Project Description:

The project aims to evaluate the performance of real-time market data streaming using both client and server implementations. This involves the development of a client application to receive real-time market data from a server and a server application to stream market data to clients.

Client-Server Implementation:

The client-server architecture enables the exchange of real-time market data between the server, which acts as a data source, and multiple clients, which receive and process the streaming data. The client-server model facilitates efficient communication and enables scalability by allowing multiple clients to connect to a single server.

Client Implementation:

The client application is responsible for establishing a WebSocket connection with the server and receiving real-time market data updates. Upon receiving data from the server, the client processes and analyzes the data, extracting relevant information such as symbol names, price changes, and other market metrics. The client may perform additional computations or data visualization tasks based on the received data.

Server Implementation:

The server application acts as a data source, continuously streaming real-time market data obtained from the Binance API to connected clients. It listens for incoming WebSocket connections from clients and broadcasts market data updates

to all connected clients in real-time. The server fetches market data from the Binance API, formats it into a suitable format for transmission to clients, and handles the WebSocket communication with clients.

Test Environment:

- Data Type/Size Streamed: Real-time market data obtained from the Binance API, including symbol names, price changes, and other market metrics. Each message size varies but is typically a few kilobytes in size.

- Hardware Details:
  - CPU: Intel Core i7-8700K (6 cores, 12 threads, 3.70 GHz)
  - RAM: 16 GB DDR4
  - Operating System: Ubuntu 23.04 LTS
  - Number of Cores and Threads Utilized: Both the WebSocket server and client processes are restricted to **run on a single core and a single thread**. Using "**taskset"** to limit CPU affinity: As mentioned earlier, we can use the taskset command to bind each Python process to a specific CPU core.

    Example:

    poyraz@poyraz-linux:~/Desktop/TankX/Testing$ taskset -c 0 python3 test_tornado_server.py

- Network Environment: Both the server and client are running on the same machine within the local network.

- Software Environment:
  - Python 3.8-
  - websockets library (latest version)
  - Tornado framework (latest version)

1) Test Results for Implementation with Websocket library:

Server Performance:
Test duration: Approximately 60.09 seconds
Average CPU usage: Around 19.79%
Average Memory usage: Approximately 7433.59 MB

Client Performance:
Test duration: Approximately 60.02 seconds
Average CPU usage: Around 15.86%
Average Memory usage: Approximately 7486.43 MB
Average Latency: 0.011669926174429393
Throughput: 112.847955998532 messages/second
Key Findings:

Test Duration: Both the server and client ran for approximately 60 seconds, indicating that the test duration was consistent for both.

CPU Usage: The server exhibited slightly higher average CPU usage (19.79%) compared to the client (15.86%). This suggests that the server may have a higher computational load, possibly due to managing multiple connections and processing incoming data.

Memory Usage: Both the server and client had relatively high memory usage, with the client showing a slightly higher average memory usage (7486.43 MB) compared to the server (7433.59 MB). This could be attributed to factors such as data processing and storage requirements.

Overall, the performance metrics indicate that both the server and client are operational within acceptable resource utilization ranges. Further optimizations or adjustments may be considered based on specific requirements or scalability considerations.

Overall, based on these results, the client demonstrates good performance metrics with reasonable CPU and memory usage, low latency, and high throughput. This suggests that the client should be able to scale well and handle additional load efficiently. However, scalability also depends on various factors such as network conditions, server capacity, and the nature of the application workload. Additional testing under varying load conditions may be necessary to fully assess scalability.

2) Test Results for Implementation with Tornado library:

Server Performance:
Test duration: 60.0614869594574 seconds
Average CPU usage: 28.769999999999996
Average Memory usage (MB): 4942.908333333334
Average Latency: 7.5530197660801 78e-05
Throughput: 43.50541640375675 messages/second

Client Performance:
Test duration: 60.036 seconds
Average CPU usage: 17.617%
Average Memory usage: 4598.153 MB
Average Latency: 37.200 milliseconds
Throughput: 95.144 messages/second

Here are some observations and interpretations:
Test Duration: The client ran for approximately 60 seconds, as intended.

Average CPU Usage: The average CPU usage during the test was 17.62%, indicating that the client's CPU usage was relatively low.

Average Memory Usage: The average memory usage during the test was 4598.153 MB, indicating the average amount of memory the client consumed.

Average Latency: The average latency observed during the test was 37.20 milliseconds. Latency represents the time taken for a message to travel from the client to the server and back. Lower latency is generally desirable, as it indicates faster communication between the client and server.

Throughput: The throughput of 95.144 messages/second indicates the rate at which the client processed messages from the server. A higher throughput indicates better performance in handling incoming messages.

Overall, these results suggest that the client performed reasonably well, with low CPU usage, moderate memory usage, relatively low latency, and a decent throughput rate. However, further optimization may be possible depending on specific performance requirements or constraints.

Results:

Based on the provided test results for the implementations using the websockets library and the Tornado library, we can make the following comparisons:

Test Duration:
Both implementations had a similar test duration of approximately 60 seconds, indicating that they were subjected to the same testing period.

Average CPU Usage:
The implementation with the websockets library showed lower average CPU usage, around 19.79%, compared to Tornado, which had an average CPU usage of approximately 28.77%. This suggests that the websockets implementation may be more efficient in terms of CPU utilization.

Average Memory Usage:
The Tornado implementation exhibited lower average memory usage, with approximately 4598.153 MB, compared to the websockets implementation, which had an average memory usage of around 7433.59 MB. This indicates that the Tornado implementation may be more memory-efficient.

Average Latency:
The Tornado implementation reported an average latency of approximately 37.200 milliseconds, while the websockets

implementation had an average latency of 0.011669926174429393 milliseconds. Lower latency is generally desirable as it indicates faster response times.

Throughput:
The websockets implementation achieved a higher throughput of approximately 112.85 messages/second compared to Tornado, which had a throughput of around 43.505 messages/second. Higher throughput suggests better performance in handling message throughput.

Overall Comparison:
The choice between the two libraries may depend on the specific requirements of the application. If low latency and high throughput are critical factors, the websockets library may be preferred. However, if efficient CPU and memory usage are more important, the Tornado library could be a better option. Overall, both libraries have their strengths and may be suitable for different use cases depending on the specific performance requirements and constraints of the application.

Prepared By:
Poyraz Özmen
poyrazozmen98@gmail.com
https://github.com/Poyraz-Ozmen/Python-Web-Socket-Analysis