ADA University, School of IT and Engineering

Laman Gafarzada, ID:12024

Principles of Distributed Systems

Assignment 1 - Report on the Unreliable HTTP Server Implementation

26.10.2024

Introduction

This report outlines the implementation of an unreliable HTTP server and a client application for testing its functionality. The server simulates unreliable behavior by returning different HTTP status codes based on defined probabilities. The server also logs request events, and the client retrieves logs for analysis.

Problem Statement

The task was to create an HTTP server that responds to requests at the `/getbalance` route with random outcomes based on specified probabilities. Additionally, the server logs each request's timestamp and IP address, which can be retrieved via the `/getlogs` route. The server must also be tested using a client that can call both endpoints without assuming the server's host or port.

System Architecture

The system is composed of three main components:

1. Server (server.py): A Python HTTP server that handles incoming requests, simulates unreliable responses, and logs each request.

2. Client (client.py): A script that sends multiple requests to the server and retrieves logs.

3. Log Analyzer (analyze_logs.py): A utility to parse and analyze the log file, providing insights into the distribution of responses.

Key Features

1. Response Code Distribution: The server simulates the distribution of response codes as follows:

- 200 OK: 50%
- 403 Forbidden: 20%
- 500 Internal Server Error: 10%
- Timeout: 20%

2. Logging: Each request is logged in a JSON format, capturing the timestamp, client's IP address, and the server's response status code.

3. Timeout Simulation: The server simulates timeouts by not responding to certain requests after a specified delay.

4. Client Requests: The client sends 100 requests to the server and handles responses, including retries for timeouts.

5. Log Analysis: The log analyzer calculates the percentage of each response type based on the log entries.

Implementation Details

1. Server Code (server.py)
The server is implemented using Python's `**http.server**` and `**socketserver**` modules:

- **Event Logging:** Each request is logged in a JSON format, with each log entry written as a new line in `**logfile.json**` for easy parsing later.
  To maintain a log of requests, the server logs the timestamp, IP address, and status code. The relevant snippet for logging is:

```
    def log_event(self, ip_address, status_code):
        """
        Logs each event with timestamp, IP address, and status code in JSON
format.
        Writes each entry as a new line in the log file.
        """
        log_entry = {
            "timestamp": datetime.now().isoformat(),  # Current time in ISOformat
            "ip_address": ip_address,                 # IP address of the request
.                                                       origin
            "status_code": status_code                # Response status code for
.                                                       the event
        }
        # Append log entry as JSON on a new line in the log file
        with open(log_file, 'a') as f:
            f.write(json.dumps(log_entry) + '\n')
```

This function appends each log entry to a JSON file, maintaining a record of server interactions.

- **Response Selection:** The `**select_outcome**` method controls the frequency of response types based on the target distribution. If quotas for each response type are met, the server randomly selects a response based on predefined probabilities.

```
    def select_outcome(self):
        """
        Selects an outcome based on the target distribution.
        Controls response type frequencies over multiple requests to meet the
target distribution.
        """
        # Generate a list of outcomes based on remaining quotas in target
distribution
        remaining_choices = [
```

```
            outcome for outcome, target in target_distribution.items()
            for _ in range(target - distribution_count[outcome])
        ]
        # If all quotas are met, select randomly based on original probabilities
        if not remaining_choices:
            outcome = random.choices(
                ['200', '403', '500', 'timeout'],
                [0.5, 0.2, 0.1, 0.2]
            )[0]
        else:
            # Select outcome from remaining choices to meet exact target
            outcome = random.choice(remaining_choices)
        distribution_count[outcome] += 1  # Update count for selected outcome
        return outcome
```

- **Request Handling:** The `**do_GET**` method handles requests for both `**/getbalance**` and `**/getlogs**`. Depending on the selected outcome, it responds appropriately.
  The server handles requests and logs events. Here's the code snippet for processing GET requests:

```
    def do_GET(self):
        """
        Handles GET requests for both /getbalance and /getlogs routes.
        """
        ip_address = self.client_address[0]  # Retrieve client's IP address

        # Handle the /getbalance endpoint
        if self.path == '/getbalance':
            outcome = self.select_outcome()  # Select response outcome
```

In this snippet, the server identifies the request path and determines the outcome based on predefined probabilities.

2. Client Code (client.py)

The client.py script that I wrote is designed to interact with an unreliable HTTP server implemented in server.py. Its primary function is to send multiple requests to the server's **/getbalance** endpoint and retrieve the corresponding logs from the **/getlogs** endpoint. This report details the code's structure, functionality, and evidence demonstrating its successful operation.

- **Imports and Main Function**: beginning by importing necessary libraries such as sys, requests, json, and time. The main function, main(host, port), is defined to orchestrate the communication with the server using the provided host and port as command-line arguments.

```
import sys  # Module to access command-line arguments
import requests  # Library for making HTTP requests
import json  # Module for handling JSON data
import time  # Module for time-related functions
```

- **Base URL Construction:** The client constructs the base URL based on command-line arguments for the IP address and port number.

```
    # Construct URLs for the /getbalance and /getlogs endpoints
    getbalance_url = f"http://{host}:{port}/getbalance"
    getlogs_url = f"http://{host}:{port}/getlogs"
```

- **GET Requests:** It sends 100 GET requests to the `/getbalance` endpoint, tracking the success and failure of each request. It also handles timeouts and retries.

```
# Making 100 requests to /getbalance to generate logs
    for i in range(total_requests):
        try:
 # sending a GET request to the /getbalance endpoint with a timeout of 10 seconds
            response = requests.get(getbalance_url, timeout=10)
            # checking the response status code
            if response.status_code == 200:
                successful_requests += 1  # Increment the success counter
                print(f"Request {i + 1}: Balance retrieved successfully.")
```

Successful Responses: If a request returns a 200 status code, the success counter is incremented, and a success message is printed.
Error Handling: For 403 and 500 responses, the failure counter is incremented, and appropriate error messages are displayed.

- **Log Retrieval:** After completing the requests, it retrieves logs from the `/getlogs` endpoint.

```
try:
    response = requests.get(getlogs_url, timeout=10)
    ...
except requests.exceptions.RequestException as e:
    ...
```

- **Execution Control:** The script checks for the correct number of command-line arguments and executes the main function if the script is run as the main module.

```
if __name__ == "__main__":
    ...
    main(host, port)
```

3. Log Analysis (analyze_logs.py)

- This script reads the `logfile.json`, counts the occurrences of each status code, and calculates the percentage distribution of responses:
- Data Parsing: It reads each line from the log file, counts occurrences of status codes (200, 403, 500, and timeout), and computes percentages based on the total number of requests logged.

Evidence of Functionality

1. Successful Execution: Running the server and client successfully produced log entries indicating a mix of HTTP responses based on the defined distribution.

```
PS C:\Users\user\INFT-3507\assignment-1> python client.py 127.0.0.1 8080
Request 1: Balance retrieved successfully.
Request 2: Balance retrieved successfully.
Request 3: Balance retrieved successfully.
Request 4: Balance retrieved successfully.
Request 5: Request error: ('Connection aborted.', RemoteDisconnected('Remote end
closed connection without response'))
Request 6: Balance retrieved successfully.
Request 7: Balance retrieved successfully.
Request 8: Error 403.
```

2. Log Output: The `logfile.json` contains entries formatted correctly, showcasing various timestamps and response codes:

```
{"timestamp": "2024-10-27T21:11:21.237353", "ip_address": "127.0.0.1",
"status_code": 200}
{"timestamp": "2024-10-27T21:11:21.771469", "ip_address": "127.0.0.1",
"status_code": 200}
{"timestamp": "2024-10-27T21:11:22.291621", "ip_address": "127.0.0.1",
"status_code": 200}
{"timestamp": "2024-10-27T21:11:22.811538", "ip_address": "127.0.0.1",
"status_code": 200}
{"timestamp": "2024-10-27T21:11:23.333375", "ip_address": "127.0.0.1",
"status_code": "timeout"}
{"timestamp": "2024-10-27T21:11:28.876434", "ip_address": "127.0.0.1",
"status_code": 200}
{"timestamp": "2024-10-27T21:11:29.394824", "ip_address": "127.0.0.1",
"status_code": 200}
{"timestamp": "2024-10-27T21:11:29.898777", "ip_address": "127.0.0.1",
"status_code": 403}
```

3. Response Distribution: Running `analyze_logs.py` post-execution provides insights into the distribution of responses, validating that the server adheres to the targeted distribution:

```
PS C:\Users\user\INFT-3507\assignment-1> python analyze_logs.py
200: 50.00%
403: 20.00%
500: 10.00%
Timeout: 20.00%
```

4. Client Resilience: The client demonstrates resilience by retrying requests that timed out, confirming that the system can handle network unreliability effectively.

Conclusion

The project successfully implements an unreliable HTTP server that simulates a variety of response types based on a defined distribution. The design effectively separates server functionality, client requests, and log analysis, ensuring a modular and maintainable codebase. The testing conducted confirms that the solution meets the project objectives, and the results indicate that the server behaves as expected in simulating unreliable network conditions.