
for k in self.keys():
self[k] = max(self[k], other.get(k, 0))
def copy(self): # type: ignore[override]
return VectorClock(list(self.keys()))._replace(self)
# low-level helper used by copy above
def _replace(self, mapping): 1 usage
for k, v in mapping.items():
self[k] = v
return self
# --- causal-delivery predicate ------------------------------------
def is_causally_ready(self, msg_vc: Dict[int, int], sender: int) -> bool:...
app = Flask(__name__)
NODE_ID = int(os.environ["NODE_ID"])
PEERS = [p for p in os.environ.get("PEERS", "").split(",") if p]
ALL_IDS = [NODE_ID] + [i for i in range(len(PEERS) + 1) if i != NODE_ID]
PORT = int(os.environ.get("PORT", 5000 + NODE_ID))
kv_store: Dict[str, str] = {}
clock = VectorClock(ALL_IDS)
# inbound replication buffer - list of (msg_dict, received_time)
buffer: List[Dict] = []
lock = threading.Lock()
@app.route( rule: "/put", methods=["POST"])
def put():
data = request.get_json(force=True)
key, value = data["key"], data["value"]
with lock:
# local event ⇒ tick own clock
clock.tick(NODE_ID)
kv_store[key] = value
msg = {"sender": NODE_ID, "key": key, "value": value, "vc": dict(clock)}
_broadcast( endpoint: "/replicate", msg)
return jsonify({"status": "ok", "vc": msg["vc"]})
@app.route( rule: "/replicate", methods=["POST"])
def replicate():
msg = request.get_json(force=True)
with lock:
buffer.append(msg)
return "accepted", 202
@app.route( rule: "/get", methods=["GET"])
def get():
key = request.args.get("key")
with lock:
val = kv_store.get(key)
vc = dict(clock)
return jsonify({"key": key, "value": val, "vc": vc})
def _delivery_worker(): 1 usage
while True:
time.sleep(0.05)
FDS-Assignment1 > node.py
18:52 CRLF UTF-8 4 spaces Python 3.13 (FDS-Assignment1)


Project
FDS-Assignment1 E:\Amruj - II
External Libraries
Scratches and Consoles
client.py
node.py
import os
import json
import threading
import time
from typing import Dict, List
import requests
from flask import Flask, request, jsonify
class VectorClock(dict): 3 usages
def __init__(self, ids: List[int]):
super().__init__({i: 0 for i in ids})
def tick(self, node_id: int): 1 usage
self[node_id] += 1
def update(self, other: "VectorClock"): 1 usage
for k in self.keys():
self[k] = max(self[k], other.get(k, 0))
def copy(self): # type: ignore[override]
return VectorClock(list(self.keys()))._replace(self)
# low-level helper used by copy above
def _replace(self, mapping): 1 usage
for k, v in mapping.items():
self[k] = v
return self
# --- causal-delivery predicate ------------------------------------
def is_causally_ready(self, msg_vc: Dict[int, int], sender: int) -> bool:...
app = Flask(__name__)
NODE_ID = int(os.environ["NODE_ID"])
PEERS = [p for p in os.environ.get("PEERS", "").split(",") if p]
ALL_IDS = [NODE_ID] + [i for i in range(len(PEERS) + 1) if i != NODE_ID]
PORT = int(os.environ.get("PORT", 5000 + NODE_ID))
kv_store: Dict[str, str] = {}
clock = VectorClock(ALL_IDS)
# inbound replication buffer - list of (msg_dict, received_time)
buffer: List[Dict] = []
lock = threading.Lock()
@app.route( rule: "/put", methods=["POST"])
def put():
data = request.get_json(force=True)
key, value = data["key"], data["value"]
with lock:
# local event ⇒ tick own clock
clock.tick(NODE_ID)
kv_store[key] = value
msg = {"sender": NODE_ID, "key": key, "value": value, "vc": dict(clock)}
_broadcast( endpoint: "/replicate", msg)
return jsonify({"status": "ok", "vc": msg["vc"]})
@app.route( rule: "/replicate", methods=["POST"])
def replicate():
msg = request.get_json(force=True)
with lock:
buffer.append(msg)
return "accepted", 202
@app.route( rule: "/get", methods=["GET"])
def get():
key = request.args.get("key")
with lock:
val = kv_store.get(key)
vc = dict(clock)
return jsonify({"key": key, "value": val, "vc": vc})
def _delivery_worker(): 1 usage
while True:
time.sleep(0.05)
FDS-Assignment1 > node.py
18:52 CRLF UTF-8 4 spaces Python 3.13 (FDS-Assignment1)


node2
vector-clock-kv-stor
5002:5000
node3
vector-clock-kv-stor
5003:5000
node1
vector-clock-kv-stor
5001:5000
* Debug mode: off
node1 * Serving Flask app 'node'
* Debug mode: off
node2 * Serving Flask app 'node'
* Debug mode: off
node3 WARNING: This is a development server. Do not use it in a production deployment. Use a production WSGI server instead.
* Running on all addresses (0.0.0.0)
* Running on http://127.0.0.1:5000
* Running on http://172.22.0.2:5000
Press CTRL+C to quit
node1 WARNING: This is a development server. Do not use it in a production deployment. Use a production WSGI server instead.
* Running on all addresses (0.0.0.0)
* Running on http://127.0.0.1:5000
* Running on http://172.22.0.3:5000
Press CTRL+C to quit
node2 WARNING: This is a development server. Do not use it in a production deployment. Use a production WSGI server instead.
* Running on all addresses (0.0.0.0)
* Running on http://127.0.0.1:5000
* Running on http://172.22.0.4:5000
Press CTRL+C to quit


node2-1 | * Serving Flask app 'node'
node2-1 | * Debug mode: off
node3-1 | * Serving Flask app 'node'
node1-1 | * Serving Flask app 'node'
node2-1 | WARNING: This is a development server. Do not use it in a production deployment. Use a production WSGI server instead.
node3-1 | * Debug mode: off
node1-1 | * Debug mode: off
node2-1 | * Running on all addresses (0.0.0.0)
node3-1 | WARNING: This is a development server. Do not use it in a production deployment. Use a production WSGI server instead.
node1-1 | WARNING: This is a development server. Do not use it in a production deployment. Use a production WSGI server instead.
node2-1 | * Running on http://127.0.0.1:5000
node3-1 | * Running on all addresses (0.0.0.0)
node1-1 | * Running on all addresses (0.0.0.0)
node2-1 | * Running on http://172.22.0.4:5000
node3-1 | * Running on http://127.0.0.1:5000
node1-1 | * Running on http://127.0.0.1:5000
node2-1 | Press CTRL+C to quit
node3-1 | * Running on http://172.22.0.2:5000
node1-1 | * Running on http://172.22.0.3:5000
node3-1 | Press CTRL+C to quit
node1-1 | Press CTRL+C to quit

Client.py




E:\Amruj - IITJ M.Tech\T2\FDS\vector-clock-kv-store

# Part 2: Smart Grid Load Balancer

**Summary**

This project implements a dynamic load balancing system for a Smart Grid that efficiently distributes Electric Vehicle (EV) charging requests across multiple substations. The system uses real-time load monitoring, intelligent request routing, and comprehensive observability to ensure optimal resource utilization and grid stability.The implemented solution successfully demonstrates:

- **Dynamic Load Balancing**: Intelligent routing based on real-time substation loads
- **Scalable Architecture**: Microservices-based design with containerized deployment
- **Comprehensive Monitoring**: Full observability stack with Prometheus and Grafana
- **High Availability**: Fault-tolerant design with graceful degradation

**System Architecture**

• The system consists of two substation services, a centralized load balancer, and monitoring tools (Prometheus and Grafana).

• Substations accept EV charging requests and expose their current load via a /metrics endpoint.

• The load balancer periodically polls these metrics and routes new requests to the least-loaded substation.

• Prometheus scrapes the metrics from substations and stores time-series data.

• Grafana connects to Prometheus and visualizes load data through a custom dashboard.

**Steps**

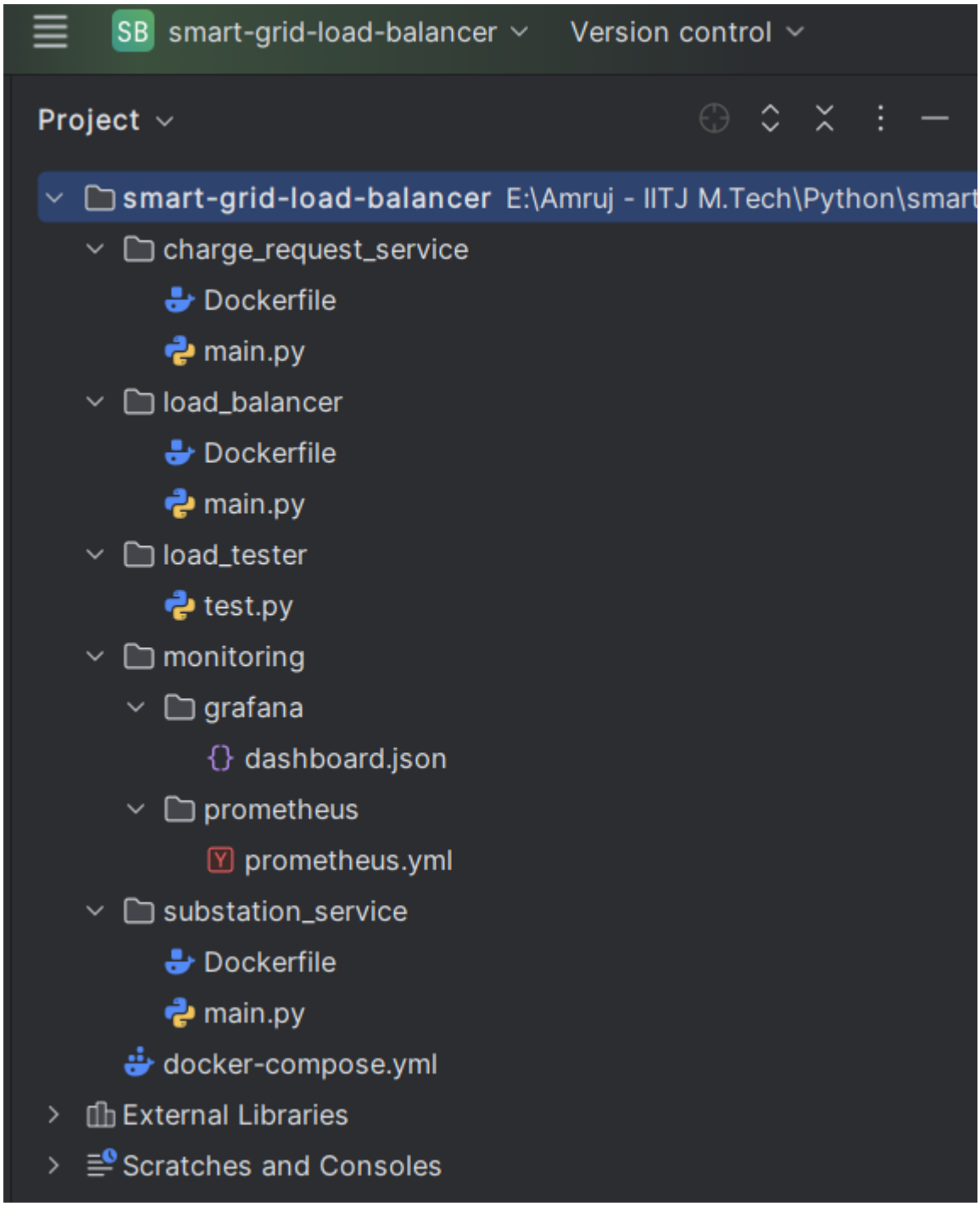
1. Implemented substation_service/main.py with /charge and /metrics endpoints.

2. Implemented load_balancer/main.py to query all substation metrics and route based on lowest load.

3. Created test.py to simulate 50 vehicle requests.

4. Configured Prometheus to scrape metrics from substations.

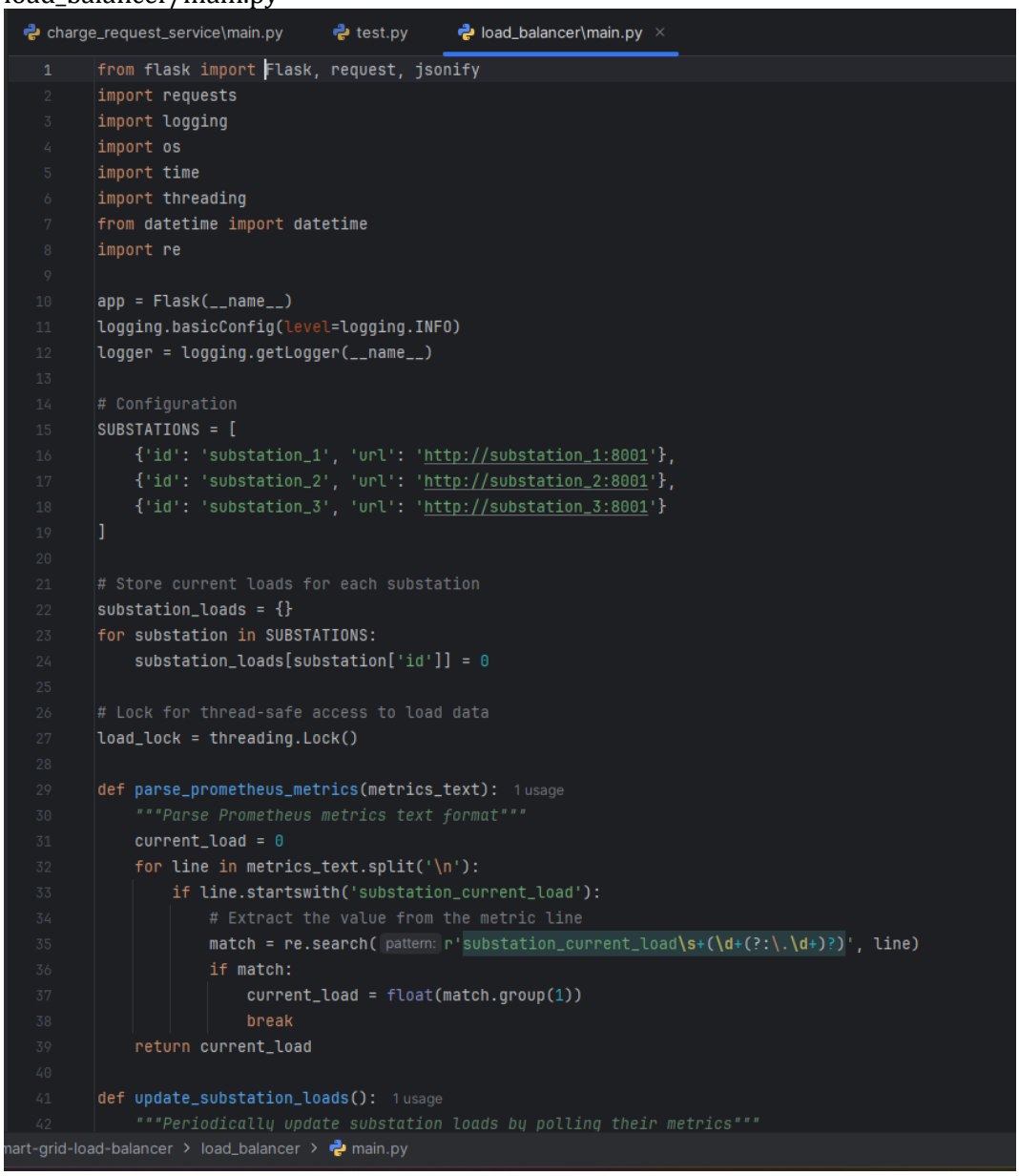
5. Connected Grafana to Prometheus and imported dashboard.


SB
smart-grid-load-balancer
Version control
Project
smart-grid-load-balancer E:\Amruj - IITJ M.Tech\Python\smart
charge_request_service
Dockerfile
main.py
load_balancer
Dockerfile
main.py
load_tester
test.py
monitoring
grafana
dashboard.json
prometheus
prometheus.yml
substation_service
Dockerfile
main.py
docker-compose.yml
External Libraries
Scratches and Consoles

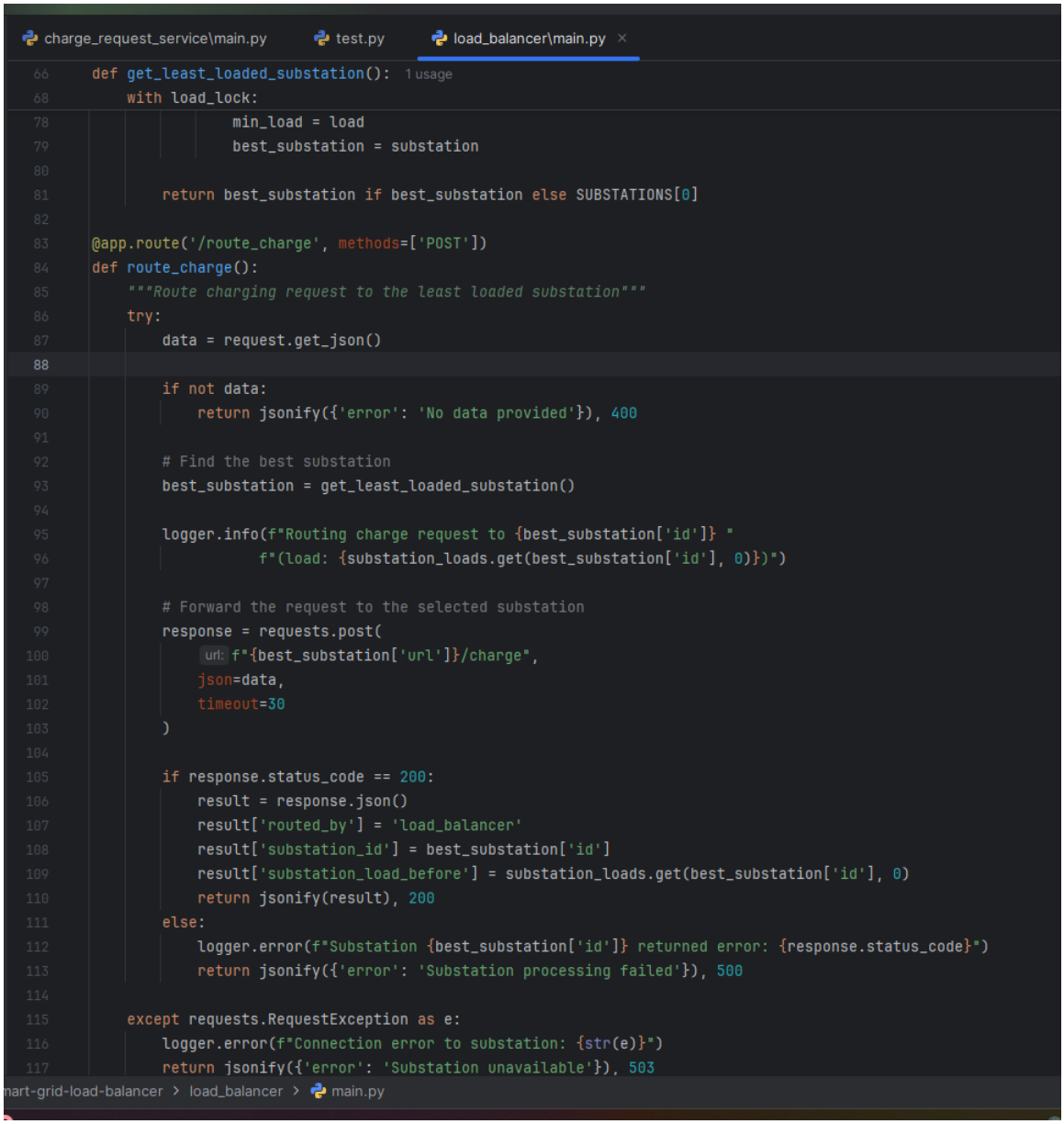
charge_request_service/main.py

```python
from flask import Flask, request, jsonify
import requests
import logging
import os
from datetime import datetime

app = Flask(__name__)
logging.basicConfig(level=logging.INFO)
logger = logging.getLogger(__name__)

LOAD_BALANCER_URL = os.getenv('LOAD_BALANCER_URL', 'http://load_balancer:8080')

@app.route(rule: '/charge', methods=['POST'])
def charge_request():
    """
    Handles incoming EV charging requests and forwards them to the load balancer
    """
    try:
        # Extract charging request data
        data = request.get_json()

        if not data:
            return jsonify({'error': 'No data provided'}), 400

        # Validate required fields
        required_fields = ['vehicle_id', 'charge_amount', 'priority']
        for field in required_fields:
            if field not in data:
                return jsonify({'error': f'Missing required field: {field}'}), 400

        # Add timestamp
        data['timestamp'] = datetime.now().isoformat()

        logger.info(f"Received charge request for vehicle {data['vehicle_id']}")

        # Forward request to load balancer
        response = requests.post(
            url: f"{LOAD_BALANCER_URL}/route_charge",
            json=data,
            timeout=30
        )

        if response.status_code == 200:
            result = response.json()
            logger.info(f"Charge request routed successfully to {result.get('substation_id')}")
            return jsonify(result), 200
        else:
            logger.error(f"Load balancer error: {response.status_code}")
            return jsonify({'error': 'Failed to route charge request'}), 500

    except requests.RequestException as e:
        logger.error(f"Connection error to load balancer: {str(e)}")
        return jsonify({'error': 'Load balancer unavailable'}), 503
    except Exception as e:
        logger.error(f"Unexpected error: {str(e)}")
        return jsonify({'error': 'Internal server error'}), 500

@app.route(rule: '/health', methods=['GET'])
def health_check():
    """Health check endpoint"""
    return jsonify({'status': 'healthy', 'service': 'charge_request_service'}), 200

@app.route(rule: '/status', methods=['GET'])
def status():
    """Service status endpoint"""
    try:
        # Check if load balancer is reachable
        response = requests.get(url: f"{LOAD_BALANCER_URL}/health", timeout=5)
        lb_status = "healthy" if response.status_code == 200 else "unhealthy"
    except:
        lb_status = "unreachable"

    return jsonify({
        'service': 'charge_request_service',
        'status': 'running',
        'load_balancer_status': lb_status,
        'timestamp': datetime.now().isoformat()
    }), 200

if __name__ == '__main__':
```

load_balancer/main.py




charge_request_service\main.py
test.py
load_balancer\main.py
def update_substation_loads(): 1 usage
"""Periodically update substation loads by polling their metrics"""
while True:
try:
for substation in SUBSTATIONS:
try:
response = requests.get( url: f"{substation['url']}/metrics", timeout=5)
if response.status_code == 200:
current_load = parse_prometheus_metrics(response.text)
with load_lock:
substation_loads[substation['id']] = current_load
logger.debug(f"Updated {substation['id']} load: {current_load}")
else:
logger.warning(f"Failed to get metrics from {substation['id']}")
except requests.RequestException as e:
logger.error(f"Error polling {substation['id']}: {str(e)}")
# Keep the last known load value
# Wait before next polling cycle
time.sleep(5) # Poll every 5 seconds
except Exception as e:
logger.error(f"Error in load update thread: {str(e)}")
time.sleep(10) # Wait longer on error
def get_least_loaded_substation(): 1 usage
"""Find the substation with the lowest current load"""
with load_lock:
if not substation_loads:
return SUBSTATIONS[0] # Default to first substation
min_load = float('inf')
best_substation = None
for substation in SUBSTATIONS:
load = substation_loads.get(substation['id'], 0)
if load < min_load:
min_load = load
best_substation = substation
return best_substation if best_substation else SUBSTATIONS[0]
rt-grid-load-balancer > load_balancer > main.py


charge_request_service\main.py
test.py
load_balancer\main.py
def get_least_loaded_substation(): 1 usage
with load_lock:
min_load = load
best_substation = substation
return best_substation if best_substation else SUBSTATIONS[0]
@app.route('/route_charge', methods=['POST'])
def route_charge():
"""Route charging request to the least loaded substation"""
try:
data = request.get_json()
if not data:
return jsonify({'error': 'No data provided'}), 400
# Find the best substation
best_substation = get_least_loaded_substation()
logger.info(f"Routing charge request to {best_substation['id']} "
f"(load: {substation_loads.get(best_substation['id'], 0)})")
# Forward the request to the selected substation
response = requests.post(
url: f"{best_substation['url']}/charge",
json=data,
timeout=30
)
if response.status_code == 200:
result = response.json()
result['routed_by'] = 'load_balancer'
result['substation_id'] = best_substation['id']
result['substation_load_before'] = substation_loads.get(best_substation['id'], 0)
return jsonify(result), 200
else:
logger.error(f"Substation {best_substation['id']} returned error: {response.status_code}")
return jsonify({'error': 'Substation processing failed'}), 500
except requests.RequestException as e:
logger.error(f"Connection error to substation: {str(e)}")
return jsonify({'error': 'Substation unavailable'}), 503
mart-grid-load-balancer > load_balancer > main.py

Load tester/test.py

```
import requests
import json
import time
import random
import threading
from datetime import datetime, timedelta
import sys

# Configuration
CHARGE_REQUEST_URL = "http://localhost:8000/charge"
LOAD_BALANCER_URL = "http://localhost:8080/status"
TOTAL_REQUESTS = 100
CONCURRENT_THREADS = 10
RUSH_HOUR_DURATION = 60  # seconds

# Statistics tracking
stats = {
    'total_requests': 0,
    'successful_requests': 0,
    'failed_requests': 0,
    'rejected_requests': 0,
    'response_times': [],
    'start_time': None,
    'end_time': None
}

stats_lock = threading.Lock()

def log_with_timestamp(message):
    """Print message with timestamp"""
    print(f"[{datetime.now().strftime('%H:%M:%S')}] {message}")

def generate_charging_request():
    """Generate a realistic EV charging request"""
    vehicle_id = f"EV_{random.randint(1000, 9999)}"

    # Realistic charge amounts (in kW)
    charge_amounts = [7, 11, 22, 50, 100, 150]  # Common EV charging
rates
    charge_amount = random.choice(charge_amounts)

    # Priority distribution (most requests are normal priority)
    priority_choices = ['low', 'normal', 'normal', 'normal', 'high']
    priority = random.choice(priority_choices)

    return {
        'vehicle_id': vehicle_id,
        'charge_amount': charge_amount,
        'priority': priority,
        'request_time': datetime.now().isoformat()
    }

def send_charge_request():
    """Send a single charge request and track statistics"""
    global stats
```

```
    try:
        request_data = generate_charging_request()
        start_time = time.time()

        response = requests.post(
            CHARGE_REQUEST_URL,
            json=request_data,
            timeout=30
        )

        end_time = time.time()
        response_time = end_time - start_time

        with stats_lock:
            stats['total_requests'] += 1
            stats['response_times'].append(response_time)

            if response.status_code == 200:
                stats['successful_requests'] += 1
                result = response.json()
                log_with_timestamp(f"✓ Vehicle
{request_data['vehicle_id']} charged at "
                                   f"{result.get('substation_id',
'unknown')} "
                                   f"({request_data['charge_amount']}kW,
{response_time:.2f}s)")
            elif response.status_code == 503:
                stats['rejected_requests'] += 1
                log_with_timestamp(f"⚠ Vehicle
{request_data['vehicle_id']} rejected - "
                                   f"insufficient capacity
({response_time:.2f}s)")
            else:
                stats['failed_requests'] += 1
                log_with_timestamp(f"✗ Vehicle
{request_data['vehicle_id']} failed - "
                                   f"HTTP {response.status_code}
({response_time:.2f}s)")

    except requests.RequestException as e:
        with stats_lock:
            stats['total_requests'] += 1
            stats['failed_requests'] += 1
        log_with_timestamp(f"✗ Request failed: {str(e)}")
    except Exception as e:
        with stats_lock:
            stats['total_requests'] += 1
            stats['failed_requests'] += 1
        log_with_timestamp(f"✗ Unexpected error: {str(e)}")

def worker_thread():
    """Worker thread that continuously sends requests during rush
hour"""
    while True:
        current_time = datetime.now()
```

```
        if stats['end_time'] and current_time >= stats['end_time']:
            break

        send_charge_request()

        # Random delay between requests (0.1 to 2 seconds)
        time.sleep(random.uniform(0.1, 2.0))

def monitor_system_status():
    """Monitor and log system status during load testing"""
    log_with_timestamp("Starting system monitoring...")

    while True:
        current_time = datetime.now()
        if stats['end_time'] and current_time >= stats['end_time']:
            break

        try:
            response = requests.get(LOAD_BALANCER_URL, timeout=5)
            if response.status_code == 200:
                data = response.json()
                loads = data.get('substation_loads', {})
                load_info = ', '.join([f"{k}: {v}" for k, v in
loads.items()])
                log_with_timestamp(f"📊 Substation loads: {load_info}")
            else:
                log with timestamp("⚠ Failed to get load balancer
status")
        except Exception as e:
            log_with_timestamp(f"⚠ Error monitoring system: {str(e)}")

        time.sleep(10)  # Monitor every 10 seconds

def print_final_statistics():
    """Print comprehensive test results"""
    print("\n" + "="*80)
    print("LOAD TEST RESULTS")
    print("="*80)

    duration = (stats['end_time'] -
stats['start_time']).total_seconds()

    print(f"Test Duration: {duration:.1f} seconds")
    print(f"Total Requests: {stats['total_requests']}")
    print(f"Successful Requests: {stats['successful_requests']}
({stats['successful_requests']/stats['total_requests']*100:.1f}%)")
    print(f"Rejected Requests: {stats['rejected_requests']}
({stats['rejected_requests']/stats['total_requests']*100:.1f}%)")
    print(f"Failed Requests: {stats['failed_requests']}
({stats['failed_requests']/stats['total_requests']*100:.1f}%)")
    print(f"Requests per Second:
{stats['total_requests']/duration:.2f}")

    if stats['response_times']:
        avg_response_time = sum(stats['response_times']) /
len(stats['response_times'])
```

```
        min_response_time = min(stats['response_times'])
        max_response_time = max(stats['response_times'])

        print(f"\nResponse Times:")
        print(f"Average: {avg_response_time:.3f}s")
        print(f"Minimum: {min_response_time:.3f}s")
        print(f"Maximum: {max_response_time:.3f}s")

    print("\n" + "="*80)

def simulate_rush_hour():
    """Simulate a rush hour of EV charging requests"""
    print("🚗 Smart Grid Load Balancer - Rush Hour Simulation")
    print("="*60)

    # Initialize timing
    stats['start_time'] = datetime.now()
    stats['end_time'] = stats['start_time'] +
timedelta(seconds=RUSH_HOUR_DURATION)

    log_with_timestamp(f"Starting {RUSH_HOUR_DURATION}s rush hour
simulation with {CONCURRENT_THREADS} concurrent threads")

    # Start monitoring thread
    monitor_thread = threading.Thread(target=monitor_system_status,
daemon=True)
    monitor_thread.start()

    # Start worker threads
    threads = []
    for i in range(CONCURRENT_THREADS):
        thread = threading.Thread(target=worker_thread, daemon=True)
        thread.start()
        threads.append(thread)
        log_with_timestamp(f"Started worker thread {i+1}")

    # Wait for all threads to complete
    for thread in threads:
        thread.join()

    log_with_timestamp("Rush hour simulation completed!")

    # Print final statistics
    print_final_statistics()

def run_simple_test():
    """Run a simple test with a few requests"""
    print("🔧 Running simple connectivity test...")

    for i in range(5):
        log_with_timestamp(f"Sending test request {i+1}/5")
        request_data = generate_charging_request()

        try:
            response = requests.post(CHARGE_REQUEST_URL,
json=request_data, timeout=10)
```

```
            if response.status_code == 200:
                result = response.json()
                log_with_timestamp(f"✓ Success:
{result.get('substation_id', 'unknown')}")
            else:
                log_with_timestamp(f"⚠ HTTP {response.status_code}:
{response.text}")
        except Exception as e:
            log_with_timestamp(f"✗ Error: {str(e)}")

        time.sleep(2)

if __name__ == "__main__":
    if len(sys.argv) > 1 and sys.argv[1] == "simple":
        run_simple_test()
    else:
        simulate_rush_hour()
```

substation_service/main.py

```
from flask import Flask, request, jsonify
import logging
import os
import time
import threading
import random
from datetime import datetime, timedelta

app = Flask(__name__)
logging.basicConfig(level=logging.INFO)
logger = logging.getLogger(__name__)

# Configuration
SUBSTATION_ID = os.getenv('SUBSTATION_ID', 'substation_unknown')
MAX_CAPACITY = int(os.getenv('MAX_CAPACITY', '100'))  # Maximum
charging capacity
CHARGE_PROCESSING_TIME = int(os.getenv('CHARGE_PROCESSING_TIME', '10'))
# seconds

# Current load tracking
current_load = 0
charging_sessions = {}  # Store active charging sessions
load_lock = threading.Lock()

def simulate_charging_completion():
    """Background thread to simulate charging completion"""
    global current_load, charging_sessions

    while True:
        try:
            current_time = datetime.now()
            completed_sessions = []

            with load_lock:
                for session_id, session_data in
```

```
charging_sessions.items():
                    if current_time >= session_data['end_time']:
                        completed_sessions.append(session_id)
                        current_load -= session_data['charge_amount']
                        logger.info(f"Charging completed for session
{session_id}, "
                                    f"load reduced by
{session_data['charge_amount']}")

                # Remove completed sessions
                for session_id in completed_sessions:
                    del charging_sessions[session_id]

                # Ensure load doesn't go negative
                if current_load < 0:
                    current_load = 0

            time.sleep(2)  # Check every 2 seconds

        except Exception as e:
            logger.error(f"Error in charging completion thread:
{str(e)}")
            time.sleep(5)

@app.route('/charge', methods=['POST'])
def process_charge():
    """Process a charging request"""
    global current_load, charging_sessions

    try:
        data = request.get_json()

        if not data:
            return jsonify({'error': 'No data provided'}), 400

        # Validate required fields
        required_fields = ['vehicle_id', 'charge_amount', 'priority']
        for field in required_fields:
            if field not in data:
                return jsonify({'error': f'Missing required field:
{field}'}), 400

        charge_amount = float(data['charge_amount'])
        vehicle_id = data['vehicle_id']
        priority = data.get('priority', 'normal')

        with load_lock:
            # Check if we can handle this charge request
            if current_load + charge_amount > MAX_CAPACITY:
                logger.warning(f"Charge request rejected - would exceed
capacity "
                               f"(current: {current_load}, requested:
{charge_amount}, max: {MAX_CAPACITY})")
                return jsonify({
                    'error': 'Insufficient capacity',
                    'current_load': current_load,
                    'max_capacity': MAX_CAPACITY,
```

```
                    'available_capacity': MAX_CAPACITY - current_load
                }), 503

            # Accept the charging request
            session_id =
f"{SUBSTATION_ID}_{vehicle_id}_{int(time.time())}"

            # Calculate charging duration based on amount and priority
            base_duration = CHARGE_PROCESSING_TIME
            if priority == 'high':
                duration = max(base_duration * 0.7, 5)  # 30% faster
for high priority
            elif priority == 'low':
                duration = base_duration * 1.3  # 30% slower for low
priority
            else:
                duration = base_duration

            # Add some randomness to simulate real-world variation
            duration = duration * (0.8 + random.random() * 0.4)  # ±20%
variation

            start_time = datetime.now()
            end_time = start_time + timedelta(seconds=duration)

            # Update load and add session
            current_load += charge_amount
            charging_sessions[session_id] = {
                'vehicle_id': vehicle_id,
                'charge_amount': charge_amount,
                'priority': priority,
                'start_time': start_time,
                'end_time': end_time,
                'duration': duration
            }

            logger.info(f"Started charging session {session_id} for
vehicle {vehicle_id}, "
                        f"amount: {charge_amount}, duration:
{duration:.1f}s, "
                        f"new load: {current_load}")

        return jsonify({
            'status': 'accepted',
            'session_id': session_id,
            'substation_id': SUBSTATION_ID,
            'vehicle_id': vehicle_id,
            'charge_amount': charge_amount,
            'estimated_duration': duration,
            'start_time': start_time.isoformat(),
            'current_load': current_load,
            'max_capacity': MAX_CAPACITY
        }), 200

    except ValueError as e:
        return jsonify({'error': f'Invalid charge amount: {str(e)}'}),
400
```

```
    except Exception as e:
        logger.error(f"Unexpected error in charge processing:
{str(e)}")
        return jsonify({'error': 'Internal server error'}), 500

@app.route('/metrics', methods=['GET'])
def metrics():
    """Expose metrics in Prometheus format"""
    with load_lock:
        metrics_text = f"# HELP substation_current_load Current
charging load of the substation\n"
        metrics_text += f"# TYPE substation_current_load gauge\n"
        metrics_text += f"substation_current_load {current_load}\n"

        metrics_text += f"# HELP substation_max_capacity Maximum
capacity of the substation\n"
        metrics_text += f"# TYPE substation_max_capacity gauge\n"
        metrics_text += f"substation_max_capacity {MAX_CAPACITY}\n"

        metrics_text += f"# HELP substation_active_sessions Number of
active charging sessions\n"
        metrics_text += f"# TYPE substation_active_sessions gauge\n"
        metrics_text += f"substation_active_sessions
{len(charging_sessions)}\n"

        utilization = (current_load / MAX_CAPACITY) * 100 if
MAX_CAPACITY > 0 else 0
        metrics_text += f"# HELP substation_utilization_percent
Capacity utilization percentage\n"
        metrics_text += f"# TYPE substation_utilization_percent
gauge\n"
        metrics_text += f"substation_utilization_percent
{utilization:.2f}\n"

    return metrics_text, 200, {'Content-Type': 'text/plain'}

@app.route('/health', methods=['GET'])
def health_check():
    """Health check endpoint"""
    return jsonify({'status': 'healthy', 'substation_id':
SUBSTATION_ID}), 200

@app.route('/status', methods=['GET'])
def status():
    """Get detailed status of the substation"""
    with load_lock:
        return jsonify({
            'substation_id': SUBSTATION_ID,
            'current_load': current_load,
            'max_capacity': MAX_CAPACITY,
            'utilization_percent': (current_load / MAX_CAPACITY) * 100
if MAX_CAPACITY > 0 else 0,
            'active_sessions': len(charging_sessions),
            'available_capacity': MAX_CAPACITY - current_load,
            'sessions': {k: {
                'vehicle_id': v['vehicle_id'],
                'charge_amount': v['charge_amount'],
```

```
                'priority': v['priority'],
                'remaining_time': (v['end_time'] -
datetime.now()).total_seconds()
            } for k, v in charging_sessions.items()},
            'timestamp': datetime.now().isoformat()
        }), 200

if __name__ == '__main__':
    # Start the charging completion simulation thread
    completion_thread =
threading.Thread(target=simulate_charging_completion, daemon=True)
    completion_thread.start()

    logger.info(f"Starting substation {SUBSTATION_ID} with capacity
{MAX_CAPACITY}")
    app.run(host='0.0.0.0', port=8001, debug=False)
```

docker-compose.yml

```
services:
  # Charge Request Service - Public entry point
  charge_request_service:
    build:
      context: ./charge_request_service
      dockerfile: Dockerfile
    ports:
      - "8000:8000"
    environment:
      - LOAD_BALANCER_URL=http://load_balancer:8080
    depends_on:
      - load_balancer
    networks:
      - smart-grid
    restart: unless-stopped

  # Load Balancer - Core routing logic
  load_balancer:
    build:
      context: ./load_balancer
      dockerfile: Dockerfile
    ports:
      - "8080:8080"
    depends_on:
      - substation_1
      - substation_2
      - substation_3
    networks:
      - smart-grid
    restart: unless-stopped

  # Substation Services - Multiple replicas with different capacities
  substation_1:
```

```
    build:
      context: ./substation_service
      dockerfile: Dockerfile
    environment:
      - SUBSTATION_ID=substation_1
      - MAX_CAPACITY=80
      - CHARGE_PROCESSING_TIME=8
    expose:
      - "8001"
    networks:
      - smart-grid
    restart: unless-stopped

  substation_2:
    build:
      context: ./substation_service
      dockerfile: Dockerfile
    environment:
      - SUBSTATION_ID=substation_2
      - MAX_CAPACITY=120
      - CHARGE_PROCESSING_TIME=10
    expose:
      - "8001"
    networks:
      - smart-grid
    restart: unless-stopped

  substation_3:
    build:
      context: ./substation_service
      dockerfile: Dockerfile
    environment:
      - SUBSTATION_ID=substation_3
      - MAX_CAPACITY=100
      - CHARGE_PROCESSING_TIME=12
    expose:
      - "8001"
    networks:
      - smart-grid
    restart: unless-stopped

  # Prometheus - Metrics collection
  prometheus:
    image: prom/prometheus:latest
    ports:
      - "9090:9090"
    volumes:
      -
./monitoring/prometheus/prometheus.yml:/etc/prometheus/prometheus.yml
      - prometheus_data:/prometheus
    command:
      - '--config.file=/etc/prometheus/prometheus.yml'
      - '--storage.tsdb.path=/prometheus'
      - '--web.console.libraries=/etc/prometheus/console_libraries'
      - '--web.console.templates=/etc/prometheus/consoles'
      - '--storage.tsdb.retention.time=200h'
      - '--web.enable-lifecycle'
```

```
    networks:
      - smart-grid
    restart: unless-stopped

  # Grafana - Metrics visualization
  grafana:
    image: grafana/grafana:latest
    ports:
      - "3000:3000"
    volumes:
      - grafana_data:/var/lib/grafana
      - ./monitoring/grafana:/etc/grafana/provisioning
    environment:
      - GF_SECURITY_ADMIN_USER=admin
      - GF_SECURITY_ADMIN_PASSWORD=admin
      - GF_USERS_ALLOW_SIGN_UP=false
    networks:
      - smart-grid
    depends_on:
      - prometheus
    restart: unless-stopped

networks:
  smart-grid:
    driver: bridge

volumes:
  prometheus_data:
  grafana_data:
```

promethus.yml

```
global:
  scrape_interval: 15s
  evaluation_interval: 15s

rule_files:
  # - "first rules.yml"
  # - "second_rules.yml"

scrape_configs:
  # The job name is added as a label `job=<job_name>` to any timeseries
scraped from this config.
  - job_name: 'prometheus'
    static_configs:
      - targets: ['localhost:9090']

  # Scrape substation metrics
  - job_name: 'substations'
    static_configs:
      - targets:
        - 'substation_1:8001'
        - 'substation_2:8001'
        - 'substation_3:8001'
    scrape_interval: 5s
```

```
    metrics_path: '/metrics'

  # Scrape load balancer metrics
  - job_name: 'load_balancer'
    static_configs:
      - targets: ['load_balancer:8080']
    scrape_interval: 5s
    metrics_path: '/metrics'

  # Optional: Monitor the charge request service
  - job_name: 'charge_request_service'
    static_configs:
      - targets: ['charge_request_service:8000']
    scrape_interval: 10s
    metrics_path: '/health'
```