Node.js Server (Nuoyu Yang):

As an asynchronous event-driven JavaScript runtime, Node.js is designed to build scalable network applications. In my part, I try to use Node.js to build a server and compare the performance with server in Java and other programming languages.

1. Asynchronous JavaScript:

JavaScript is a single-threaded programming language which means only one thing can happen at a time. That is, the JavaScript engine can only process one statement at a time in a single thread.

While the single-threaded languages simplify writing code because you don’t have to worry about concurrency issues, this also means you can’t perform long operations such as network access without blocking the main thread.

Imagine requesting some data from an API. Depending upon the situation the server might take some time to process the request while blocking the main thread making the web page unresponsive.

That’s where asynchronous JavaScript comes into play. Using asynchronous JavaScript (such as callbacks, promises, and async/await), you can perform long network requests without blocking the main thread.

2. Node.js server:

We need to import “express” framework to simplify the steps to implement “get” and “post” operations. To parse contents from request body of HTTP and operate MySql database, we also need to import “body-parser” middleware and “mysql” driver. To implement asynchronous operations, we need to use call back functions in our server.

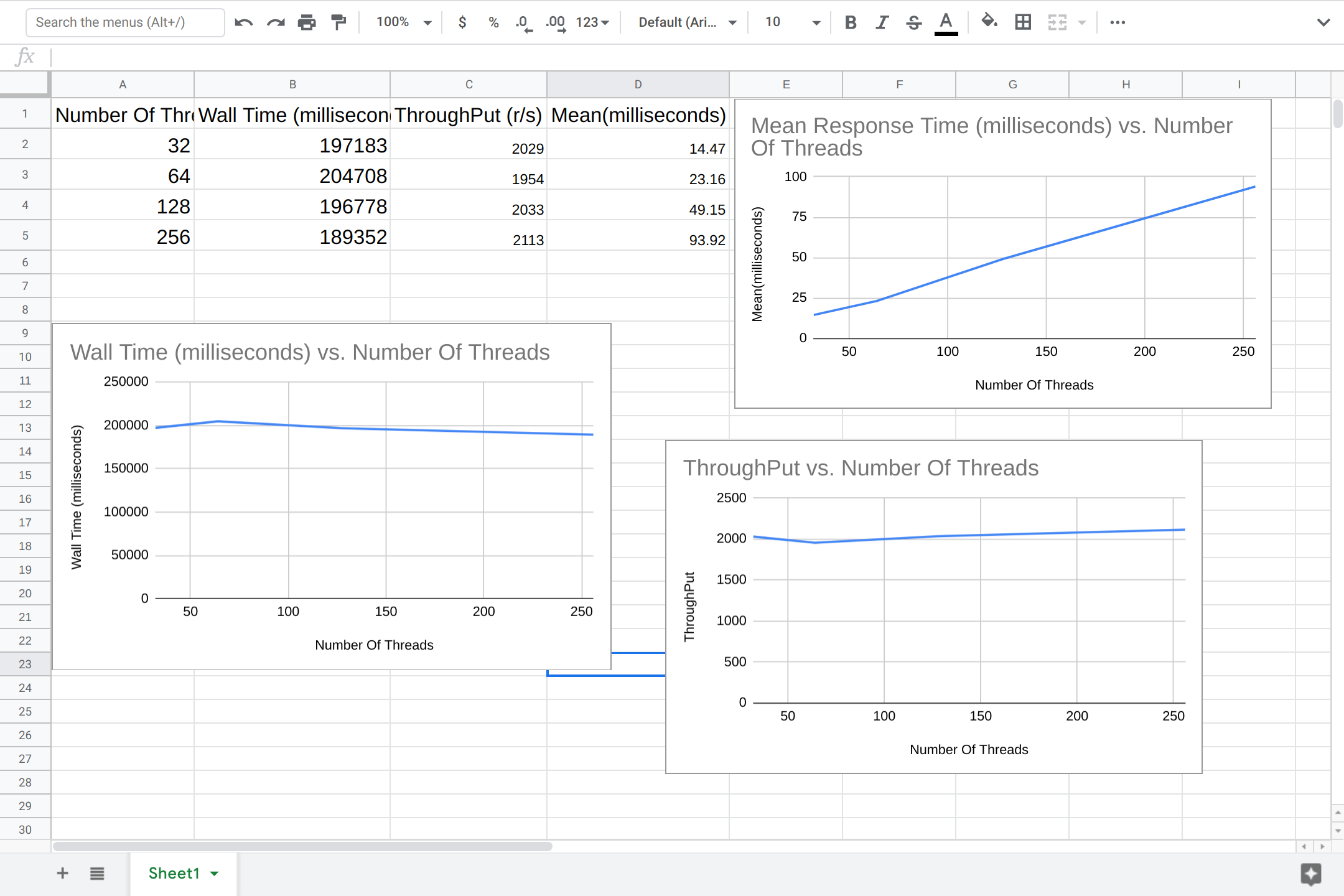
3. Prediction:

As a single thread asynchronous server, we think that node.js server may have a great performance when the number of requests is relatively small. That means with the threads of client increasing, the wall time and throughput may keep steady or be changed slightly and the mean latency will increase.

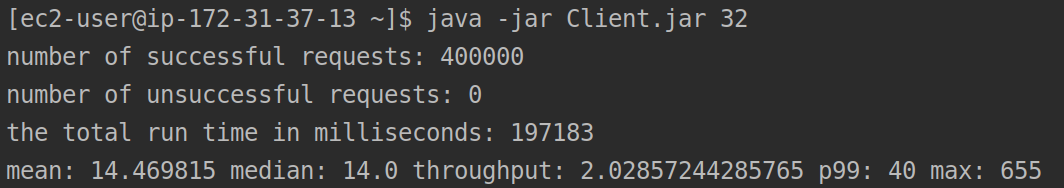
4.Result:

The result of Node.js server is as follows:

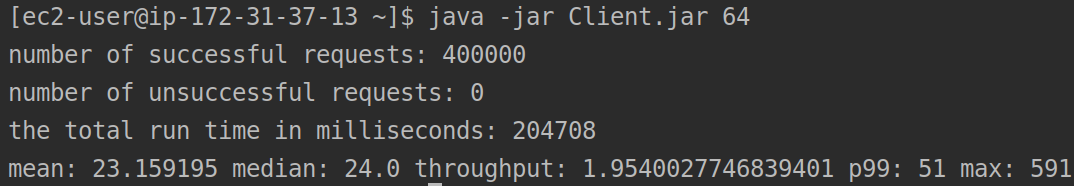
Single Server:



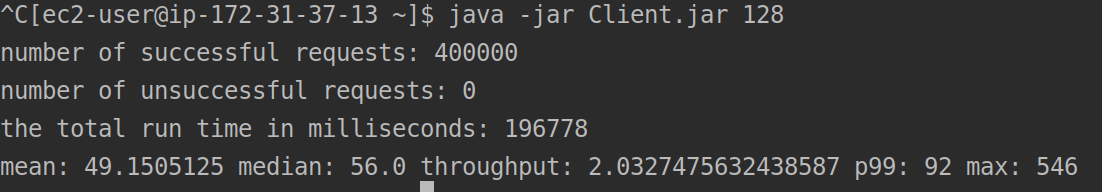
32 threads:



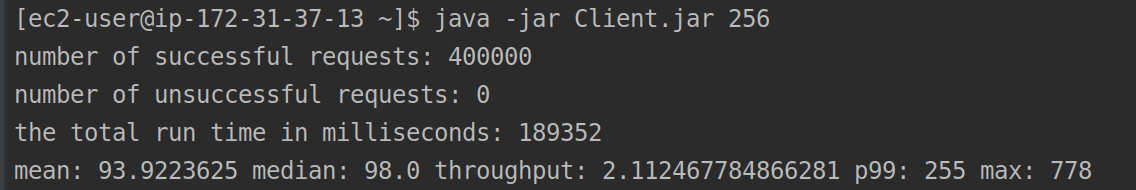
64 threads:



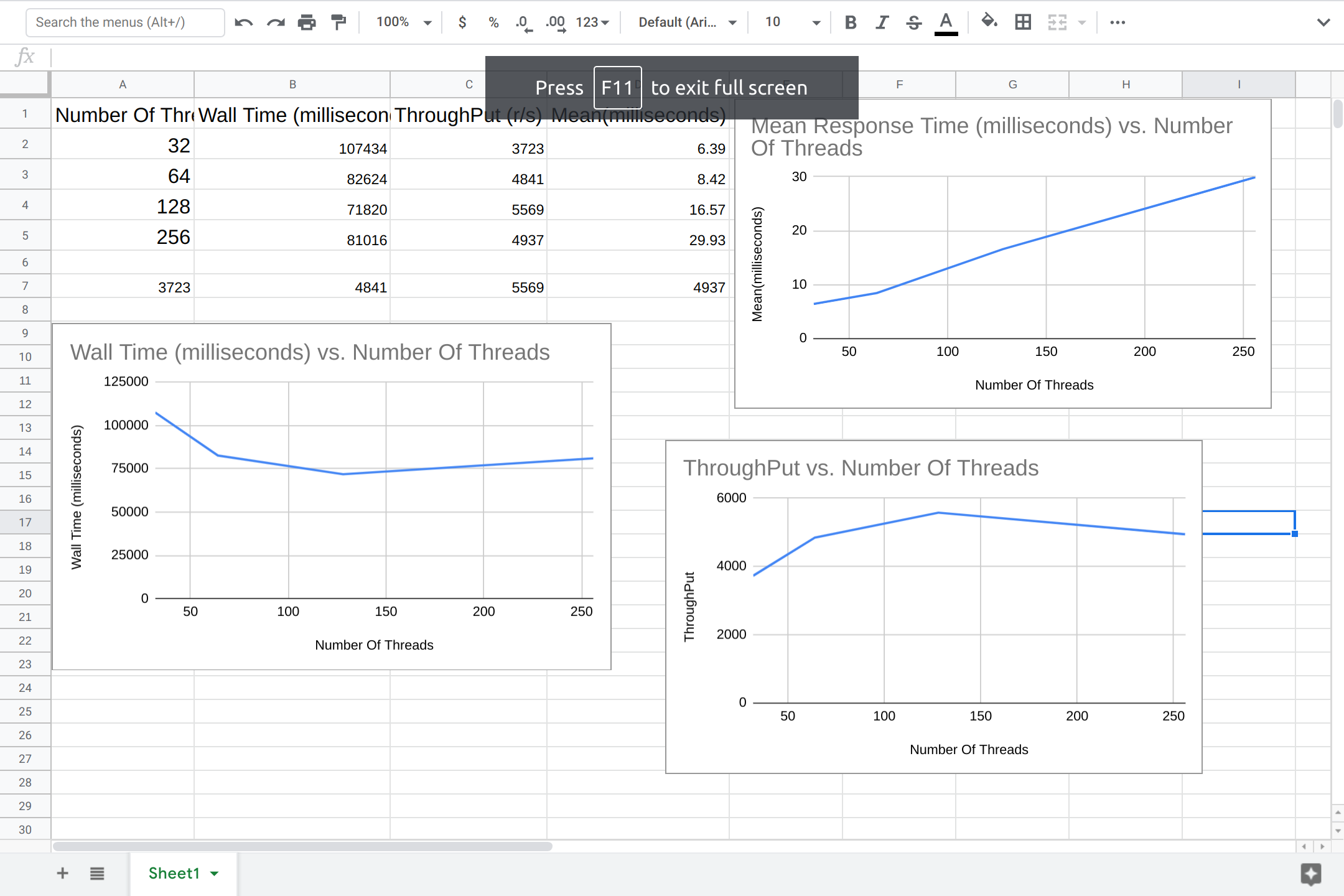
128 threads:



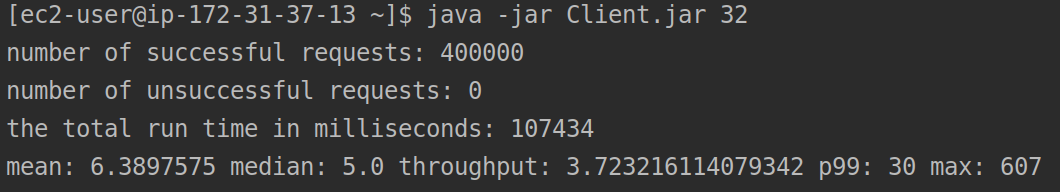
256 threads:



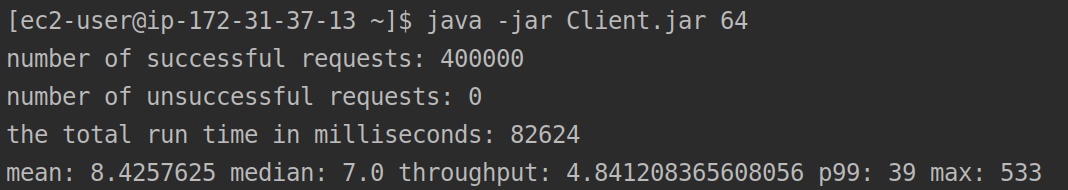
Load Balancer (4 Layers):



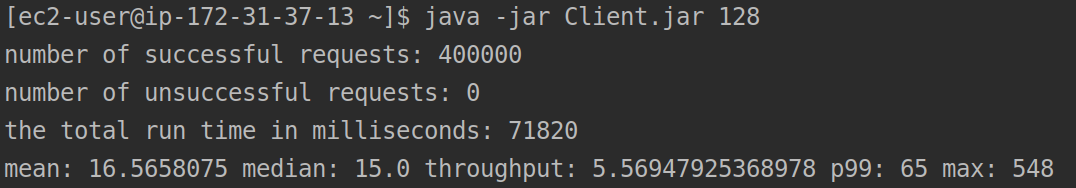
32 threads:



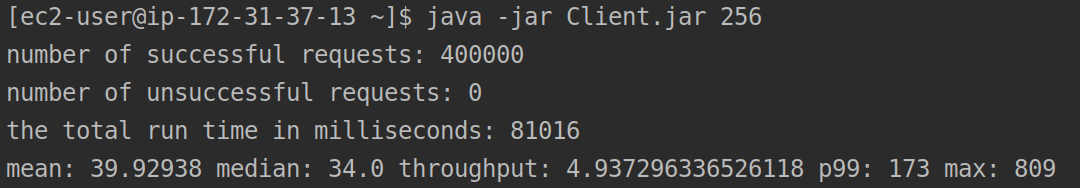
64 threads:



128 threads:



256 threads:



The results show that, in single server, as we expected, the wall time and throughput keep steady and mean latency increase when number of client threads increase from 32 to 256.

In ELB with 4 servers, however, the wall time shows a curve with relatively higher on both sides and lower in middle. A possible reason is that the best performance of single Node.js server is on the ¼ of total requests when number of threads is around 128.