FAST-NUCES KARACHI CAMPUS
(FALL-2025)

(CS-3001) COMPUTER NETWORKS

COURSE PROJECT:
[MICROSERVICES NETWORK MANAGEMENT]

SECTION:
BS-CS-5A

INSTRUCTORS:
DR. FARRUKH SALIM SHAIKH

PROJECT MEMBERS:
23K-0907 Zubair Ahmed
23K-0516 Usman Khalid
23K-0562 Affan Rehman

➢ PROJECT TYPE

2: Cloud-based Networking for Facebook-class webapps/backends

5: Distributed Systems Networking

➢ KEY POINTS

2a - Virtual network configuration (simulated)
2b - Load balancing and scalability
5a - **gRPC** and **REST**-based microservice communication
5b - Scalable secure API Gateways
5e - Fault detection and recovery mechanisms
4c - ChatOps for network management

➢ INTRODUCTION/DESCRIPTION

MicroNet Manager: A Microservices Networking Platform

MicroNet Manager is a comprehensive distributed systems project that demonstrates modern networking concepts through a practical, working implementation. The system features multiple microservices communicating via different protocols, coordinated through an intelligent API gateway with real-time monitoring and management capabilities.

At its core, this project implements **3** specialized microservices:

1. **User Service** - Manages user data with efficient communication protocols
2. **Product Service** - Handles product catalog information with load balancing
3. **Order Service** - Processes and tracks customer orders

These services are unified through an **API Gateway** that provides:

1. **Intelligent request routing** to appropriate services
2. **Real-time health monitoring** with automatic failure detection
3. **Load balancing** across service instances
4. **Protocol translation** between different communication styles

The system features a modern web dashboard that provides live visualization of service health, performance metrics, and system architecture. Additionally, it includes **WebSocket-based ChatOps** for real-time system management through conversational commands.

Built using **Python, FastAPI,** and **gRPC**, this project demonstrates practical implementation of distributed systems principles, fault tolerance mechanisms, and modern API design patterns in a cloud-native architecture.

- NON-TECHNICAL APPROACH
- REAL-WORLD SCENARIO EXAMPLE

IMAGINE A SHOPPING MALL

Think of our project like a modern shopping mall with different specialized shops and a central information desk.

THE MALL SETUP (OUR SYSTEM)

1. The Main Entrance & Information Desk (API Gateway)

This is like the mall's main entrance and information desk

Location: Port 8000 (like saying "Go to Gate 8000")

Job: Directs you to the right shop and keeps track of everything

Special Feature: Has a live chat system where you can ask questions and get instant answers

2. The Customer Service Shop (User Service)

Location: Shop #8001 in the mall

Job: Manages customer information and profiles

Example: When you ask "Who is customer 123?", it tells you their name and email

3. The Electronics Store (Product Service)

Location: Shop #8002 in the mall

Job: Shows product information and prices

Example: When you ask "What's product 1?", it says "It's a Laptop costing $999"

Special Feature: Has two counters (load balancing) - you get directed to whichever counter is free

4. The Order Processing Center (Order Service)

Location: Shop #8003 in the mall

Job: Tracks orders and their status

Example: When you ask "What's order 2?", it says "Processing, amount $59.99"

SCENARIO 1: YOU WANT TO BUY SOMETHING

Normal Process:

1. You enter the mall (go to localhost:8000)
2. You ask the information desk: "I want to see product 1"
3. Information desk checks which electronics counter is free
4. Electronics store tells you: "Laptop - $999"
5. Information desk gives you the answer

What Actually Happens Behind the Scenes:

Your browser (customer) talks to the API Gateway (information desk)

Gateway sends your request to Product Service (electronics store)

Product Service responds with product details

Gateway shows you the final answer

SCENARIO 2: THE LIVE CHAT SYSTEM (WEBSOCKET CHATOPS)

Imagine this: The mall has walkie-talkies that let you talk directly to security control!

You connect your walkie-talkie (click "Connect" in ChatOps)

You say: "status" (check if all shops are open)

Control center immediately replies: "All shops are OPEN and working!"

You say: "fail user" (pretend customer service is closed)

Control center: "Customer service is now CLOSED for maintenance"

SCENARIO 3: WHEN THINGS GO WRONG

What happens when a shop closes unexpectedly?

• Customer asks: "I want to see user 123"

• Information desk checks: "Oh no! Customer service shop is closed!"

• Immediately tells customer: "Sorry, user service is unavailable right now"

• Dashboard shows red "CLOSED" sign on that shop

• Manager gets alert and can reopen it via walkie-talkie

➢ NETWORKING CONCEPTS USED

A) NETWORK & COMMUNICATION TERMS

1. Ports

Technical: Network endpoints where services listen for connections
Simple: Like different shop numbers in a mall (8000, 8001, etc.) - each number leads to a different service

2. Services

Technical: Independent applications that perform specific business functions
Simple: Like different specialized shops in a mall - each does one job really well

3. API Gateway

Technical: Single entry point that routes requests to appropriate microservices
Simple: Like the mall's main information desk - the one place you go for everything

4. WebSocket

Technical: Communication protocol enabling real-time bidirectional data exchange
Simple: Like having walkie-talkies instead of sending letters - instant two-way conversation

5. Load Balancing

Technical: Distributing network traffic across multiple servers to optimize resource use
Simple: Like having multiple cashiers so no one waits too long in line

6. Protocol

TECHNICAL: Set of rules governing how devices communicate over a network
SIMPLE: Like having agreed-upon hand signals so everyone understands each other

7. SERVICE DISCOVERY

TECHNICAL: How services find and communicate with each other

Simple: Like having a phone directory that tells you which number to call for which department

8. CIRCUIT BREAKER PATTERN

Technical: Prevents cascading failures by stopping requests to failing services

Simple: Like automatically turning off electricity to a short-circuiting appliance

9. ASYNCHRONOUS COMMUNICATION

Technical: Sending messages without waiting for immediate response

Simple: Like sending an email instead of making a phone call

B) ARCHITECTURE TERMS

1. Microservices

Technical: Architecture where an application is composed of small independent services
Simple: Like having specialized shops instead of one giant department store - each focuses on one thing

2. Distributed Systems

Technical: Multiple computers working together as a single coherent system
Simple: Like a team of workers in different locations coordinating to complete a project

3. API (Application Programming Interface)

Technical: Set of definitions and protocols for building and integrating software
Simple: Like a restaurant menu - it tells you what you can order and how to order it

4. REST (Representational State Transfer)

Technical: Architectural style for designing networked applications using HTTP
Simple: Like ordering from a menu - you choose what you want from available options

5. gRPC (Google Remote Procedure Call)

Technical: High-performance framework for service-to-service communication
Simple: Like having a direct hotline between departments - fast and efficient

C) PERFORMANCE & RELIABILITY TERMS

1. Health Check

Technical: Regular monitoring to verify services are functioning properly
Simple: Like a doctor's regular check-up to make sure everything is working well

2. Fault Detection

Technical: System capability to identify when components fail or underperform
Simple: Like having sensors that alert when a machine stops working

3. Fault Tolerance

Technical: System's ability to continue operating when some components fail
Simple: Like having backup generators that turn on when power goes out

4. Scalability

Technical: System's capacity to handle growing amounts of work
Simple: Like being able to add more cashiers when the store gets busy

5. Latency

Technical: Time delay in data transmission over the network
Simple: Like the time between asking a question and getting an answer

D) DEVELOPMENT & TOOLS TERMS

1. FastAPI

Technical: Modern Python web framework for building APIs with automatic documentation
Simple: Like a construction kit that helps build web services quickly and neatly

2. Protocol Buffers

Technical: Method of serializing structured data for efficient communication
Simple: Like having a secret code that makes messages smaller and faster to deliver

3. Uvicorn

Technical: Lightning-fast ASGI server for running Python web applications
Simple: Like the engine that powers and runs our web services smoothly

4. Virtual Environment

Technical: Isolated Python environment for project-specific dependencies
Simple: Like having a separate toolbox for each project so tools don't get mixed up

E) MONITORING & MANAGEMENT TERMS

1. Dashboard

Technical: Web interface displaying real-time system metrics and status
Simple: Like a control panel with lights and gauges showing how everything is working

2. ChatOps

Technical: Managing operations and infrastructure through chat interfaces
Simple: Like being able to control a smart home by talking to it

3. Real-time

Technical: Processing and delivering data immediately as it occurs
Simple: Like live TV - you see things happening right as they happen

4. JSON (JavaScript Object Notation)

Technical: Lightweight data-interchange format that's easy for humans to read
Simple: Like writing information in a standard notebook format everyone can understand

F) DATA FLOW TERMS

1. Request

Technical: Message sent by a client to initiate an action on a server
Simple: Like asking a question or making an order

2. Response

Technical: Message sent by a server answering a client's request
Simple: Like getting an answer to your question or receiving your order

3. Client

Technical: Application or device that requests services from servers
Simple: Like a customer in a store - asks for things and receives them

4. Server

Technical: Application or device that provides services to clients
Simple: Like a shop assistant - helps customers and provides what they need

5. Bidirectional Communication

Technical: Data flow that occurs in both directions simultaneously
Simple: Like a telephone conversation - both people can talk and listen at the same time

➢ GRPC VS REST

A) REST: Like Ordering from a Regular Restaurant Menu

How REST Works (The Restaurant Experience)

Imagine you're in a restaurant with a paper menu:

1. You get a menu (API documentation) with all available dishes
2. You tell the waiter exactly what you want using specific codes:
   - "I want **GET** /burgers" → "Bring me a burger"
   - "I want **POST** /orders" → "I'd like to place an order"
   - "I want **PUT** /orders/123" → "I want to update my order #123"
   - "I want **DELETE** /orders/123" → "Cancel my order #123"
3. The waiter writes down your order in a common language everyone understands (like English/JSON)
4. The kitchen prepares your food and the waiter brings it back

B) gRPC: Like a High-Tech Food Assembly Line

How gRPC Works (The Factory Experience). Imagine an automated food factory with specialized machinery:

1. You have a pre-defined contract (protocol) that says exactly how to communicate
2. You send binary instructions (not human-readable text) directly to the machine
3. The machine understands exactly what you want without any translation
4. Everything is super fast and efficient because there's no "waiting for the waiter"

➢ PROJECT STRUCTURE

D:\CN Project\

```
│
├── frontend/
│   ├── index.html          Web dashboard + ChatOps UI
│   ├── style.css           Complete styling
│   └── script.js           WebSocket client + API logic
│
├── api_gateway/
│   ├── main.py             API Gateway + WebSocket ChatOps server
│   └── config.py           Configuration
│
├── user_service/
│   ├── server.py           User Service (REST)
│   └── test_grpc.py        gRPC test (optional)
│
├── product_service/
│   └── server.py           Product Service (REST)
│
├── order_service/
│   └── server.py           Order Service (REST)
│
├── client/
│   └── client.py           Test client
│
└── (Auto-generated files - ignore)
    ├── user_pb2.py
    ├── user_pb2_grpc.py
    ├── user.proto          (Study)
    └── dependency folders
```

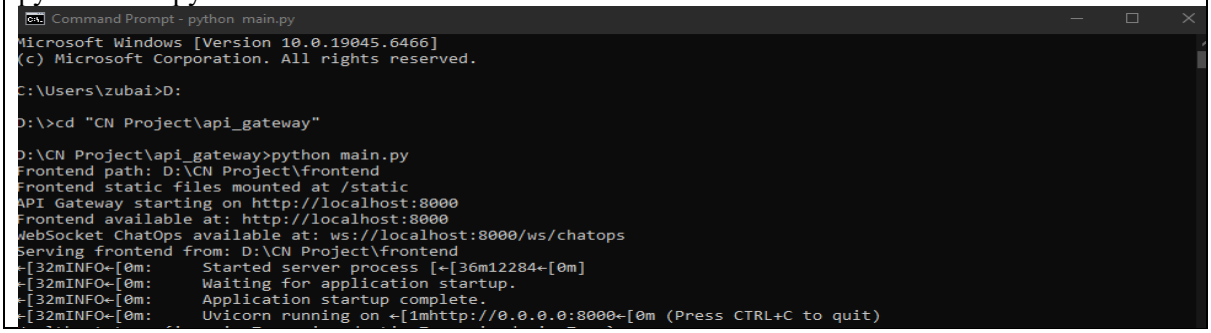
➢ EXECUTING THE PROJECT (OPEN 4 CMD TO RUN MAIN FILES)

D:
cd "CN Project\user_service"
python server.py

D:
cd "CN Project\product_service"
python server.py

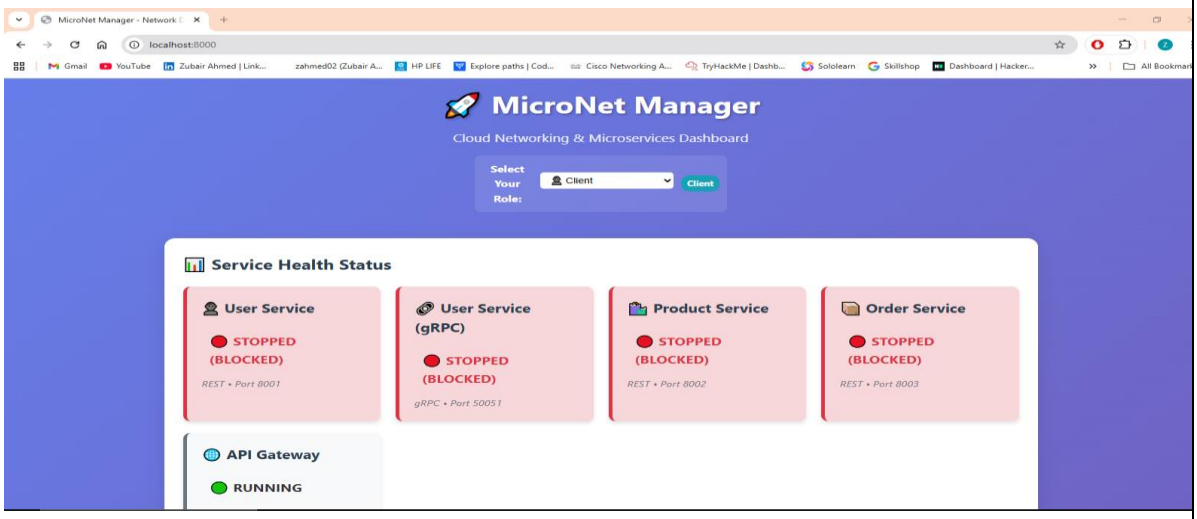
D:
cd "CN Project\order_service"
python server.py

D:
cd "CN Project\api_gateway"
python main.py

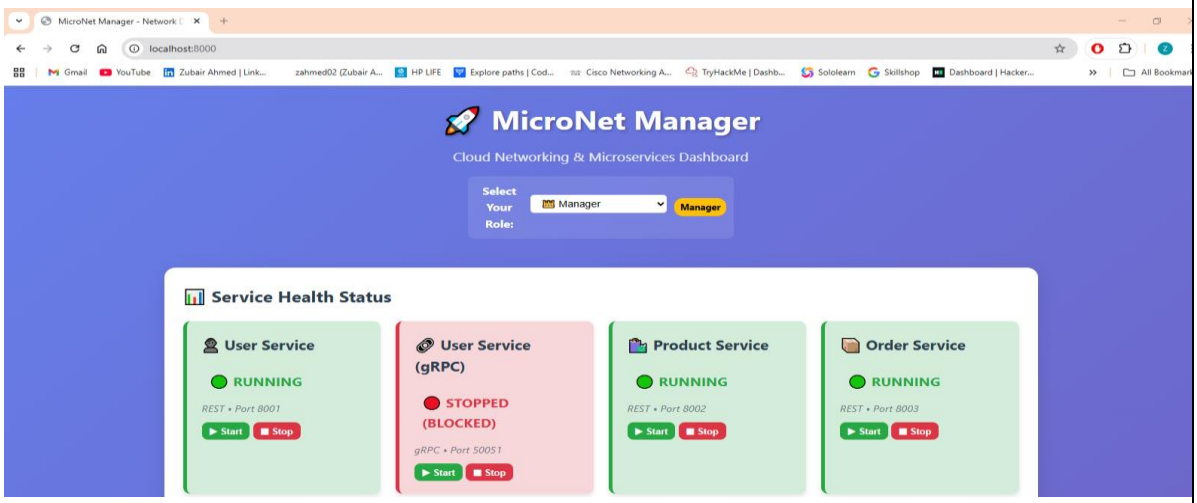


INTERFACE (FRONTEND)

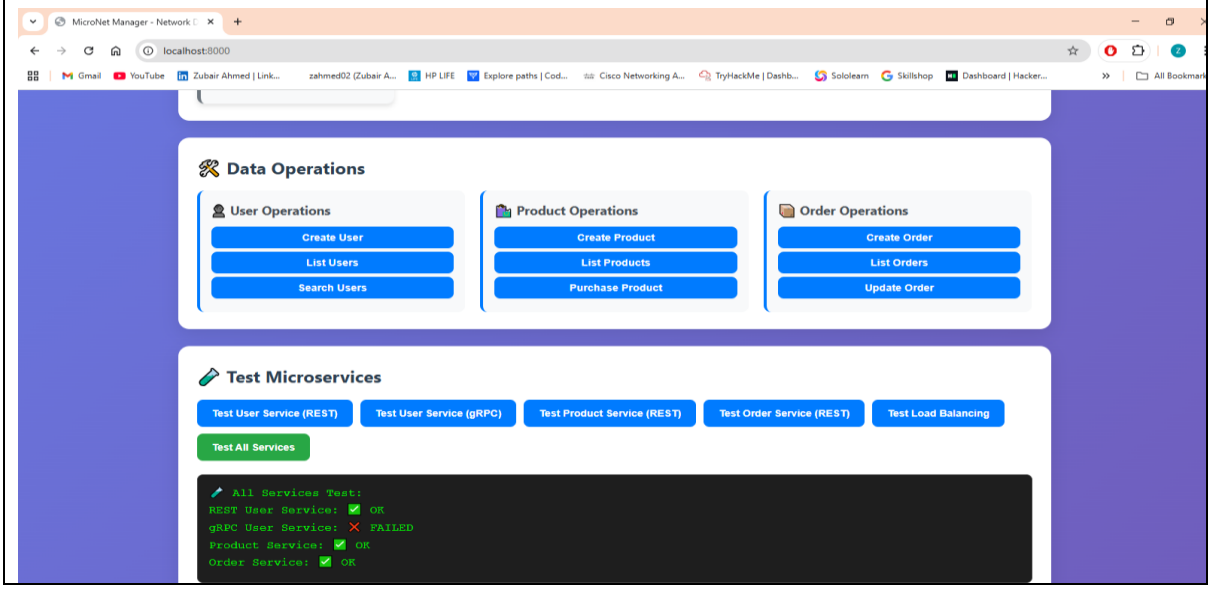
PART-1



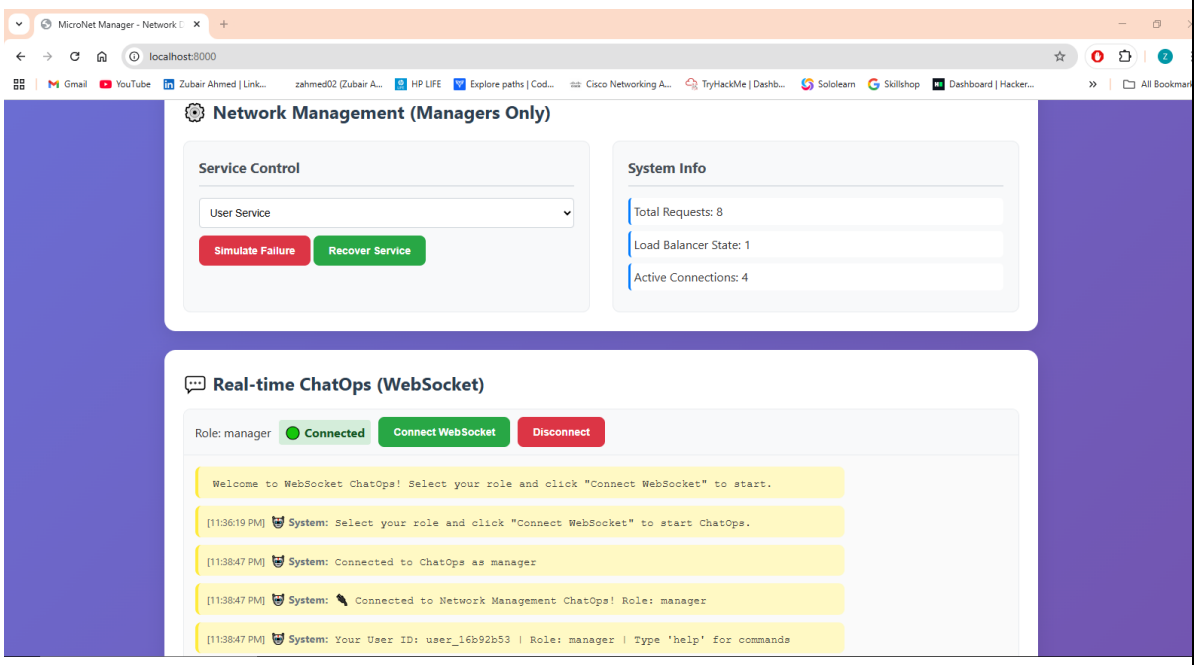
PART-2



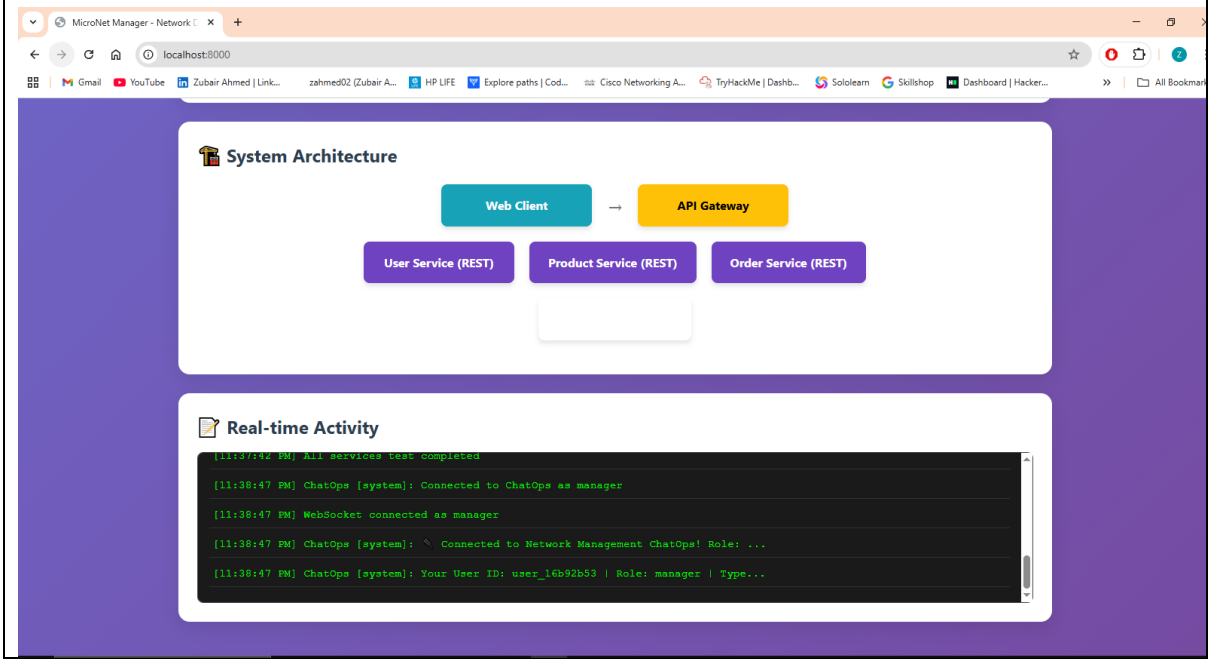
PART-3



PART 4


MicroNet Manager - Network
localhost:8000
Network Management (Managers Only)
Service Control
User Service
Simulate Failure
Recover Service
System Info
Total Requests: 8
Load Balancer State: 1
Active Connections: 4
Real-time ChatOps (WebSocket)
Role: manager
Connected
Connect WebSocket
Disconnect
Welcome to WebSocket ChatOps! Select your role and click "Connect WebSocket" to start.
[11:36:19 PM] System: Select your role and click "Connect WebSocket" to start ChatOps.
[11:38:47 PM] System: Connected to ChatOps as manager
[11:38:47 PM] System: Connected to Network Management ChatOps! Role: manager
[11:38:47 PM] System: Your User ID: user_16b92b53 | Role: manager | Type 'help' for commands

PART 5


MicroNet Manager - Network
localhost:8000
System Architecture
Web Client
API Gateway
User Service (REST)
Product Service (REST)
Order Service (REST)
Real-time Activity
[11:38:47 PM] ChatOps [system]: Connected to ChatOps as manager
[11:38:47 PM] WebSocket connected as manager
[11:38:47 PM] ChatOps [system]: Connected to Network Management ChatOps! Role: ...
[11:38:47 PM] ChatOps [system]: Your User ID: user_16b92b53 | Role: manager | Type...

➢ SYSTEM/NETWORK DESIGN:

CLIENT LAYER (What users see)

└── Web Browser → Localhost:8000

GATEWAY LAYER (The traffic director)

└── API Gateway (Port 8000) → Routes requests + WebSocket Chat

SERVICE LAYER (The specialized workers)

├── User Service (Port 8001) → Manages customer data

├── Product Service (Port 8002) → Handles products with load balancing

└── Order Service (Port 8003) → Processes orders

COMMUNICATION PATHS:

• Browser ↔ Gateway: HTTP/WebSocket

• Gateway ↔ Services: HTTP requests

• Services ↔ Services: Can talk to each other if needed

➢ LEARNING OUTCOMES:

NETWORKING SKILLS:

• How different network protocols work together

• Port management and service communication

• Real-time web communication with WebSocket

APPLICATION/SYSTEM DESIGN:

• Microservices design patterns

• API Gateway design and implementation

• Fault tolerance and system resilience

PRACTICAL PROGRAMMING:

• FastAPI for building web services

• WebSocket programming for real-time features

➢ PROJECT FUNCTIONS/FEATURES

INTELLIGENT ROUTING

• Gateway acts like a smart receptionist - knows where to send each request

• If you ask for users → goes to User Service

• If you ask for products → goes to Product Service

LOAD BALANCING MAGIC

• Product Service has "virtual multiple counters"

• Requests rotate between them automatically

• Prevents one service from getting overwhelmed

FAULT DETECTION SYSTEM

• Gateway constantly checks: "Are you alive?" to all services

• If a service doesn't answer → marks it as unhealthy

• Dashboard immediately shows red status

REAL-TIME CHATOPS

• Like having walkie-talkies with your system

• Type "status" → get instant health report

• Managers can start/stop services via chat commands

ROLE-BASED ACCESS

• Clients: Can only view and test services

• Managers: Have superpowers to control everything

• Prevents accidents from unauthorized users

➢ TESTING AND RESULTS

1. FUNCTIONAL TESTING

(1) We tested if all the microservices worked correctly.

(2) We sent requests to create, read, update, and delete users, products, and orders.

(3) Each service responded correctly and stored the right information.

(4) The system passed all basic functional tests perfectly.

2. API GATEWAY AND LOAD BALANCING

(1) We checked if the API Gateway could route requests properly.

(2) We sent multiple product requests through the gateway.

(3) The gateway successfully directed these requests to different virtual instances.

(4) This proved our load balancing mechanism was working as intended.

3. WEBSOCKET AND REAL-TIME CHATOPS

(1) We tested the real-time chat system for managers.

(2) Commands like 'status' and 'help' received instant replies.

(3) Managers could successfully start and stop services through chat commands.

(4) The WebSocket connection remained stable during all tests.

4. FAULT TOLERANCE AND FAILURE RECOVERY

(1) We manually stopped the User Service to simulate a failure.

(2) The API Gateway instantly detected the service was down.

(3) The dashboard correctly showed a red "stopped" status for the service.

(4) Managers could then restart the service, and the system recovered successfully.

5. OVERALL SYSTEM PERFORMANCE

(1) The system handled all test requests without crashing.

(2) The web dashboard updated in real-time to show service health.

(3) The role-based access control correctly limited features for clients.

(4) In conclusion, all parts of our network design worked together reliably.

➢ IMPLEMENTATION

LIBRARY	PURPOSE	WHERE USED
FASTAPI	Creates web servers and APIs quickly.	All services (user, product, order) and API gateway
UVICORN	Runs the web servers as a production server.	All Python server files to start the applications
PYDANTIC	Validates data and defines request/response formats.	All services for data models like UserCreate, ProductUpdate
WEBSOCKET	Enables real-time, two-way communication for ChatOps.	API gateway and frontend for live management chat
REQUESTS	Allows services to communicate with each other over HTTP.	API gateway to forward requests to microservices
THREADING	Runs background tasks without blocking main operations.	API gateway for health checks while handling requests
SUBPROCESS	Starts and stops service processes programmatically.	API gateway for service management by managers
JSON	Converts data between Python and web-friendly format.	All services for sending and receiving data
UUID	Creates unique identifiers for WebSocket connections.	API gateway to track connected ChatOps users

➢ CONCLUSION

In short, our MicroNet Manager project successfully brings modern networking ideas to life. We built a complete system with three specialized services (User, Product, Order) that all work together, coordinated by a smart central gateway.

The project shows how real cloud systems handle important tasks like directing traffic (load balancing), checking if services are healthy (fault detection), and letting managers control everything through a live chat (ChatOps).

By creating a working dashboard and connecting everything with both REST and WebSocket communication, we didn't just learn theory, we built a real, miniature version of the systems that power today's internet. This project proves we understand how to design, build, and manage the networks that run our digital world.