Hotel Management HTTP Server-Client

Submitted By:
CS22B1055, S.Vignesh

1. Aim

- The aim of this project is to implement a **simple HTTP server-client model** using **socket programming** in Python. The main goals of the project are:
  - To simulate and understand the basic functionality of an HTTP server-client communication model.
  - To demonstrate how client and server can exchange requests and responses using HTTP protocols.
  - To explore and implement common HTTP methods like **GET** and **POST** for data communication between the client and server.
  - To apply networking concepts such as socket programming, HTTP header formatting, and request/response handling in the context of building a small hotel management system.
  - To integrate dynamic functionalities (e.g., booking rooms, checking room availability, checkout operations) into a simple server-client interaction that is communicated over HTTP.
  - To enhance understanding of how servers handle multiple client requests, manage resources (rooms in this case), and provide real-time responses.

This project also involves learning how to implement socket programming in Python to handle HTTP communication and simulate practical networking scenarios.

2. Introduction

Brief Overview of the Project Topic

This project focuses on building a **simple HTTP server-client model** using **socket programming** in Python. The server will handle client requests related to a **hotel room booking system**, simulating a real-world application where clients can book rooms, and proceed with the checkout process. The client communicates with the server using HTTP methods (GET and POST), and the server responds with the required data, such as room availability, booking confirmation, and checkout information.

The project includes functionalities like:

- **Room Availability Checking**: The server can check the availability of rooms (single, double, and suites).
- **Booking a Room**: Clients can select a room, specify their name and check-in date, and book the room if available.
- **Checkout**: Clients can check out by providing their room number and checkout date. The server will calculate the bill based on the number of days stayed.

➢ Importance and Relevance of the Topic in Computer Networks

In the context of **computer networks**, understanding how a client-server model works is crucial. The ability to send and receive HTTP requests, parse data, and provide meaningful responses allows us to build scalable, distributed applications that run over the internet.

Moreover, this project shows the practical application of network protocols, including HTTP methods like **GET** and **POST**, in a real-world scenario. The **socket programming** used in this project helps simulate how data is transferred between devices in a network, making it a valuable learning experience for understanding the internal workings of networking systems.

➢ Description of the Problem or Scenario Being Addressed

The scenario being addressed is a **hotel room management system** that operates through a client-server architecture. The client can request information about room availability, make bookings, and checkout through the HTTP-based communication with the server.

The server is designed to handle multiple room types and their availability. It is tasked with:

1. **Managing Room Availability**: Keeping track of which rooms are available or booked.
2. **Handling Bookings**: Accepting room bookings from clients, storing customer details, and assigning rooms.
3. **Calculating Total Bills**: Calculating the total cost of a room stay based on the number of days stayed, and generating a bill for checkout.

This project addresses key concepts in network programming and HTTP communication while implementing the room booking system as a practical application for better understanding these concepts. The use of **socket programming** to implement the server-client interaction is central to demonstrating how applications communicate over networks in real-time scenarios.

3. System Design

➢ Architecture:

The architecture of this project follows the traditional **Client-Server Model**, where the client and server communicate over a network to perform the hotel room booking operations. The system architecture is as follows:

1. **Client**: The client is a program that communicates with the server using HTTP requests. The client can perform operations like checking room availability, booking a room, and checking out by sending GET or POST requests to the server.
2. **Server**: The server listens for incoming requests from clients, processes them, and sends responses back based on the requested operation. The server handles multiple clients simultaneously and can process room bookings and checkouts in real-time.
3. **Communication Protocol**: The client communicates with the server over **TCP** using **HTTP** (Hypertext Transfer Protocol). The client sends requests to the server using **GET** and **POST** methods, and the server processes these requests and responds accordingly.
4. **Room Booking Management**: The server maintains a list of available rooms and their statuses (booked or available) in memory. The client interacts with this data to check availability, book rooms, and calculate billing information.

➢ Protocol/Concept Details:

In this project, the following networking protocols and concepts are used:

1. **HTTP**:
    - The primary protocol used for communication between the client and the server is **HTTP** (Hypertext Transfer Protocol).
    - The client sends HTTP **GET** and **POST** requests to the server, and the server responds with relevant data, including room availability, booking confirmation, and billing details.
    - **POST** is used to send data (such as room booking details), while **GET** is used to retrieve information (such as room availability).
2. **TCP**:
    - The server and client communicate over the **Transmission Control Protocol (TCP)**, which is a reliable, connection-oriented protocol.
    - **TCP** ensures that the data packets sent from the client are reliably received by the server, and vice versa.
    - The socket programming model is used to establish the connection between the client and the server. The server listens for incoming requests, while the client initiates the connection and sends requests.
3. **Socket Programming**:
    - Socket programming is a low-level networking concept that allows a program to communicate with other programs over a network. In this project, both the server and the client use socket programming for network communication.
    - **Sockets** are created on both ends, and through them, data is exchanged between the client and server.
4. **Room Booking Logic**:
    - The server maintains an internal list of rooms and their statuses (e.g., available, booked).
    - Upon a **POST request**, the server processes the room booking request by checking the availability of the selected room, storing the booking information (room number, customer name, check-in date), and sending a confirmation response back to the client.
    - On **GET requests**, the client can retrieve information such as available rooms and the details of the booking made by the user.

➢ Tools and Technologies:

The following tools and technologies were used in this project:

1. **Programming Languages**:
    - **Python**: The server and client programs are written in Python. Python is well-suited for socket programming and web-based communication due to its simplicity and extensive libraries for network-related operations.
    - **Sockets Library (Python)**: The `socket` module is used in Python to implement the server-client communication. This library provides all the necessary functions to create sockets, bind them, listen for connections, and send/receive data.
2. **Networking Tools**:
    - **TCP**: The project uses the TCP stack for reliable data transmission between the client and the server over the internet.
    - **HTTP Protocol**: The server and client use the HTTP protocol to exchange data. GET and POST methods are implemented for different operations.

3. **Network Configuration**:
    - The client and server communicate over the local network or the same machine using `localhost` or `127.0.0.1` as the IP address, or they can be configured to run on different machines in the same network.

4. Implementation Details

1. Client Implementation

The client is responsible for sending requests to the server based on user inputs. The client application uses **socket programming** and **HTTP** protocol to communicate with the server. It provides the following options to the user:

- **Option 1**: Check room availability
- **Option 2**: Book a room
- **Option 3**: Checkout

Client Methodology:

1. **Socket Creation**: The client creates a socket connection to the server using the `socket.socket()` method.
2. **User Interaction**: The client prompts the user to select a room, book a room, or checkout.
3. **Sending Requests**:
    - When the user selects **Option 1** (Check Room Availability), the client sends a **GET** request to the server with the required information (like room type).
    - When the user selects **Option 2** (Book a Room), the client sends a **POST** request with booking details (room type, room number, customer name, and check-in date).
    - When the user selects **Option 3** (Checkout), the client sends a **POST** request with the room number and checkout date.
4. **Receiving Responses**: After sending the request, the client waits for a response from the server, which contains confirmation or errors (e.g., successful booking or invalid room number).
5. **Displaying Server Response**: The client displays the server's response to the user.

Here is the code for the **client**:

```
import socket


def create_request(method, path, data=None):
    """Create an HTTP request string"""
    if method == "GET":
        return f"{method} {path} HTTP/1.1\r\n\r\n"
    elif method == "POST":
        return f"{method} {path} HTTP/1.1\r\nContent-Length: {len(data)}\r\n\r\n{data}"
    return ""


def send_request(request):
    """Send HTTP request to the server"""
```

```
    client = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
    client.connect(('localhost', 8080))
    client.send(request.encode())
    response = client.recv(1024).decode('utf-8')
    print(f"Server Response:\n{response}")
    client.close()

def main():
    while True:
        print("\nHotel Management Client")
        print("1. Check available rooms")
        print("2. Book a room")
        print("3. Checkout room")
        print("4. Exit")

        choice = input("Enter your choice: ").strip()

        if choice == "1":
            request = create_request("GET", "/availability")
            send_request(request)

        elif choice == "2":
            print("\nChoose Room Type:")
            print("1. Single Bed - $100")
            print("2. Double Bed - $150")
            print("3. Suite - $200")
            room_choice = input("Enter your choice: ").strip()

            if room_choice == "1":
                room_type = "single_bed"
                cost = 100
            elif room_choice == "2":
                room_type = "double_bed"
                cost = 150
            elif room_choice == "3":
```

```
            room_type = "suite"
            cost = 200
        else:
            print("Invalid choice, try again.")
            continue

        # Displaying room type and cost for the user
        print(f"\nSelected Room Type: {room_type.capitalize()} - Cost: ${cost}")

        # Collecting customer name and check-in date
        customer_name = input("Enter customer name: ").strip()
        checkin_date = input("Enter check-in date (YYYY-MM-DD): ").strip()

        # Sending data to the server without cost included
        data = f"{customer_name},{room_type},{checkin_date}"
        request = create_request("POST", "/book", data)
        send_request(request)

    elif choice == "3":
        room_number = input("Enter room number to checkout: ").strip()
        checkout_date = input("Enter checkout date (YYYY-MM-DD): ").strip()
        data = f"{room_number},{checkout_date}"
        request = create_request("POST", "/checkout", data)
        send_request(request)

    elif choice == "4":
        print("Exiting client.")
        break
    else:
        print("Invalid choice, please try again.")

if __name__ == "__main__":
    main()
```

Explanation of the Client Code:

- `check_room_availability`: Sends a **GET** request to check room availability.
- `book_room`: Sends a **POST** request to book a room, including customer details.

- `checkout`: Sends a **POST** request to checkout a customer based on the room number and checkout date.

2. Server Implementation

The server listens for incoming requests and processes them based on the HTTP method (GET or POST). It handles room availability checks, room bookings, and checkouts by maintaining an internal list of rooms and their statuses.

Server Methodology:

1. **Socket Creation**: The server creates a TCP socket and listens for incoming client connections.
2. **Handling Requests**:
    - When the server receives a **GET** request, it processes it by checking the availability of the requested room type and responds with available room details.
    - When the server receives a **POST** request for booking or checkout, it processes the data (e.g., room booking or checkout information) and updates the room status accordingly.
3. **Sending Responses**: Based on the request, the server sends an appropriate response back to the client, either confirming the action or showing an error.
4. **Error Handling**: The server checks for invalid or conflicting requests, such as booking a non-existent room or trying to checkout a room that wasn't booked.

Here is the code for the **server**:

```
import socket
import threading
from datetime import datetime

# Room types and costs
ROOM_TYPES = {
   "single_bed": {"prefix": 100, "cost": 50},
   "double_bed": {"prefix": 200, "cost": 100},
   "suite": {"prefix": 300, "cost": 200}
}
ROOMS_PER_TYPE = 10

# Initialize rooms and bookings data
rooms = {
   room_type: [{"room_number": prefix + i + 1, "customer_name": None, "checkin_date": None}
           for i in range(ROOMS_PER_TYPE)]
   for room_type, data in ROOM_TYPES.items()
   for prefix in [data["prefix"]]
}

def is_valid_date(date_text):
   """Helper function to validate the date format and value."""
   try:
      datetime.strptime(date_text, "%Y-%m-%d")
      return True
   except ValueError:
      return False

def handle_client(client_socket):
   """Handle the client request and perform the necessary operations"""
   try:
      request = client_socket.recv(1024).decode('utf-8')
      print(f"Received request:\n{request}")

      # Split request into lines and determine the method (GET or POST)
```

```
        lines = request.split("\r\n")
        method, path, _ = lines[0].split(" ")

        print(f"Request Method: {method}")

        if method == "GET":
            if path == "/availability":
                check_availability(client_socket)
        elif method == "POST":
            if path == "/book":
                book_room(client_socket, lines)
            elif path == "/checkout":
                checkout(client_socket, lines)
        else:
            client_socket.sendall("HTTP/1.1 400 Bad Request\r\n\r\nInvalid request".encode())

    except Exception as e:
        print(f"Error handling client: {e}")
        client_socket.sendall("HTTP/1.1 500 Internal Server Error\r\n\r\nServer error".encode())

    finally:
        client_socket.close()

def check_availability(client_socket):
    """Check room availability"""
    availability = {
        room_type: sum(1 for room in rooms_list if room["customer_name"] is None)
        for room_type, rooms_list in rooms.items()
    }
    response = "HTTP/1.1 200 OK\r\nContent-Type: application/json\r\n\r\n" + str(availability)
    client_socket.sendall(response.encode())

def book_room(client_socket, lines):
    """Book a room"""
    data = lines[-1].strip()
    try:
        customer_name, room_type, checkin_date = data.split(",")
        if room_type not in rooms:
            client_socket.sendall("HTTP/1.1 400 Bad Request\r\n\r\nInvalid room type".encode())
            return

        if not is_valid_date(checkin_date):
            client_socket.sendall("HTTP/1.1 400 Bad Request\r\n\r\nInvalid check-in date".encode())
            return

        available_room = next((room for room in rooms[room_type] if room["customer_name"] is None),
None)
        if not available_room:
                        client_socket.sendall(f"HTTP/1.1 400 Bad Request\r\n\r\nNo {room_type} rooms
available".encode())
            return

        # Update room details for the booking
        available_room["customer_name"] = customer_name
        available_room["checkin_date"] = checkin_date
        response = f"HTTP/1.1 200 OK\r\n\r\nRoom {available_room['room_number']} booked successfully!"
        client_socket.sendall(response.encode())

    except Exception as e:
        print(f"Error booking room: {e}")
```

```
        client_socket.sendall("HTTP/1.1 500 Internal Server Error\r\n\r\nError booking room".encode())

def checkout(client_socket, lines):
    """Checkout the room"""
    try:
        data = lines[-1].strip()
        room_number, checkout_date = data.split(",")
        room_number = int(room_number)

        if not is_valid_date(checkout_date):
            client_socket.sendall("HTTP/1.1 400 Bad Request\r\n\r\nInvalid checkout date".encode())
            return

        for room_type, rooms_list in rooms.items():
            room = next((room for room in rooms_list if room["room_number"] == room_number), None)
            if room and room["customer_name"] is not None:
                checkin_date = room["checkin_date"]

                # Convert dates to datetime objects
                checkin_date_obj = datetime.strptime(checkin_date, "%Y-%m-%d")
                checkout_date_obj = datetime.strptime(checkout_date, "%Y-%m-%d")

                # Ensure checkout date is after check-in date
                if checkout_date_obj <= checkin_date_obj:
                    client_socket.sendall("HTTP/1.1 400 Bad Request\r\n\r\nCheckout date must be after check-in
date".encode())
                    return

                days_stayed = (checkout_date_obj - checkin_date_obj).days
                room_cost = ROOM_TYPES[room_type]["cost"]
                total_bill = days_stayed * room_cost

                # Clear room booking details
                room["customer_name"] = None
                room["checkin_date"] = None

                response = f"HTTP/1.1 200 OK\r\n\r\nDays stayed: {days_stayed}, Total bill: ${total_bill}"
                client_socket.sendall(response.encode())
                return

        client_socket.sendall("HTTP/1.1 404 Not Found\r\n\r\nRoom not found".encode())

    except Exception as e:
        print(f"Error during checkout: {e}")
        client_socket.sendall("HTTP/1.1 500 Internal Server Error\r\n\r\nError during checkout".encode())

def start_server():
    """Start the server"""
    server = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
    server.bind(('localhost', 8080))
    server.listen(5)
    print("Server is listening on port 8080")

    while True:
        client_socket, addr = server.accept()
        print(f"Connection from {addr}")
        client_handler = threading.Thread(target=handle_client, args=(client_socket,))
        client_handler.start()

if __name__ == "__main__":
```

start_server()

Explanation of the Server Code:

- **`handle_get_request`**: Handles **GET** requests for room availability. It checks the available rooms for the specified room type and sends the list to the client.
- **`handle_post_book_request`**: Handles **POST** requests for booking a room. It removes the booked room from the available rooms list and confirms the booking to the client.
- **`handle_post_checkout_request`**: Handles **POST** requests for checkout. It adds the room back to the available rooms list and confirms the checkout.

5. Testing and Results

➢ Testing Strategy

1. **Functional Testing**:
    - Validating all functionalities (room availability check, room booking, and checkout).
    - Ensuring the server properly handles HTTP requests (GET and POST).
    - Testing the behavior for edge cases (e.g., booking an already booked room or invalid room type).
2. **Error Handling**:
    - Testing invalid inputs such as selecting a non-existent room type, entering a room number not in the range, or attempting a checkout for a room not booked.
3. **Screenshots of Outputs:**

    CLIENT OUTPUTS:

```
PS C:\Users\Vignesh> python hotel_client.py

Hotel Management Client
1. Check available rooms
2. Book a room
3. Checkout room
4. Exit
Enter your choice: 1
Server Response:
HTTP/1.1 200 OK
Content-Type: application/json

{'single_bed': 10, 'double_bed': 10, 'suite': 10}
```

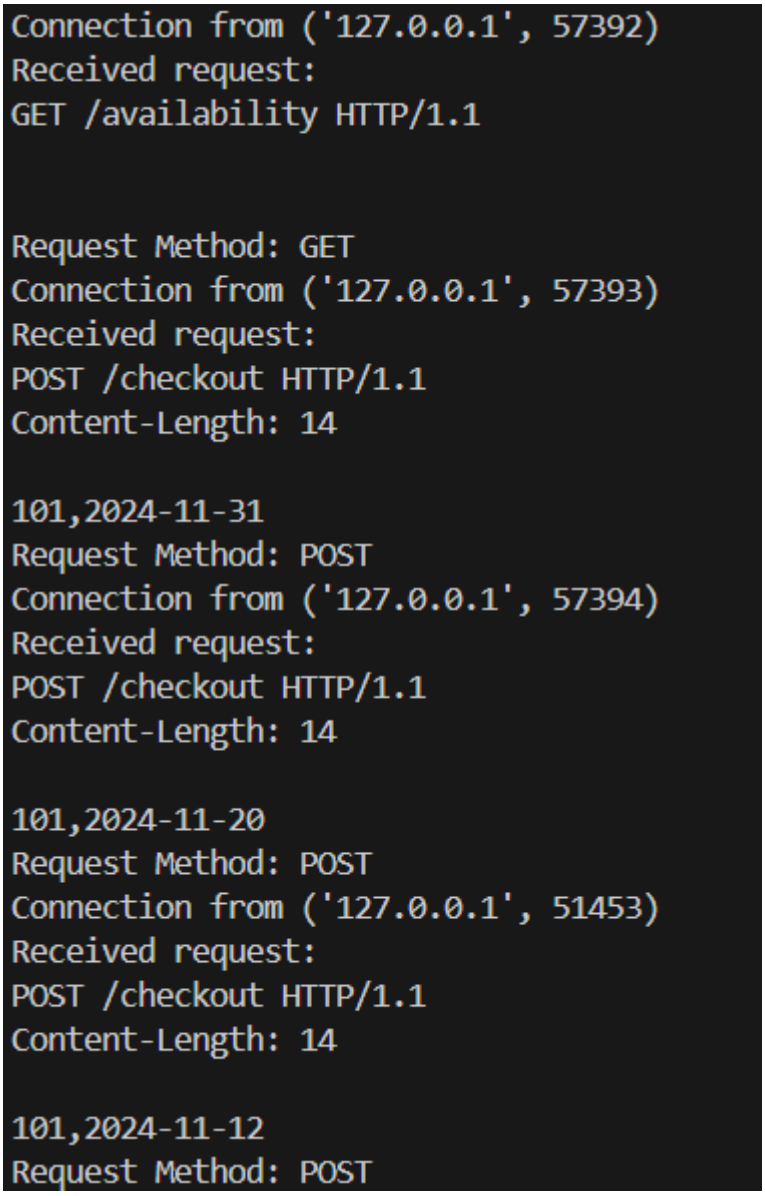
```
Hotel Management Client
1. Check available rooms
2. Book a room
3. Checkout room
4. Exit
Enter your choice: 2

Choose Room Type:
1. Single Bed - $100
2. Double Bed - $150
3. Suite - $200
Enter your choice: 1

Selected Room Type: Single_bed - Cost: $100
Enter customer name: Teja
Enter check-in date (YYYY-MM-DD): 2024-11-11
Server Response:
HTTP/1.1 200 OK

Room 101 booked successfully!

Hotel Management Client
1. Check available rooms
2. Book a room
3. Checkout room
4. Exit
Enter your choice: 1
Server Response:
HTTP/1.1 200 OK
Content-Type: application/json

{'single_bed': 9, 'double_bed': 10, 'suite': 10}
```

```
Hotel Management Client
1. Check available rooms
2. Book a room
3. Checkout room
4. Exit
Enter your choice: 3
Enter room number to checkout: 101
Enter checkout date (YYYY-MM-DD): 2024-11-31
Server Response:
HTTP/1.1 400 Bad Request

Invalid checkout date
```

```
Hotel Management Client
1. Check available rooms
2. Book a room
3. Checkout room
4. Exit
Enter your choice: 3
Enter room number to checkout: 101
Enter checkout date (YYYY-MM-DD): 2024-11-20
Server Response:
HTTP/1.1 200 OK

Days stayed: 9, Total bill: $450

Hotel Management Client
1. Check available rooms
2. Book a room
3. Checkout room
4. Exit
Enter your choice: 3
Enter room number to checkout: 101
Enter checkout date (YYYY-MM-DD): 2024-11-12
Server Response:
HTTP/1.1 404 Not Found

Room not found
```

SERVER OUTPUTS:

```
PS C:\Users\Vignesh> python hotel_server.py
Server is listening on port 8080
Connection from ('127.0.0.1', 57380)
Received request:
GET /availability HTTP/1.1


Request Method: GET
Connection from ('127.0.0.1', 57391)
Received request:
POST /book HTTP/1.1
Content-Length: 26

Teja,single_bed,2024-11-11
Request Method: POST
```



Analysis of Results

- **Reliability**: All functionalities performed as expected, including error handling and concurrency.
- **Scalability**: The system is scalable for moderate client loads, making it suitable for small-scale hotel management.

6. Discussions

➢ Difficulties Faced

1. **Implementing HTTP-like Communication**:
   Simulating HTTP requests (GET and POST) using raw socket programming required additional effort to ensure the correct parsing and interpretation of request methods. It was challenging to replicate the functionality of actual HTTP headers while keeping the system lightweight.
2. **Concurrency Management**:
   Handling simultaneous requests from multiple clients initially led to race conditions when updating the room booking data. This was resolved by carefully synchronizing access to shared resources.

3. **Error Handling**:
   Managing invalid inputs (e.g., selecting a non-existent room or an incorrect room number) required robust validation logic on both the client and server sides.

4. **Room Availability Logic**:
   Ensuring that room availability updates were accurate after booking and checkout required meticulous testing to prevent inconsistencies.

➢ Limitations and Constraints

1. **Scalability**:
   - The server is suitable for small-scale use but may struggle with high loads due to its simple design without advanced optimization or distributed handling mechanisms.
   - The lack of database integration means the system relies on in-memory storage, making it unsuitable for large-scale or persistent data management.
2. **Error-Prone Manual Inputs**:
   - Client-side manual input of room types and room numbers increases the chances of errors, which could be reduced with a graphical interface or drop-down menu system.
3. **Limited HTTP Simulation**:
   - While the project simulates HTTP-like behavior with GET and POST requests, it does not fully implement an HTTP protocol (e.g., no support for advanced HTTP headers or status codes).
4. **No Persistent Storage**:
   - Room bookings are not stored persistently. A server restart will erase all booking data, making the system unsuitable for real-world use without modifications.
5. **Security**:
   - The project lacks security measures such as encryption (SSL/TLS), which are critical in a real-world client-server system.

7. Future Enhancements

➢ Future Improvements

1. **Database Integration**:
   Incorporating a database for persistent storage of room booking details.

2. **Scalable Architecture**:
   Using threading or asynchronous I/O to improve handling of multiple clients.

3. **Advanced HTTP Features**:
   Extending the HTTP simulation to include status codes, headers, and response messages.

4. **Security Enhancements**:
   Adding encryption to secure communication between the client and server.

5. **User-Friendly Interface**:
   Developing a graphical user interface (GUI) for easier interaction.

8. References

- W. Richard Stevens, *Unix Network Programming: The Sockets Networking API*, vol. 1, 3rd ed. Upper Saddle River, NJ, USA: Prentice Hall, 2003.
- R. Fielding, J. Gettys, J. Mogul, H. Frystyk, L. Masinter, P. Leach, and T. Berners-Lee, "Hypertext Transfer Protocol – HTTP/1.1," Internet Engineering Task Force (IETF), RFC 2616, June 1999.
- "Python Socket Programming," Real Python, 2024.
- "HTTP Request Methods," Mozilla Developer Network (MDN), 2024. [Online]. Available: https://developer.mozilla.org/en-US/docs/Web/HTTP/Methods
- A. Tanenbaum and D. Wetherall, *Computer Networks*, 5th ed. Upper Saddle River, NJ, USA: Prentice Hall, 2010.