CAH11-01 Networks and Operating Systems
Application Layer in Networks

Aims of the Seminar

Welcome to the Python Workshop on the Application Layer in Networks! This workshop is designed for you to learn about the application layer of the TCP/IP model and how to implement basic network applications using Python. By the end of this workshop, you will have a solid understanding of the application layer, and you will have built a few simple network applications.

We will be using the python socket library. The Python socket library provides a low-level interface for network communication using sockets. Sockets are endpoints for sending and receiving data across a network. The socket library allows you to create different types of sockets, such as TCP sockets for reliable, connection-oriented communication, and UDP sockets for unreliable, connectionless communication.

Core Concepts

- **Sockets**: Think of sockets as communication endpoints. They have an IP address and a port number associated with it.
- **IP Address**: A numerical identifier for a device on a network.
- **Port Number**: A number that identifies a specific application or process on a device.
- **Protocols**: Rules that govern how data is transmitted over a network. Common protocols include TCP (Transmission Control Protocol) and UDP (User Datagram Protocol).

Key Functions

`socket.socket()`: Creates a new socket object. You specify the address family (e.g., IPv4 or IPv6) and the socket type (e.g., TCP or UDP).
`socket.bind()`: Binds a socket to a specific IP address and port number. This is typically done on the server side.
`socket.listen()`: Puts a server socket into listening mode, so it can accept incoming connections.
`socket.accept()`: Accepts a connection from a client. This function returns a new socket object for the connection and the client's address.
`socket.connect()`: Establishes a connection to a remote server. This is done on the client side.
`socket.send()`: Sends data through a socket.
`socket.recv()`: Receives data from a socket.
`socket.close()`: Closes a socket connection.

Prerequisites

- Basic understanding of Python programming.
- Familiarity with basic networking concepts (e.g., IP addresses, ports).

Workshop Outline

1. Introduction to the Application Layer
2. Building a Simple HTTP Client
3. Building a Simple HTTP Server
4. Building a Simple Chat Application
5. Challenges and Exercises

Feel free to discuss your work with peers, or with any member of the teaching staff.

Reminder

We encourage you to discuss the content of the workshop with the delivery team and any findings you gather from the session.

Workshops are not isolated, if you have questions from previous weeks, or lecture content, please come and talk to us.

Exercises herein represent an example of what to do; feel free to expand upon this.

Helpful Resources

The official Python documentation provides detailed information about the socket library: https://docs.python.org/3/library/socket.html

Socket Programming in Python (Guide) https://realpython.com/python-sockets/

A Complete Guide to Socket Programming in Python https://www.datacamp.com/tutorial/a-complete-guide-to-socket-programming-in-python

1. Introduction to the Application Layer

What is the Application Layer?
The application layer is the top layer of the OSI model and is responsible for providing network services directly to end-user applications. Common protocols at this layer include HTTP, FTP, SMTP, and DNS.

Key Concepts:
- Protocols: Rules that define how data is transmitted and received.
- Client-Server Model: A distributed application structure that partitions tasks or workloads between providers (servers) and requesters (clients).
- Sockets: Endpoints for sending and receiving data across a network.

2. Basic Python socket examples

Objectives to introduce python socket and get a hands-on introduction to its functionalities.

a. Finding Website IP address

Let us start with a basic example of finding the IP address of a website. To do so we will use:

```
socket.gethostbyname()
```

Translate a host name to IPv4 address format. The IPv4 address is returned as a string, such as '100.50.200.5'.

Code Example:

```
import socket

def get_ip_address(website_url):
    try:
        ip_address = socket.gethostbyname(website_url)
        print(f"The IP address of {website_url} is {ip_address}")
    except socket.gaierror:
        print(f"Unable to get the IP address for {website_url}")

# Example usage
website = input("Enter the website URL (without 'https://'): ")
get_ip_address(website)
```

This code takes a website URL as input from the user, uses the `socket.gethostbyname()` function to try and find the corresponding IP address, and then prints either the IP address or an error message if the lookup fails. The try-except block is crucial for handling potential errors during the IP address lookup process.

Exercise 1: Try 3 websites and report their IP addresses.

b. Trace Route

Tracert (short for "trace route") is a command-line utility used to trace the network route packets take to reach a specific destination. It's available on Windows systems. On Unix-like systems (macOS, Linux), the equivalent command is traceroute. Both commands serve the same fundamental purpose.

Here's a breakdown of what tracert does and why it's useful:

What it Does:

Tracert works by sending a series of packets to the destination, gradually increasing the Time To Live (TTL) value of each packet. The TTL is a limit on how many "hops" a packet can take before it's discarded.

1. **First Packet (TTL = 1):** The first packet has a TTL of 1. This means it will likely expire after the first hop (the first router it encounters). The router then sends an ICMP (Internet Control Message Protocol) "Time Exceeded" message back to the sender (your computer). tracert records the IP address and round-trip time (RTT) of this router.
2. **Second Packet (TTL = 2):** The second packet has a TTL of 2, so it can travel two hops. It will reach the second router, which will then send a "Time Exceeded" message back. tracert records this router's information.
3. **Subsequent Packets:** This process continues, with the TTL increasing by one for each packet, until the packet reaches its final destination. When the destination is reached, it responds with an ICMP "Echo Reply" message.

Output of tracert:
The output of tracert shows a list of the routers (or "hops") that the packets traversed to reach the destination. For each hop, it typically displays:

- **Hop Number:** The number of the hop in the route.
- **IP Address/Hostname:** The IP address and, if available, the hostname of the router.
- **Round-Trip Time (RTT):** The time it takes for the packet to reach the router and for the "Time Exceeded" message to return. Tracert usually shows three RTT measurements for each hop, as it sends multiple probes.

Code Example: [Please note that it will take about 1-2 min to finish]

```
import subprocess

def tracert(domain):
    try:
        result = subprocess.run(["tracert", domain], capture_output=True, text=True)
        print(result.stdout)
    except FileNotFoundError:
        print("tracert command not found. Make sure it's available.")
    except Exception as e:  # Catching general exceptions for now
        print(f"An error occurred: {e}")

domain = input("Enter the website or IP address: ")
tracert(domain)
```

Exercise 2: Experiment with 3 different domain names and IP addresses to see the trace route information.

Exercise 3: Analyse the output of tracert. It should look like this:

```
Tracing route to google.co.uk [172.217.169.35]
over a maximum of 30 hops:

  1     *        *        *     Request timed out.
  2     4 ms     9 ms    10 ms  vlan3211.fw-cdc.kent.ac.uk [172.31.254.49]
  3     7 ms     5 ms     5 ms  212.219.171.129
  4     4 ms     4 ms     5 ms  10.49.3.205
  5    15 ms     6 ms     6 ms  172.17.251.250
  6    20 ms     4 ms     5 ms  212.219.171.137
  7     8 ms     4 ms     6 ms  ae0.londtw-sbr2.ja.net [146.97.41.85]
  8     6 ms     7 ms     6 ms  ae28.londtt-sbr1.ja.net [146.97.33.61]
  9     8 ms     9 ms     4 ms  72.14.205.74
 10    14 ms     7 ms     6 ms  192.178.97.249
 11     7 ms     6 ms     7 ms  172.253.66.87
 12     6 ms     6 ms     6 ms  lhr48s08-in-f3.1e100.net [172.217.169.35]

Trace complete.
```

Exercise 4: (🤔) Identify potential bottlenecks or slow points in the network path for each domain.

Exercise 5: Experiment trace routes to the same domain from different locations (if possible) to see how the paths might vary.

3. Building a Simple HTTP Client

The application layer is the top layer of the TCP/IP model (or the OSI model). It's the layer closest to the user and is responsible for providing network services to applications. Think of it as the layer where applications that use the network, like web browsers, email clients, and file transfer programs, operate. This is where protocols like HTTP, SMTP (for email), and FTP (for file transfer) live. In this section, let us simulate a very basic web browser.

Objective:
Create a simple Python script that acts as an HTTP client to fetch data from a web server.

Code Example:

```
import socket

# Create a socket object
client_socket = socket.socket(socket.AF_INET, socket.SOCK_STREAM)

# Define the server address and port
server_address = ('www.example.com', 80)

# Connect to the server
client_socket.connect(server_address)

# Send an HTTP GET request
request = "GET / HTTP/1.1\r\nHost: www.example.com\r\n\r\n"
client_socket.send(request.encode())

# Receive the response
response = client_socket.recv(4096)
print(response.decode())

# Close the socket
client_socket.close()
```

How This Script Works:

This script simulates a very basic web browser. It uses the socket library in Python to:

1. **Create a Socket:** A socket is like an endpoint for network communication. The script creates a socket object, specifying that it will use the Internet Protocol (IP) and TCP (Transmission Control Protocol). TCP is a reliable, connection-oriented protocol, meaning it ensures that data is delivered in the correct order and without errors.
2. **Connect to the Server:** The script then connects the socket to the web server at www.example.com on port 80. Port 80 is the standard port for HTTP. This establishes a connection between your Python script (acting as a client) and the web server.
3. **Send an HTTP Request:** This is the crucial part. The script crafts an HTTP GET request. This request tells the web server that the client wants to retrieve the main web page (/). The Host header specifies the domain name. The \r\n sequences represent carriage return and line feed characters, which are used to separate lines in

HTTP headers. This request is then sent over the socket to the server.

4. **Receive the Response:** The script waits for the server to respond. The recv() method receives the data sent by the server (the HTTP response) and stores it in the response variable.
5. **Print the Response:** The script then prints the received response. This response will be the HTML content of the www.example.com web page, along with some HTTP headers that provide information about the response.
6. **Close the Socket:** Finally, the script closes the socket, terminating the connection with the server.

4. Python Requests library

While using the socket module provides a deeper understanding of how network communication works, it can be cumbersome for everyday tasks. Python's high-level requests library simplifies HTTP requests and is widely used in industry.

Why Use requests?

- Easier syntax for making HTTP requests.
- Automatic handling of headers, cookies, and character encoding.
- Built-in error handling and support for HTTPS.

To installing the requests Library using the following command:

```
!pip install requests
```

Using the requests library let us re develop the previous script:

```
import requests

response = requests.get('http://www.example.com')
print(response.text)
```

You'll notice that the script is easier to use.

Comparison with the socket Module

Feature	socket Module	requests Library
Low-level control	✅ Yes	❌ No
Easy to use	❌ No	✅ Yes
HTTPS Support	❌ Manual	✅ Automatic
Header Handling	❌ Manual	✅ Automatic
Error Handling	❌ Minimal	✅ Built-in

5. HTTP Requests Type

HTTP is a versatile protocol that allows clients and servers to communicate in various ways depending on the type of operation required. The most common request types include:

- **GET**: Fetches data from a server (read-only).
- **POST**: Sends data to a server to create or update a resource.
- **PUT**: Updates an existing resource on the server.
- **DELETE**: Deletes a resource on the server.

POST Requests: Sends data to a server

The POST method is used to send data to a server to create or update a resource. Unlike a GET request, where the server only retrieves information, a POST request submits data as part of the request body. POST requests are used in web applications for tasks such as:

- User registration and login forms
- Uploading files or submitting forms
- Creating new records in a database

Why Use POST Instead of GET?

- Data in the Body: Data is sent in the request body, not the URL, which keeps the URL clean and allows larger data payloads.

- Secure Data Handling: POST requests are more secure for sensitive data as the information is not stored in browser history or visible in URL logs.
- State Change: POST requests are intended to cause changes on the server (like creating new resources).

Code Example:

```
import requests

url = 'https://jsonplaceholder.typicode.com/posts'
data = {
    "title": "Sample Post",
    "body": "This is an example post body.",
    "userId": 1
}

response = requests.post(url, json=data)
print(f"Status Code: {response.status_code}")
print("Response Body:", response.json())
```

PUT Request: Updating Data on a Server

A **PUT request** is used to update an existing resource on the server. It typically requires the full updated data to be sent in the request body.

Code example:

```
import requests

url = 'https://jsonplaceholder.typicode.com/posts/1'
updated_data = {
    "id": 1,
    "title": "Updated Title",
    "body": "This post content has been updated.",
    "userId": 1
}

response = requests.put(url, json=updated_data)
print(f"Status Code: {response.status_code}")
print("Updated Resource:", response.json())
```

DELETE Request: Removing Data from a Server

A **DELETE request** is used to remove a resource from the server. It typically doesn't require a request body.

Code example:

```
import requests

url = 'https://jsonplaceholder.typicode.com/posts/1'

response = requests.delete(url)
print(f"Status Code: {response.status_code}")
if response.status_code == 200:
    print("Resource successfully deleted.")
```

Congratulations! You've completed the Python Workshop on the Application Layer in Networks. You've learned how to build basic network applications using Python, and you've tackled some challenging exercises to further your understanding. Keep experimenting and building more complex applications to deepen your knowledge of networking and Python.

Happy coding! 🚀