Lecture 3: Web Crawler, basics of HTTP

Client/Server Description

Introduction to Client-Server Architecture

In the world of computer networking and distributed computing, the client-server architecture is a fundamental and widely used model that defines how devices and software applications interact with each other over a network. This architecture forms the backbone of the modern internet and plays a crucial role in how information and services are accessed, processed, and shared.

Understanding Client-Server Architecture

At its core, the client-server architecture is a model of communication where two distinct entities, namely the **client** and the **server**, collaborate to accomplish various tasks. These entities are essentially software programs or hardware devices that perform specific roles in the communication process.

- **Client**: The client is a device or software application that initiates a request for a service or resource. It acts as the user's interface to access data or services provided by the server. Clients send requests to servers, receive responses, and typically display or utilize the results. Examples of clients include web browsers, email clients, and mobile apps.
- **Server**: The server, on the other hand, is a specialized device or software application responsible for fulfilling client requests. Servers listen for incoming requests, process them, and send back the appropriate responses. They are designed to provide specific services or resources, such as web pages, databases, email, or file storage.

Key Characteristics of Client-Server Architecture

1. **Decentralization**: In client-server architecture, the server is typically a centralized system that stores and manages data or services. Clients, which can be distributed across various locations, request and access these resources remotely.
2. **Communication**: Clients and servers communicate with each other over a network using predefined protocols and standards. Common communication protocols include HTTP for web browsers and servers, SMTP/IMAP for email clients and email servers, and FTP for file transfer clients and servers.
3. **Request-Response Model**: The client initiates a request, and the server processes that request and returns a response. This request-response pattern is the foundation of client-server communication.
4. **Scalability**: The client-server model allows for scalability. Servers can handle multiple client connections simultaneously, making it possible to serve a large number of users efficiently.
5. **Resource Sharing**: Servers provide shared resources, such as databases, files, or services, that clients can access. This sharing of resources enables collaboration and data access across multiple clients.
6. **Security**: Security measures can be implemented at both the client and server ends to protect data and ensure secure communication. Authentication, encryption, and access control are essential components of client-server security.

Use Cases of Client-Server Architecture

Client-server architecture is prevalent in various domains and applications, including:

- **Web Services**: When you browse a website, your web browser (the client) requests web pages and other resources from a web server. The server processes these requests and sends back the requested content.
- **Email**: Email clients (e.g., Outlook, Gmail) communicate with email servers to send and receive messages.
- **Databases**: Client applications access database servers to retrieve and update data, commonly used in business and enterprise environments.
- **File Sharing**: Network-attached storage (NAS) devices allow clients to access and share files stored on a central server.

- **Online Gaming**: Online multiplayer games often use client-server architecture to facilitate gameplay and data synchronization.
- **Cloud Computing**: Cloud services use client-server principles to provide on-demand computing resources, storage, and applications.

HTTP Protocol:

1. Detailed overview of the HTTP (Hypertext Transfer Protocol).
   - Explanation of HTTP methods (GET, PUT, POST, DELETE).
   - Understanding request and response headers.
2. Open your terminal.
3. Type the following command to initiate a Telnet session with the web server (replace `example.com` with the actual website you want to connect to):

```
telnet example.com 80
```

4. Once you're connected, enter the following GET request manually:

```
GET / HTTP/1.1
Host: example.com
Connection: close

```

   Make sure to press Enter twice after entering the request to send it.
5. The server will respond with the HTTP response, and you'll see the content displayed in your terminal.

Here's a breakdown of the GET request:

- `GET / HTTP/1.1`: This is the GET request line, specifying the HTTP method, path (`/` for the root), and HTTP version (HTTP/1.1).
- `Host: example.com`: This is the `Host` header, which should match the target website's domain.
- `Connection: close`: This header indicates that the connection should be closed after the response is received.

This method allows you to make a GET request using Telnet directly in your terminal without the need for a separate script.

Simple Example of GET Request

Certainly, here's Python code that demonstrates a basic GET request using the `requests` library. It also includes handling HTTP status codes and exceptions, as well as parsing HTML content from a web page using the `BeautifulSoup` library for parsing:

```
import requests
from bs4 import BeautifulSoup

# Replace with the URL you want to GET
url = "https://example.com"

try:
    # Send a GET request
    response = requests.get(url)

    # Check the status code
    if response.status_code == 200:
        print("GET request successful!")

        # Parse the HTML content with BeautifulSoup
        soup = BeautifulSoup(response.text, 'html.parser')

        # Example: Extract and print the page title
        page_title = soup.title.string
```

```
        print(f"Page Title: {page_title}")

        # You can further parse and extract data from the HTML as needed

    else:
        print(f"GET request failed with status code: {response.status_code}")

except requests.exceptions.RequestException as e:
    print(f"An error occurred: {e}")

except Exception as e:
    print(f"An unexpected error occurred: {e}")

# DOM Parser.
# SAX Parser.
# Xpath Parser.
```

In this code:

1. We import the necessary libraries, `requests` for making HTTP requests and `BeautifulSoup` for parsing HTML content.
2. Replace the `url` variable with the URL of the web page you want to fetch.
3. The `try` block sends a GET request to the specified URL.
4. It checks the HTTP status code to determine if the request was successful (status code 200).
5. If the request is successful, we parse the HTML content using BeautifulSoup. In this example, we extract and print the page title as a demonstration.
6. If the request fails or an exception occurs, appropriate error messages are displayed.

Make sure to install the `requests` and `beautifulsoup4` libraries if you haven't already by running:

```
pip install requests
pip install beautifulsoup4
```

This code will fetch a web page, parse its HTML content, and print the page title. You can extend it to extract and manipulate other elements from the page as needed.

GET/PUT/POST/Delete Methods

1. PUT Method:

Purpose: The PUT method is used to update or replace an existing resource or create a new resource if it doesn't already exist at the specified URI (Uniform Resource Identifier).

Key Characteristics:

- Idempotent: A PUT request is idempotent, meaning that making the same request multiple times will have the same effect as making it once. It should not have any side effects on subsequent requests.
- Replaces Entire Resource: When you send a PUT request, you typically send the entire representation of the resource you want to update. The server replaces the existing resource with the new data provided.
- URI Specifies the Resource: The URI in the request specifies the location of the resource you want to update.

Common Use Cases:

- Updating an existing record in a database.
- Replacing a file on a server with an updated version.
- Creating a new resource if it doesn't exist (sometimes called "PUT or Create").

2. POST Method:

Purpose: The POST method is used to submit data to be processed to a specified resource. Unlike PUT, POST does not replace the entire resource; instead, it creates a new subordinate resource or processes the data in a specific way on the server.

Key Characteristics:

- Not Idempotent: POST requests are not idempotent. Making the same POST request multiple times may result in different outcomes or side effects.
- Data in the Request Body: POST requests often include data in the request body. This data can be in various formats, such as JSON, XML, or form data.
- URI May Not Specify the Resource: The URI in a POST request may not always specify the exact resource. It may simply specify the endpoint or route to handle the data.

Common Use Cases:

- Creating a new resource on the server (e.g., adding a new user to a database).
- Submitting form data on a web page.
- Uploading a file to a server.
- Initiating a transaction or process that may have side effects.

3. DELETE Method:

Purpose: The DELETE method is used to request the removal of a resource at the specified URI.

Key Characteristics:

- Idempotent: Similar to PUT, DELETE requests are idempotent. Repeated requests to delete the same resource will have the same effect as making the request once.
- No Request Body: DELETE requests typically do not have a request body. The URI specifies the resource to be deleted.
- Resource Deletion: When a DELETE request is successful, the resource at the specified URI is removed from the server.

Common Use Cases:

- Deleting a record from a database.
- Removing a file from a server.
- Deleting a user account or item from a list.

In summary, PUT is used for updating or replacing resources, POST is used for creating new resources or processing data, and DELETE is used for removing resources. Understanding the differences between these HTTP methods is crucial when designing RESTful APIs and working with web services. Each method has its own specific use cases and behavior, so choosing the right one is essential for proper web application functionality.

Requests in python

Free API - Huge List of Public APIs For Testing [No Key] - Apipheny

```python
# PUT REQUEST
api_url = "https://httpbin.org/put"
data = {"name":"test","salary":"123","age":"23"}

try:
    # Send a PUT request with the data
    put_response = requests.put(api_url, json=data)

    # Check the response status code
    if put_response.status_code == 200:
        print("PUT Request Successful!")
        print(put_response.json())  # Response data, if any
    else:
```

```
        print(f"PUT Request Failed with Status Code: {put_response.status_code}")

except requests.exceptions.RequestException as e:
    print(f"An Error Occurred: {e}")
```

```
# POST REQUEST
api_url = "https://httpbin.org/post"
data = {"name":"test","salary":"123","age":"23"}

try:
    put_response = requests.post(api_url, json=data)

    # Check the response status code
    if put_response.status_code == 200:
        print("PUT Request Successful!")
        print(put_response.json())  # Response data, if any
    else:
        print(f"POST Request Failed with Status Code: {put_response.status_code}")

except requests.exceptions.RequestException as e:
    print(f"An Error Occurred: {e}")
```

```
# DELETE REQUEST
api_url = "https://httpbin.org/delete"
data = {"name":"test","salary":"123","age":"23"}

try:
    put_response = requests.delete(api_url)

    # Check the response status code
    if put_response.status_code == 200:
        print("DELETE Request Successful!")
        print(put_response.json())  # Response data, if any
    else:
        print(f"PUT Request Failed with Status Code: {put_response.status_code}")

except requests.exceptions.RequestException as e:
    print(f"An Error Occurred: {e}")
```

Web Crawling Fundamentals:

1. **Introduction to web crawling and web scraping.**
   - Importance of web crawling in data extraction and indexing.
2. **Building a Simple Web Crawler:**
   - Step-by-step guide on creating a basic web crawler in Python.
   - Setting up the crawling environment.
   - Defining the crawling scope and depth.
3. **Advanced Web Crawling Techniques:**
   - Handling different types of content (HTML, JSON, XML).
   - Implementing data storage and saving crawled data.
   - Managing concurrency and throttling requests.

```python
import requests
from bs4 import BeautifulSoup

# Replace with the URL of the web page you want to scrape
url = "http://hasdeu.md/"

try:
    # Send a GET request to the URL
    response = requests.get(url)

    # Check if the request was successful (status code 200)
    if response.status_code == 200:
        # Parse the HTML content of the page
        soup = BeautifulSoup(response.text, 'html.parser')

        # Find all anchor (link) tags in the HTML
        links = soup.find_all('a')

        # Extract and print the href attribute of each link
        for link in links:
            href = link.get('href')
            if href:
                print(href)

    else:
        print(f"Failed to retrieve the web page. Status code: {response.status_code}")

except requests.exceptions.RequestException as e:
    print(f"An error occurred during the request: {e}")

except Exception as e:
    print(f"An unexpected error occurred: {e}")
```

In this code:

- Replace `"<https://example.com"` > with the URL of the web page you want to scrape.
- The script sends a GET request to the specified URL using the `requests` library.
- If the request is successful (status code 200), it uses `BeautifulSoup` to parse the HTML content of the page.
- It then finds all anchor (`<a>`) tags in the HTML and extracts the `href` attribute from each one.
- The extracted links are printed to the console.

Make sure you have the `requests` and `beautifulsoup4` libraries installed by running `pip install requests beautifulsoup4` before running this code.

Ethical Considerations and Best Practices:

1. Discussing ethical aspects of web crawling.
    - Respecting robots.txt and website terms of service.
    - Avoiding overloading servers and getting blocked.

Practical Applications and Case Studies:

1. Exploring real-world use cases of web crawlers (e.g., search engines, data mining, price tracking).
    - Analyzing case studies of successful web crawling projects.
2. **Testing and Debugging:**

- Tips and techniques for testing your web crawler.
- Common debugging issues and how to resolve them.

3. **Scaling and Optimization:**
   - Strategies for scaling your web crawler to handle large datasets.
   - Performance optimization techniques.
4. **Q&A and Troubleshooting:**
   - Addressing common questions and challenges faced by learners.
   - Providing solutions to common problems.
5. **Final Project and Assessment:**
   - Encouraging students to build a web crawler for a specific task.
   - Assessment criteria and guidelines for the final project.
6. **Conclusion and Next Steps:**
   - Recap of key learnings.
   - Suggesting further resources for advanced web crawling and data scraping.