SRM Institute of Science and Technology

Department of Computer Applications

Delhi – Meerut Road, Sikri Kalan, Ghaziabad, Uttar Pradesh – 201204

Circular – 2020-21

MCA 3rd semester
CLOUD COMPUTING (PCA20D08J)

Lab Manual

Lab 1: Implementation of RPC and Banker's Algorithm

Title: Implementation of Remote Procedure Call (RPC) and Banker's Algorithm

Aim: To implement Remote Procedure Call (RPC) for distributed communication and the Banker's Algorithm for deadlock avoidance in resource allocation.

Procedure:

Part A: Remote Procedure Call (RPC)

1. **Define Interface:** Create an interface definition file (e.g., `.x` file for ONC RPC or a simple Python interface) that specifies the remote procedures, their parameters, and return types.
2. **Generate Stubs:** Use an RPC compiler (if applicable) to generate client-side and server-side stub code from the interface definition.
3. **Implement Server:** Write the server-side logic for the remote procedures. This involves implementing the actual functionality that the client will invoke.
4. **Implement Client:** Write the client-side application that calls the remote procedures as if they were local functions. The client stub handles marshalling parameters and unmarshalling results.
5. **Compile and Run:** Compile both client and server programs. Start the server first, then run the client to test the RPC communication.

Part B: Banker's Algorithm

1. **Data Structures:** Represent the system state using appropriate data structures:
   - `Available`: A 1-D array indicating the number of available resources of each type.
   - `Max`: A 2-D array defining the maximum demand of each process for each resource type.
   - `Allocation`: A 2-D array defining the number of resources of each type currently allocated to each process.
   - `Need`: A 2-D array indicating the remaining resource need of each process (`Need = Max - Allocation`).
2. **Safety Algorithm:** Implement the safety algorithm:
   - Initialize `Work = Available` and `Finish[i] = false` for all processes `i`.
   - Find a process `i` such that `Finish[i] == false` and `Need[i] <= Work`.
   - If such a process is found, update `Work = Work + Allocation[i]` and `Finish[i] = true`. Add `i` to the safe sequence.

- Repeat until no such process can be found.
- If `Finish[i] == true` for all `i`, then the system is in a safe state.

3. **Resource Request Algorithm:** Implement the resource request algorithm for a process `P_k` requesting resources:
   - Check if `Request_k <= Need_k`. If not, the request is invalid.
   - Check if `Request_k <= Available`. If not, `P_k` must wait.
   - If both conditions are met, tentatively allocate resources: `Available = Available - Request_k`, `Allocation_k = Allocation_k + Request_k`, `Need_k = Need_k - Request_k`.
   - Run the safety algorithm. If the system is safe, grant the request. Otherwise, revert the allocation and `P_k` must wait.

Source Code:

```
# Placeholder for Python RPC Server Code
# Example: simple_rpc_server.py
# import xmlrpc.server
#
# def add_numbers(x, y):
#     return x + y
#
# server = xmlrpc.server.SimpleXMLRPCServer(("localhost", 8000))
# server.register_function(add_numbers, "add")
# print("RPC Server listening on port 8000...")
# server.serve_forever()

# Placeholder for Python RPC Client Code
# Example: simple_rpc_client.py
# import xmlrpc.client
#
# proxy = xmlrpc.client.ServerProxy("http://localhost:8000/")
# result = proxy.add(5, 3)
# print(f"Result of remote addition: {result}")

# Placeholder for Python Banker's Algorithm Code
# Example: bankers_algorithm.py
# def is_safe(processes, available, max_needs, allocation):
#     num_processes = len(processes)
#     num_resources = len(available)
#
#     work = list(available)
#     finish = [False] * num_processes
#     safe_sequence = []
#
#     need = [[max_needs[i][j] - allocation[i][j] for j in range(num_resources)]
#             for i in range(num_processes)]
#
#     count = 0
#     while count < num_processes:
#         found = False
#         for p in range(num_processes):
#             if not finish[p]:
#                 if all(need[p][j] <= work[j] for j in range(num_resources)):
#                     work = [work[j] + allocation[p][j] for j in
range(num_resources)]
#                     finish[p] = True
#                     safe_sequence.append(processes[p])
#                     found = True
#                     count += 1
#         if not found:
#             break
#
#     if all(finish):
#         print(f"System is in a safe state. Safe sequence: {safe_sequence}")
```

```
#         return True
#     else:
#         print("System is in an unsafe state.")
#         return False
#
# # Example usage:
# # processes = ["P0", "P1", "P2", "P3", "P4"]
# # available = [3, 3, 2]
# # max_needs = [[7, 5, 3], [3, 2, 2], [9, 0, 2], [2, 2, 2], [4, 3, 3]]
# # allocation = [[0, 1, 0], [2, 0, 0], [3, 0, 2], [2, 1, 1], [0, 0, 2]]
# # is_safe(processes, available, max_needs, allocation)
```

Input:

- **RPC:** Client making a remote procedure call (e.g., `add(5, 3)`).
- **Banker's Algorithm:**
  - Number of processes and resource types.
  - Available resources vector.
  - Maximum need matrix for each process.
  - Allocation matrix for each process.
  - (Optional) A resource request from a specific process.

Expected Output:

- **RPC:** The result of the remote procedure call (e.g., `Result of remote addition: 8`).
- **Banker's Algorithm:**
  - A message indicating whether the system is in a safe state or unsafe state.
  - If safe, the safe sequence of processes.
  - If a resource request is made, whether the request is granted or if the process must wait.

Lab 2: File Sharing in LAN using BitTorrent

Title: Creating and Distributing a Torrent File for LAN File Sharing

Aim: To understand the principles of peer-to-peer file sharing and practically create and distribute a Torrent file to share a file within a Local Area Network (LAN) environment.

Procedure:

1. **Install BitTorrent Client:** Install a BitTorrent client software on your system (e.g., qBittorrent, uTorrent, Transmission).
2. **Prepare File for Sharing:** Choose a file or folder you wish to share on the LAN. Ensure it is easily accessible.
3. **Create New Torrent:**
    - Open your chosen BitTorrent client.
    - Navigate to the option to "Create New Torrent" or similar (usually found under File menu).
    - Add the file(s) or folder(s) you want to share.
    - **Add Tracker:** For LAN sharing, you might use a local tracker (if available) or simply rely on DHT (Distributed Hash Table) and Peer Exchange (PEX) for peer discovery if your client supports it. If setting up a dedicated local tracker is not feasible, ensure DHT and PEX are enabled in your client settings. You can also specify the IP address of another machine on the LAN as a "Web Seed" or "HTTP Source" if direct HTTP download is desired, though this isn't strictly part of the torrent protocol itself.
    - **Save Torrent File:** Choose a location to save the `.torrent` file.
4. **Start Seeding:** Once the `.torrent` file is created, your client will automatically start "seeding" (sharing) the file. Keep the client running.
5. **Distribute Torrent File:** Share the generated `.torrent` file with other users on the LAN (e.g., via email, USB drive, shared network folder).
6. **Download on Another Machine:** On another machine connected to the same LAN, open the `.torrent` file using a BitTorrent client. The client should connect to your seeding machine and start downloading the file.

Source Code:

- N/A (This lab involves the use of existing software applications rather than writing code.)

Input:

- The file(s) or folder(s) to be shared.
- (Optional) Local tracker URL (if a private tracker is set up).

Expected Output:

- A successfully created `.torrent` file.
- The original file being actively seeded by your BitTorrent client.
- The file successfully downloaded by another machine on the LAN using the distributed `.torrent` file.

Lab 3: Demonstration and Assessment of Implemented Algorithms

Title: Demonstration and Assessment of Previously Implemented Algorithms

Aim: To demonstrate the functionality and correctness of the algorithms implemented in previous labs (e.g., RPC and Banker's Algorithm) and to assess their performance and behavior under various conditions.

Procedure:

1. **Preparation:** Ensure that the implemented RPC client/server and Banker's Algorithm programs from Lab 1 are readily available and compiled (if necessary).
2. **Demonstrate RPC:**
    - Start the RPC server.
    - Run the RPC client multiple times with different input parameters for the remote procedure calls.
    - Observe the output on both the client and server sides to confirm successful communication and correct results.
    - Introduce a network delay or disconnect the server briefly to observe error handling (if implemented).
3. **Demonstrate Banker's Algorithm:**
    - Execute the Banker's Algorithm program with various initial system states (available resources, max needs, current allocation).
    - Test cases should include:
        - A state that is clearly safe.
        - A state that is clearly unsafe.
        - A state that becomes unsafe after a resource request.
        - A state that remains safe after a resource request.
    - For each test case, provide the input parameters and observe the output (safe sequence or unsafe state).
4. **Assessment:**
    - **Correctness:** Verify that the algorithms produce the expected results for all test cases.
    - **Robustness:** Check how the algorithms handle invalid inputs or unexpected scenarios (e.g., negative resource values, non-existent processes).
    - **Performance (Qualitative):** Note any noticeable delays or computational overhead, especially for larger inputs (e.g., many processes/resources for Banker's).

Source Code:

- N/A (This lab focuses on demonstrating and assessing existing code, not writing new code.)

Input:

- **RPC:** Various input arguments for the remote procedure calls (e.g., different numbers for addition).
- **Banker's Algorithm:**
    - Multiple sets of `Available`, `Max`, and `Allocation` matrices representing different system states.
    - (Optional) Resource request vectors for different processes.

Expected Output:

- **RPC:** Consistent and correct results from remote procedure calls, demonstrating successful client-server interaction.
- **Banker's Algorithm:** Accurate determination of safe/unsafe states and correct safe sequences for all test cases.
- **Overall:** A qualitative assessment of the algorithms' correctness, robustness, and observed performance.

Lab 4: Utilizing Google Collaboration Tools

Title: Using Google Collaboration Tools: Docs, Sheets, and Slides

Aim: To gain practical experience in creating, editing, and collaboratively sharing documents, spreadsheets, and presentations using Google Docs, Google Sheets, and Google Slides.

Procedure:

1. **Access Google Drive:** Open your web browser and navigate to Google Drive (drive.google.com). Log in with your Google account.
2. **Create Google Docs:**
    - Click on `+ New` -> `Google Docs` -> `Blank document`.
    - Type some sample text, format it (bold, italics, headings), and insert an image.
    - Rename the document (e.g., "My Collaborative Document").
    - Click the `Share` button (top right). Enter the email address of a peer or instructor, set permissions (e.g., "Editor," "Commenter," "Viewer"), and send the invitation.
    - Observe real-time collaboration if a peer joins and edits.
3. **Create Google Sheets:**
    - Click on `+ New` -> `Google Sheets` -> `Blank spreadsheet`.
    - Enter some sample data (e.g., names, numbers).
    - Use a basic formula (e.g., `=SUM(A1:A5)`).
    - Create a simple chart from the data.
    - Rename the spreadsheet (e.g., "My Collaborative Data").
    - Share the spreadsheet with a peer or instructor, similar to Google Docs.
4. **Create Google Slides:**
    - Click on `+ New` -> `Google Slides` -> `Blank presentation`.
    - Add a title slide and a content slide.
    - Insert text, a shape, and an image on the content slide.
    - Apply a simple theme.
    - Rename the presentation (e.g., "My Collaborative Presentation").
    - Share the presentation with a peer or instructor, similar to Google Docs.
5. **Explore Sharing Options:** Experiment with different sharing permissions (e.g., "Anyone with the link," "Restricted") and observe their effects.
6. **Review Version History:** For each document, explore the "Version history" feature to see how changes are tracked and how to revert to previous versions.

Source Code:

- N/A (This lab involves the use of web-based applications, not writing code.)

Input:

- Text, numerical data, and images for documents, sheets, and slides.
- Email addresses of collaborators for sharing.

Expected Output:

- Successfully created Google Docs, Sheets, and Slides with sample content and formatting.
- Documents are correctly shared with specified collaborators, who can access and interact based on their assigned permissions.
- Demonstration of real-time collaboration features (if a peer is available).

- Understanding of version history and different sharing options.

Lab 5: Exploration of Public Cloud Services

Title: Exploration of Major Public Cloud Service Providers

Aim: To explore and understand the fundamental services and offerings provided by prominent public cloud platforms such as Amazon Web Services (AWS), Google Cloud Platform (GCP), Salesforce, and DigitalOcean.

Procedure:

1. **Account Setup (If applicable):** If you don't already have accounts, sign up for free-tier accounts or trial periods for at least two of the following: AWS, GCP, DigitalOcean. (Note: Salesforce is primarily SaaS, so a developer account or trial might be more appropriate for exploration).
2. **Dashboard Navigation:**
    - Log in to the management console/dashboard of each chosen cloud provider.
    - Familiarize yourself with the layout, navigation menus, and search functionalities.
3. **Identify Core Services:** For each cloud provider, identify and briefly explore the following categories of services:
    - **Compute:** Virtual Machines (EC2 on AWS, Compute Engine on GCP, Droplets on DigitalOcean), Serverless Functions (Lambda on AWS, Cloud Functions on GCP).
    - **Storage:** Object Storage (S3 on AWS, Cloud Storage on GCP, Spaces on DigitalOcean), Block Storage (EBS on AWS, Persistent Disk on GCP).
    - **Networking:** Virtual Private Clouds (VPC on AWS, VPC on GCP), Load Balancers, DNS services.
    - **Databases:** Relational Databases (RDS on AWS, Cloud SQL on GCP), NoSQL Databases (DynamoDB on AWS, Firestore/Datastore on GCP).
    - **Developer Tools/Management:** Monitoring, Logging, CI/CD services.
4. **Service Overview:** For each identified service, try to understand:
    - Its primary purpose.
    - Key features.
    - Common use cases.
    - How it integrates with other services.
5. **Cost Exploration (Optional):** Briefly explore the pricing models for a few services (e.g., VM instances, storage) using the provider's pricing calculator or documentation.
6. **Documentation Review:** Browse the official documentation for each provider to understand their offerings in more detail.

Source Code:

- N/A (This lab involves exploration and research of cloud platforms, not writing code.)

Input:

- N/A (The input is your interaction with the cloud provider dashboards and documentation.)

Expected Output:

- A clear understanding of the dashboard layout and navigation for each explored cloud provider.
- Identification and basic comprehension of core services (Compute, Storage, Networking, Databases) offered by each provider.

- Ability to articulate the primary purpose and key features of at least two services from each explored provider.
- Notes or a summary comparing the service offerings and general user experience across different cloud platforms.

Lab 6: Cloud Service and Deployment Models Analysis & Cloud Provider Comparison

Title: Cloud Service and Deployment Models Analysis and Comparative Report of Cloud Service Providers

Aim: To gain a comprehensive understanding of various cloud service models (IaaS, PaaS, SaaS) and deployment models (Public, Private, Hybrid, Community), and to prepare a detailed comparative report on services (VM configuration, cost, network bandwidth, etc.) offered by different Cloud Service Providers.

Procedure:

Part A: Quizzes on Service and Deployment Models

1. **Study Material:** Review concepts related to:
    - **Service Models:** Infrastructure as a Service (IaaS), Platform as a Service (PaaS), Software as a Service (SaaS). Understand their characteristics, responsibilities (who manages what), and examples.
    - **Deployment Models:** Public Cloud, Private Cloud, Hybrid Cloud, Community Cloud. Understand their definitions, advantages, disadvantages, and typical use cases.
2. **Take Quizzes:** Participate in quizzes or self-assessment tests provided by the instructor or found online to evaluate your understanding of these models. Focus on distinguishing between them and identifying appropriate scenarios for each.

Part B: Comparative Report of Cloud Service Providers

1. **Select Providers:** Choose at least two major public cloud providers for comparison (e.g., AWS, Google Cloud Platform, Microsoft Azure).
2. **Research Parameters:** For each chosen provider, research and gather data on the following aspects:
    - **Virtual Machine (VM) Configuration:**
        - Common instance types (e.g., general purpose, compute optimized, memory optimized).
        - Available CPU (vCPUs), RAM, and storage options.
        - Operating System support.
    - **Cost:**
        - Pricing models for VMs (on-demand, reserved instances, spot instances).
        - Storage costs (object storage, block storage).
        - Data transfer (ingress/egress) costs.
        - (If possible) Provide a sample cost calculation for a specific workload.
    - **Network Bandwidth:**
        - Typical network performance for different VM sizes.
        - Available network services (e.g., VPN, Direct Connect/Interconnect).
    - **Other Key Services:** Briefly compare other significant services like databases, serverless functions, and security features.
    - **Strengths and Weaknesses:** Identify the main advantages and disadvantages of each provider.
3. **Report Structure:** Organize your findings into a well-structured report:
    - **Introduction:** Briefly introduce cloud computing and the purpose of the report.

    - **Cloud Models Overview:** Summarize IaaS, PaaS, SaaS, and the deployment models.
    - **Provider Comparison:** Dedicate sections to each provider, detailing your findings. Use tables and charts for clear comparison.
    - **Conclusion:** Summarize the key differences and similarities, and provide recommendations based on different use cases.

Source Code:

- N/A (This lab involves research, analysis, and report writing, not writing code.)

Input:

- N/A (The input is the research data gathered from cloud provider websites, documentation, and pricing calculators.)

Expected Output:

- Successful completion of quizzes with a good understanding of cloud service and deployment models.
- A comprehensive comparative report detailing the configurations, pricing, network capabilities, and other relevant features of selected cloud service providers. The report should highlight their respective strengths and weaknesses.

Lab 7: Development of a Simple Web Service

Title: Creating a Simple Web Service using Python Flask (Client-Server Model)

Aim: To develop a basic web service using Python Flask, demonstrating a client-server interaction model over HTTP.

Procedure:

1. **Environment Setup:**
    - Ensure Python is installed on your system.
    - Install Flask: `pip install Flask`.
2. **Server-Side Implementation (Flask App):**
    - Create a Python file (e.g., `server.py`).
    - Import Flask and create a Flask application instance.
    - Define a route (e.g., `/hello`) that handles GET requests.
    - Inside the route function, return a simple JSON response or a string.
    - Run the Flask application.
3. **Client-Side Implementation:**
    - Create another Python file (e.g., `client.py`).
    - Use the `requests` library (install if needed: `pip install requests`) to make HTTP GET requests to the Flask server's endpoint.
    - Print the response received from the server.
4. **Execution:**
    - Open two separate terminal windows.
    - In the first terminal, run the server: `python server.py`.
    - In the second terminal, run the client: `python client.py`.
    - Observe the output on both terminals.

Source Code:

```
# server.py (Flask Web Service)
from flask import Flask, jsonify, request

app = Flask(__name__)

# Define a simple route that returns a greeting
@app.route('/hello', methods=['GET'])
def hello_world():
    """
    Handles GET requests to the /hello endpoint.
    Returns a JSON response with a greeting message.
    """
    name = request.args.get('name', 'World') # Get 'name' parameter from URL,
default to 'World'
    return jsonify(message=f"Hello, {name}!")

# Define a route for basic arithmetic operation
@app.route('/add', methods=['POST'])
def add_numbers():
    """
    Handles POST requests to the /add endpoint.
    Expects JSON input with 'num1' and 'num2'.
    Returns the sum of the two numbers.
    """
    data = request.get_json()
```

```
    if not data or 'num1' not in data or 'num2' not in data:
        return jsonify(error="Invalid input. Please provide 'num1' and 'num2' in
JSON format."), 400
    try:
        num1 = float(data['num1'])
        num2 = float(data['num2'])
        result = num1 + num2
        return jsonify(sum=result)
    except ValueError:
        return jsonify(error="Invalid number format."), 400

if __name__ == '__main__':
    # Run the Flask app on all available interfaces (0.0.0.0) and port 5000
    # In a production environment, use a WSGI server like Gunicorn or uWSGI
    app.run(host='0.0.0.0', port=5000, debug=True) # debug=True for development,
disable in production
```
```python
# client.py (Web Service Client)
import requests
import json

SERVER_URL = "[http://127.0.0.1:5000](http://127.0.0.1:5000)" # Assuming server
runs locally on port 5000

def test_hello_endpoint(name=""):
    """
    Tests the /hello GET endpoint.
    """
    print(f"\n--- Testing GET /hello (name={name if name else 'default'}) ---")
    url = f"{SERVER_URL}/hello"
    if name:
        url += f"?name={name}"
    try:
        response = requests.get(url)
        response.raise_for_status() # Raise HTTPError for bad responses (4xx or
5xx)
        print("Response Status Code:", response.status_code)
        print("Response JSON:", response.json())
    except requests.exceptions.RequestException as e:
        print(f"Error accessing /hello: {e}")

def test_add_endpoint(num1, num2):
    """
    Tests the /add POST endpoint.
    """
    print(f"\n--- Testing POST /add ({num1}, {num2}) ---")
    url = f"{SERVER_URL}/add"
    payload = {"num1": num1, "num2": num2}
    headers = {"Content-Type": "application/json"}
    try:
        response = requests.post(url, data=json.dumps(payload), headers=headers)
        response.raise_for_status()
        print("Response Status Code:", response.status_code)
        print("Response JSON:", response.json())
    except requests.exceptions.RequestException as e:
        print(f"Error accessing /add: {e}")

if __name__ == '__main__':
    # Test the /hello endpoint
    test_hello_endpoint()
    test_hello_endpoint(name="Alice")

    # Test the /add endpoint
    test_add_endpoint(10, 20)
    test_add_endpoint(5.5, 3.2)
    test_add_endpoint("abc", 10) # Invalid input test
```

Input:

- **Server:** Listens for incoming HTTP requests.
- **Client:** Sends HTTP GET requests to `/hello` (optionally with a `name` query parameter) and HTTP POST requests to `/add` with JSON payload containing `num1` and `num2`.

Expected Output:

- **Server Console:** Shows incoming requests and their status (e.g., `GET /hello 200 OK`).
- **Client Console:**
  - For `/hello` (no name): `Response JSON: {'message': 'Hello, World!'}`
  - For `/hello?name=Alice`: `Response JSON: {'message': 'Hello, Alice!'}`
  - For `/add` (10, 20): `Response JSON: {'sum': 30.0}`
  - For `/add` (5.5, 3.2): `Response JSON: {'sum': 8.7}`
  - For invalid `/add` input: `Response Status Code: 400`, `Response JSON: {'error': 'Invalid input...'}` or `{'error': 'Invalid number format.'}`

Lab 8: Virtual Machine Setup and Chat Application Deployment

Title: Installation of Virtualization Software and Deployment of a Chat Application Across Virtual Machines

Aim: To install and configure Oracle VirtualBox or VMware Workstation, create two virtual machines, and deploy a simple chat application to demonstrate inter-VM communication.

Procedure:

1. **Install Virtualization Software:**
    - Download and install Oracle VirtualBox (free) or VMware Workstation (commercial) on your host operating system. Follow the on-screen instructions.
2. **Create Virtual Machines (VMs):**
    - Launch the virtualization software.
    - Create two new virtual machines.
    - **Operating System:** Install a lightweight Linux distribution (e.g., Ubuntu Server, Debian, or even a minimal desktop version) on both VMs. Ensure both VMs have the same OS for simplicity.
    - **Hardware Configuration:** Allocate sufficient RAM (e.g., 1-2 GB) and CPU cores (e.g., 1-2) to each VM. Allocate enough disk space (e.g., 20 GB).
    - **Network Configuration:** This is crucial for inter-VM communication.
        - **Option 1 (Host-Only Adapter):** Configure both VMs to use a "Host-Only Adapter" network. This creates a virtual network between your host machine and the VMs, and also between the VMs themselves, isolated from your external network. Assign static IP addresses within this network range to each VM.
        - **Option 2 (Internal Network - VirtualBox):** In VirtualBox, you can create an "Internal Network" and connect both VMs to it. This creates a completely isolated network where only the connected VMs can communicate. Assign static IP addresses.
        - **Option 3 (NAT Network - VirtualBox):** If you need VMs to access the internet AND communicate with each other, configure a "NAT Network" (VirtualBox) or a custom NAT segment (VMware) and ensure they are on the same subnet.
        - **Verify Connectivity:** After configuring networking, log into each VM and try to `ping` the other VM's IP address to confirm network connectivity.
3. **Deploy Chat Application:**
    - **Choose a Simple Chat App:** Use a simple client-server chat application (e.g., a Python socket-based chat, or a basic web-based chat from Lab 7 if adapted for direct IP communication).
    - **Transfer Code:** Transfer the chat application code to both VMs (e.g., using SCP, shared folders, or by copying and pasting if the VM has a GUI).
    - **Install Dependencies:** Install any necessary dependencies for the chat application on both VMs (e.g., Python, `requests` library).
    - **Server VM:** On one VM, run the chat application in server mode.
    - **Client VM:** On the other VM, run the chat application in client mode, connecting to the IP address of the server VM.
4. **Test Chat:** Type messages in the client VM and verify that they appear in the server VM's console, and vice-versa if the chat is bidirectional.

Source Code:

```
# chat_server.py (Simple Python Socket Chat Server)
import socket
import threading

HOST = '0.0.0.0' # Listen on all available interfaces
PORT = 12345     # Port for the chat application

clients = []
client_names = {} # To store client names/addresses

def handle_client(conn, addr):
    """Handles communication with a single client."""
    print(f"[NEW CONNECTION] {addr} connected.")
    client_names[conn] = f"Client-{addr[1]}" # Default name using port

    # Ask client for a name
    conn.sendall(b"Enter your name: ")
    try:
        name_data = conn.recv(1024).decode('utf-8').strip()
        if name_data:
            client_names[conn] = name_data
            broadcast(f"{client_names[conn]} has joined the chat!".encode('utf-
8'))
        else:
            broadcast(f"{client_names[conn]} has joined the chat!".encode('utf-
8'))
    except Exception as e:
        print(f"Error getting name from {addr}: {e}")
        broadcast(f"{client_names[conn]} has joined the chat!".encode('utf-8'))

    while True:
        try:
            message = conn.recv(1024)
            if not message:
                break # Client disconnected
            print(f"[{client_names[conn]}] {message.decode('utf-8')}")
            broadcast(f"{client_names[conn]}: ".encode('utf-8') + message, conn)
# Broadcast to others
        except Exception as e:
            print(f"Error handling client {addr}: {e}")
            break

    print(f"[DISCONNECTED] {client_names[conn]} disconnected.")
    clients.remove(conn)
    broadcast(f"{client_names[conn]} has left the chat.".encode('utf-8'))
    conn.close()

def broadcast(message, sender_conn=None):
    """Sends a message to all connected clients except the sender."""
    for client_conn in clients:
        if client_conn != sender_conn:
            try:
                client_conn.sendall(message)
            except Exception as e:
                print(f"Error broadcasting to client: {e}")
                clients.remove(client_conn) # Remove problematic client
                try:
                    client_conn.close()
                except:
                    pass

def start_server():
    """Starts the chat server."""
    server = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
```

```
    server.setsockopt(socket.SOL_SOCKET, socket.SO_REUSEADDR, 1) # Allow reuse
of address
    server.bind((HOST, PORT))
    server.listen(5) # Listen for up to 5 pending connections
    print(f"[LISTENING] Server is listening on {HOST}:{PORT}")

    while True:
        conn, addr = server.accept()
        clients.append(conn)
        thread = threading.Thread(target=handle_client, args=(conn, addr))
        thread.start()
        print(f"[ACTIVE CONNECTIONS] {threading.active_count() - 1}") # -1 for
main thread

if __name__ == '__main__':
    start_server()
```
```python
# chat_client.py (Simple Python Socket Chat Client)
import socket
import threading
import sys

# Replace with the actual IP address of your server VM
SERVER_IP = '192.168.56.101' # Example: Use the IP of your server VM in the
host-only network
SERVER_PORT = 12345

def receive_messages(sock):
    """Receives messages from the server and prints them."""
    while True:
        try:
            data = sock.recv(1024)
            if not data:
                print("\nDisconnected from server.")
                break
            print(data.decode('utf-8'))
        except ConnectionResetError:
            print("\nServer disconnected.")
            break
        except Exception as e:
            print(f"\nError receiving: {e}")
            break

def send_messages(sock):
    """Sends messages from the user to the server."""
    while True:
        try:
            message = input()
            sock.sendall(message.encode('utf-8'))
        except EOFError: # Handles Ctrl+D on Linux/macOS
            print("\nExiting chat.")
            break
        except Exception as e:
            print(f"Error sending: {e}")
            break

def start_client():
    """Starts the chat client."""
    client_socket = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
    try:
        client_socket.connect((SERVER_IP, SERVER_PORT))
        print(f"Connected to chat server at {SERVER_IP}:{SERVER_PORT}")

        # Start a thread to receive messages
        receive_thread = threading.Thread(target=receive_messages,
args=(client_socket,))
```

```
        receive_thread.daemon = True # Allow main thread to exit if receive
thread is running
        receive_thread.start()

        # Main thread for sending messages
        send_messages(client_socket)

    except ConnectionRefusedError:
        print(f"Connection refused. Make sure the server is running at
{SERVER_IP}:{SERVER_PORT}")
    except socket.gaierror:
        print(f"Could not resolve server IP: {SERVER_IP}. Check the IP
address.")
    except Exception as e:
        print(f"An error occurred: {e}")
    finally:
        client_socket.close()
        print("Client socket closed.")
        sys.exit(0) # Ensure client exits cleanly

if __name__ == '__main__':
    start_client()
```

Input:

- **VM Setup:** ISO image of the chosen Linux distribution for installation.
- **Chat Application:** Messages typed by users in the client VM's terminal.

Expected Output:

- Successfully installed and configured Oracle VirtualBox/VMware Workstation.
- Two running virtual machines with network connectivity between them.
- The chat server running on one VM and the chat client running on the other.
- Messages typed on the client VM appearing on the server VM's console, and (if the chat app supports it) messages from the server appearing on the client VM's console.
- Demonstration of successful inter-VM communication.

Lab 9: Web Service Implementation Review and Virtual Environment Testing

Title: Review of Web Service Implementation and Verification in a Virtual Environment

Aim: To review the proper connection establishment and functionality of a previously implemented web service (from Lab 7) and to thoroughly verify its working within a virtualized environment (using VMs from Lab 8).

Procedure:

1. **Prepare Virtual Environment:**
    - Ensure your virtualization software (VirtualBox/VMware) is running.
    - Launch the two virtual machines created in Lab 8.
    - Verify network connectivity between the VMs (e.g., `ping` from one VM to the other).
2. **Deploy Web Service and Client:**
    - **Server VM:** Transfer the `server.py` (Flask web service) code from Lab 7 to one of the virtual machines. Install Flask (`pip install Flask`) if not already present. Run the Flask server on this VM. Note its IP address (e.g., `192.168.56.101`).
    - **Client VM:** Transfer the `client.py` (web service client) code from Lab 7 to the second virtual machine. Install `requests` (`pip install requests`) if not already present. **Crucially, update the `SERVER_URL` in `client.py` to point to the IP address of the server VM.**
3. **Test Connection and Service:**
    - From the client VM, run `python client.py`.
    - Observe the client's output to confirm successful connection and correct responses from the web service.
    - Test both GET (`/hello`) and POST (`/add`) endpoints with various valid and invalid inputs.
4. **Network Monitoring (Optional but Recommended):**
    - On either the host machine or one of the VMs, use a network packet analyzer like Wireshark (if installed and configured to capture traffic on the virtual network interface) to observe the HTTP requests and responses between the client and server VMs. This helps verify that proper HTTP communication is occurring.
5. **Review and Debug:**
    - If the connection fails or the service doesn't work as expected, review:
        - Firewall settings on both VMs (ensure the port the Flask server is listening on is open).
        - IP addresses and network configuration of the VMs.
        - Server logs for errors.
        - Client error messages.
    - Ensure the Flask server is listening on `0.0.0.0` (all interfaces) to be accessible from other VMs.
6. **Documentation:** Document any issues encountered and the steps taken to resolve them.

Source Code:

- N/A (This lab focuses on reviewing and testing existing code in a new environment.)

Input:

- HTTP requests from the client VM to the server VM, including various parameters and payloads.

Expected Output:

- Successful deployment of the web service server on one VM and the client on another.
- Client successfully connecting to the server and receiving correct responses for all web service calls (e.g., "Hello, World!", sum of numbers).
- Demonstration of stable and proper HTTP communication between the client and server within the virtualized environment.
- Identification and resolution of any connectivity or functionality issues.

Lab 10: Cloud Web Application Security Scanning

Title: Scanning Cloud Web Applications for Vulnerabilities using Security Tools

Aim: To utilize specialized security tools such as Acunetix and Ettercap to scan web applications deployed on the cloud for common vulnerabilities and understand their reporting mechanisms.

Procedure:

1. **Set Up Target Web Application (Cloud-hosted):**
    - For ethical hacking purposes, it is **CRUCIAL** to use a web application that you have explicit permission to scan. This could be:
        - A deliberately vulnerable web application (e.g., OWASP Juice Shop, DVWA) deployed on a cloud VM.
        - A test web service you developed and deployed on the cloud.
        - A cloud-based sandbox environment provided for security testing.
    - Note down the public IP address or domain name of this target web application.
2. **Install and Configure Acunetix (or similar DAST tool):**
    - Acunetix is a commercial Dynamic Application Security Testing (DAST) tool. If a license is not available, consider open-source alternatives like OWASP ZAP or Nikto.
    - Install the chosen DAST tool on your local machine or a dedicated VM.
    - Configure the tool to scan the target cloud web application. This typically involves:
        - Entering the target URL.
        - Setting scan profiles (e.g., full scan, SQL injection, XSS).
        - (Optional) Providing authentication credentials if the application requires login.
    - Start the scan and monitor its progress.
3. **Install and Configure Ettercap (for network-level analysis):**
    - Install Ettercap on a Linux machine (e.g., Kali Linux VM) that is on the same network segment as your target cloud application (if possible, or simulate a network attack). Ettercap is primarily for Man-in-the-Middle (MITM) attacks and network sniffing. Its direct use against a remote cloud application might be limited unless you control the network path.
    - **Note:** Ettercap is more suited for local network attacks. For cloud environments, focus on cloud-native network security tools or vulnerability scanners that operate externally. If using Ettercap, it would be to demonstrate network sniffing or ARP poisoning within a controlled virtual network that mirrors a cloud segment.
    - Configure Ettercap to sniff traffic or perform an ARP spoofing attack (in a controlled, isolated lab environment).
4. **Analyze Scan Results:**
    - **DAST Tool (Acunetix/OWASP ZAP):** Review the generated scan report. Identify:
        - Types of vulnerabilities found (e.g., SQL Injection, Cross-Site Scripting (XSS), Broken Authentication, Security Misconfigurations).
        - Severity level of each vulnerability.
        - Proof of concept (if provided).
        - Recommended remediation steps.
    - **Ettercap:** Analyze captured network traffic for sensitive information leakage (e.g., unencrypted credentials, session tokens) or evidence of successful MITM attacks.
5. **Document Findings:** Record the identified vulnerabilities, their impact, and potential remediation strategies.

Source Code:

- N/A (This lab involves using specialized security tools, not writing code.)

Input:

- URL of the target cloud web application.
- (Optional) Authentication credentials for the web application.
- (For Ettercap) Network interface to monitor, target IPs.

Expected Output:

- A detailed vulnerability scan report from Acunetix (or equivalent DAST tool) listing identified web application vulnerabilities, their severity, and remediation advice.
- (If Ettercap is used in a controlled network setup) Captured network traffic demonstrating potential information leakage or successful network attacks.
- A summary of findings regarding the security posture of the cloud web application.

Lab 11: Cloud Network Vulnerability Assessment and Information Leakage Detection

Title: Cloud Network Vulnerability Assessment and Detection of Information Leakage

Aim: To identify and assess vulnerabilities within cloud network configurations and to detect potential leakage of sensitive information to unauthorized third parties within a cloud environment.

Procedure:

1. **Understand Cloud Network Architecture:**
    - Review the network topology of your cloud environment (e.g., AWS VPC, GCP VPC). Understand concepts like Virtual Private Clouds (VPCs), subnets, route tables, security groups/network access control lists (NACLs), public/private IPs, Internet Gateways, NAT Gateways, VPNs.
2. **Identify Potential Vulnerabilities:**
    - **Misconfigured Security Groups/NACLs:** Check for overly permissive inbound rules (e.g., SSH/RDP open to `0.0.0.0/0`), outbound rules that allow unauthorized data exfiltration, or rules that don't enforce least privilege.
    - **Public IP Exposure:** Identify instances or services that are unnecessarily exposed to the public internet.
    - **Insecure Network Services:** Look for unencrypted communication channels or services running on default/weak ports.
    - **Lack of Network Segmentation:** Assess if critical resources are adequately isolated within private subnets.
3. **Use Cloud-Native Security Tools:**
    - **AWS:** Explore AWS Security Hub, Amazon GuardDuty, AWS Config, VPC Flow Logs.
    - **GCP:** Explore Security Command Center, Cloud Audit Logs, VPC Flow Logs.
    - Configure and review logs (e.g., VPC Flow Logs) to monitor network traffic patterns and identify suspicious connections.
4. **Perform Network Scanning (from a controlled environment):**
    - From a controlled VM (e.g., Kali Linux) within your cloud VPC (if allowed) or from an external machine (for public-facing services), use tools like Nmap to scan for open ports and services on your cloud instances.
    - **Caution:** Ensure you have explicit permission to scan these IPs.
5. **Detect Information Leakage:**
    - **Configuration Files:** Check if sensitive information (e.g., API keys, database credentials) is hardcoded in application code or configuration files stored in publicly accessible storage buckets (e.g., S3, Cloud Storage).
    - **Publicly Accessible Logs:** Review if application logs containing sensitive data are publicly exposed.
    - **Outbound Traffic Analysis:** If possible, monitor outbound network traffic from your cloud instances to detect unauthorized data exfiltration attempts.
    - **DNS/Subdomain Enumeration:** Use tools like `sublist3r` or `Amass` to find publicly exposed subdomains that might reveal internal network structures or forgotten services.
6. **Document Findings:** Record all identified network vulnerabilities and any instances of information leakage.

Source Code:

- N/A (This lab involves configuration review, tool usage, and analysis, not writing code.)

Input:

- Cloud account credentials (for accessing dashboards and logs).
- Network configuration details (VPC, subnets, security groups).
- Public IP addresses/domain names of cloud resources.

Expected Output:

- A list of identified cloud network vulnerabilities (e.g., overly permissive security group rules, public exposure of private services).
- Evidence of potential information leakage (e.g., sensitive data in public logs, exposed API keys).
- An understanding of how to use cloud-native tools and external scanners to assess cloud network security.

Lab 12: Vulnerability Assessment Report Generation

Title: Generating a Detailed Vulnerability Assessment Report

Aim: To compile a comprehensive and detailed report describing the vulnerabilities identified in previous labs (specifically Lab 10 and Lab 11), along with suitable and actionable recommendations to remedy these loopholes.

Procedure:

1. **Gather Data:** Collect all findings from Lab 10 (Web Application Scanning) and Lab 11 (Cloud Network Vulnerability Assessment and Information Leakage). This includes:
    - Vulnerability names and descriptions.
    - Severity levels (e.g., Critical, High, Medium, Low, Informational).
    - Impact of the vulnerability (what could happen if exploited).
    - Affected systems/components (e.g., specific web application, VM, network segment).
    - Proof of concept (screenshots, tool output snippets, steps to reproduce).
2. **Report Structure:** Organize the report with the following sections:
    - **Title Page:** Report title, date, author(s), organization.
    - **Table of Contents:** List all sections and sub-sections.
    - **Executive Summary:** A high-level overview for management, summarizing the key findings, overall risk posture, and most critical recommendations.
    - **Introduction:** Purpose of the assessment, scope (what was scanned), and methodology (tools used, approach).
    - **Detailed Findings:** This is the core section. For each vulnerability:
        - **Vulnerability Name:** Clear and concise title.
        - **Description:** Explain what the vulnerability is.
        - **Severity:** Assign a risk level (e.g., CVSS score or qualitative: Critical, High, Medium, Low).
        - **Impact:** Describe the potential consequences if the vulnerability is exploited.
        - **Affected Assets:** List the specific web application, server, or network component affected.
        - **Proof of Concept (PoC):** Provide evidence (screenshots, logs, command outputs) demonstrating the vulnerability.
        - **Recommendation:** Provide clear, actionable steps to remediate the vulnerability. Be specific (e.g., "Update Flask to version X.Y.Z," "Change security group rule to allow port 22 only from IP 1.2.3.4").
    - **Overall Risk Analysis:** Discuss the overall risk profile of the assessed environment.
    - **Conclusion:** Summarize the report and emphasize the importance of addressing the findings.
    - **Appendices (Optional):** Any additional supporting documentation.
3. **Write the Report:** Draft the report, ensuring clarity, conciseness, and professionalism. Use technical terms accurately but explain them where necessary for a broader audience.
4. **Review and Refine:** Proofread the report for grammatical errors, typos, and logical inconsistencies. Ensure all findings are accurately represented and recommendations are practical.

Source Code:

- N/A (This lab involves writing a comprehensive report, not programming.)

Input:

- Findings and data collected from Lab 10 and Lab 11.

Expected Output:

- A well-structured, detailed, and professional vulnerability assessment report that clearly describes identified vulnerabilities, their impact, and provides actionable remediation steps. The report should be suitable for presentation to technical and non-technical stakeholders.

Lab 13: OpenStack All-in-One Installation and Configuration

Title: Installation and Configuration of OpenStack All-in-One using DevStack/Packstack

Aim: To install and configure a single-node, all-in-one OpenStack environment using either DevStack (for development) or Packstack (for RHEL/CentOS-based systems) to gain hands-on experience with deploying a private cloud.

Procedure:

Option A: Using DevStack (Recommended for learning/development)

1. **Prepare Host Machine:**
    - Use a fresh installation of a supported Linux distribution (e.g., Ubuntu Server LTS).
    - Ensure the system has sufficient resources (minimum 8GB RAM, 2 CPU cores, 50GB disk space).
    - Update the system: `sudo apt update && sudo apt upgrade -y`.
    - Install `git`: `sudo apt install git -y`.
2. **Clone DevStack:**
    - Create a non-root user (e.g., `stack`) for DevStack: `sudo useradd -s /bin/bash -d /opt/stack -m stack` and `echo "stack ALL=(ALL) NOPASSWD: ALL" | sudo tee /etc/sudoers.d/stack`.
    - Switch to the `stack` user: `sudo su - stack`.
    - Clone the DevStack repository: `git clone https://opendev.org/openstack/devstack.git /opt/stack/devstack`.
    - Navigate into the devstack directory: `cd /opt/stack/devstack`.
3. **Create `local.conf`:**
    - Create a `local.conf` file in `/opt/stack/devstack` to configure DevStack. This file specifies passwords, enabled services, and other settings.
    - A minimal `local.conf` might look like this:
    - `[[local|localrc]]`
    - `ADMIN_PASSWORD=secretadmin`
    - `DATABASE_PASSWORD=secret_db`
    - `RABBIT_PASSWORD=secret_rabbit`
    - `SERVICE_PASSWORD=secret_service`
    - `HOST_IP=<YOUR_HOST_IP_ADDRESS> # Replace with your server's IP`
    - `# Optional: Enable additional services`
    - `# enable_service horizon`
    - `# enable_service neutron`
    - `# enable_service nova`

4. **Run Stack Script:**
    - Execute the `stack.sh` script: `./stack.sh`.
    - This script will download all necessary packages, install OpenStack components, and configure them. This process can take a significant amount of time (30-60 minutes or more).
5. **Verify Installation:**
    - Upon successful completion, the script will output URLs for the Horizon dashboard and credentials.
    - Log in to the Horizon dashboard using the provided URL and credentials (usually `admin`/`secretadmin`).

- Verify that all core OpenStack services (Nova, Glance, Neutron, Keystone, Cinder) are running.

Option B: Using Packstack (for RHEL/CentOS/Fedora)

1. **Prepare Host Machine:**
   - Use a fresh installation of CentOS/RHEL.
   - Ensure sufficient resources (minimum 8GB RAM, 2 CPU cores, 50GB disk space).
   - Disable SELinux: `sudo setenforce 0` (and modify `/etc/selinux/config` to `SELINUX=permissive`).
   - Disable Firewall: `sudo systemctl stop firewalld && sudo systemctl disable firewalld`.
   - Enable EPEL repository: `sudo yum install epel-release -y`.
   - Install `openstack-packstack`: `sudo yum install -y openstack-packstack`.
2. **Generate Answer File:**
   - Generate a default answer file: `packstack --gen-answer-file=answer.txt`.
3. **Edit Answer File:**
   - Edit `answer.txt` to configure OpenStack. Key parameters to set:
     - `CONFIG_HORIZON_INSTALL=y` (to install dashboard)
     - `CONFIG_NEUTRON_INSTALL=y`
     - `CONFIG_NOVA_INSTALL=y`
     - Set passwords for `CONFIG_KEYSTONE_ADMIN_PW`, `CONFIG_MYSQL_PW`, `CONFIG_RABBITMQ_PW`.
     - `CONFIG_NTP_SERVERS=<your_ntp_server>` (optional)
     - `CONFIG_PROVISION_DEMO=n` (optional, to avoid demo setup)
4. **Run Packstack:**
   - Execute Packstack with the answer file: `packstack --answer-file=answer.txt`.
   - This will install and configure OpenStack.
5. **Verify Installation:**
   - Upon completion, Packstack will provide the Horizon dashboard URL and credentials.
   - Log in to the Horizon dashboard and verify service status.

Source Code:

- N/A (This lab involves using installation scripts and configuration files, not writing application code.)

Input:

- Linux operating system installation media.
- `local.conf` (for DevStack) or `answer.txt` (for Packstack) with configured parameters.

Expected Output:

- A successfully installed and running OpenStack all-in-one environment.
- Ability to access the OpenStack Horizon dashboard via a web browser using the provided URL and credentials.
- Verification that all core OpenStack services are operational.

Lab 14: Launching Virtual Machines in OpenStack

Title: Launching Virtual Machines (Instances) in OpenStack through the Dashboard

Aim: To gain practical experience in launching and managing virtual machines (instances) within an OpenStack cloud environment using its web-based management interface, Horizon.

Procedure:

1. **Access OpenStack Dashboard (Horizon):**
    - Open your web browser and navigate to the Horizon dashboard URL provided after your OpenStack installation (from Lab 13).
    - Log in using your `admin` credentials (or a project user if configured).
2. **Navigate to Instances:**
    - From the left-hand navigation pane, click on `Project` -> `Compute` -> `Instances`.
3. **Launch Instance:**
    - Click the `Launch Instance` button.
    - **Details Tab:**
        - **Instance Name:** Provide a unique name for your VM (e.g., `my-first-vm`).
        - **Availability Zone:** Leave as default or select a specific zone if available.
        - **Count:** Number of instances to launch (start with 1).
    - **Source Tab:**
        - **Image Name:** Select a suitable image (e.g., `cirros` for a tiny test VM, or an Ubuntu/CentOS cloud image if available).
        - **Create New Volume:** No, for simplicity, boot from image.
    - **Flavor Tab:**
        - **Flavor:** Select an appropriate instance size (e.g., `m1.tiny` for Cirros, or a larger one for a full OS). Flavors define vCPUs, RAM, and disk size.
    - **Networks Tab:**
        - **Available Networks:** Drag and drop the available network (e.g., `private` or `demo-net`) to the `Allocated Networks` section. This connects your VM to a virtual network.
    - **Security Groups Tab:**
        - Select a security group that allows necessary inbound traffic (e.g., `default` or one that allows SSH). If no suitable one exists, you might need to create one under `Project` -> `Network` -> `Security Groups` beforehand, allowing SSH (port 22) and ICMP.
    - **Key Pair Tab:**
        - **Key Pair:** Select an existing key pair or create a new one. This is essential for SSH access to your VM. If creating new, remember to download the private key (`.pem` file).
    - **Configuration/Metadata/Server Groups (Optional):** Leave as default for a basic launch.
    - Click `Launch Instance`.
4. **Monitor Instance Status:**
    - Return to the `Instances` page.
    - Monitor the status of your newly launched VM. It will go through states like `Spawning`, `Building`, and finally `Active`.
    - Note down the assigned IP address (both private and public/floating IP if associated).
5. **Associate Floating IP (If necessary for external access):**

    - If your instance only has a private IP and you need to access it from outside the OpenStack network (e.g., from your host machine), you need to associate a floating IP.
    - On the `Instances` page, select your instance, click `Actions` -> `Associate Floating IP`.
    - Allocate a new IP if none are available, then associate it.

Source Code:

- N/A (This lab involves interacting with the OpenStack Horizon dashboard, not writing code.)

Input:

- Selection of VM image, flavor, network, security group, and key pair within the OpenStack dashboard.

Expected Output:

- A successfully launched virtual machine instance with an "Active" status visible in the OpenStack Horizon dashboard.
- The instance having an assigned private IP address.
- (If configured) The instance having an associated public/floating IP address.
- Readiness for SSH access or other network communication.

Lab 15: OpenStack Dashboard Access and Instance Verification

Title: Accessing OpenStack Dashboard and Verifying Instance Functionality

Aim: To successfully access the OpenStack Horizon dashboard through a web browser and to verify the operational status and connectivity of launched virtual machine instances by logging into them via SSH or by pinging them.

Procedure:

1. **Access OpenStack Horizon Dashboard:**
    - Open a web browser (on your host machine or another machine with network access to your OpenStack deployment).
    - Enter the URL for your OpenStack Horizon dashboard (e.g., `http://<your_openstack_ip>/dashboard`).
    - Enter your `admin` username and password (or project user credentials) to log in.
    - Verify that you can navigate through the dashboard, see your project, and view the `Instances` page.
2. **Identify Instance IP Address:**
    - In the Horizon dashboard, go to `Project` -> `Compute` -> `Instances`.
    - Locate the instance you launched in Lab 14.
    - Note down its IP address. This could be a private IP or a public/floating IP. Use the public/floating IP if you are accessing from outside the OpenStack network.
3. **Verify Instance Working (Method 1: Ping):**
    - Open a terminal or command prompt on your host machine (or another machine with network access to the instance's IP).
    - Use the `ping` command to test connectivity to your OpenStack instance's IP address:
    - `ping <instance_ip_address>`
    - Observe if you receive successful replies. If not, troubleshoot network connectivity (e.g., security group rules, firewall on the instance OS, network configuration).
4. **Verify Instance Working (Method 2: SSH Login):**
    - Ensure you have the private key (`.pem` file) associated with the key pair used when launching the instance.
    - Open a terminal or command prompt.
    - Set appropriate permissions for the private key file (e.g., `chmod 400 your_key.pem` on Linux/macOS).
    - Use the `ssh` command to log into the instance. The default username for Cirros is `cirros`, for Ubuntu cloud images it's `ubuntu`, and for CentOS it's `centos`.
    - `ssh -i your_key.pem <username>@<instance_ip_address>`

      Example for Cirros: `ssh -i my_cirros_key.pem cirros@192.168.1.100`
    - If successful, you will get a command prompt for your OpenStack instance.
    - Perform a simple command like `ls` or `pwd` to confirm you are inside the VM.
5. **Troubleshooting:**
    - If `ping` fails: Check security group rules (allow ICMP), network configuration, and host firewall.

- If `ssh` fails: Check security group rules (allow port 22), ensure the correct username and private key are used, verify the instance is fully booted.

Source Code:

- N/A (This lab involves accessing a web interface and using standard command-line tools for verification.)

Input:

- OpenStack Horizon dashboard URL.
- Username and password for Horizon.
- IP address of the launched OpenStack instance.
- Private key file (`.pem`) for SSH access.

Expected Output:

- Successful login to the OpenStack Horizon dashboard.
- Ability to view and manage instances from the dashboard.
- Successful `ping` replies from the OpenStack instance, indicating network reachability.
- Successful SSH login to the OpenStack instance, providing a command-line interface to the VM.