Performance Test Report of CPU Intensive REST API on AWS t2.micro Wildfly Server

Server-side Design

Server Configuration

1. AWS t2.micro Cloud Server
    - CPU: Single Core, Intel Xeon E5-2670v2 @2.50GHz
    - Memory: 1GB
    - Hard Disk: General SSD 30GB
    - Public IP: 52.42.56.228
2. Software Platform
    - Windows Server 2012 R2
    - Wildlfly 10.0.0
    - Apache Maven 3.3.0
    - JDK 1.8.0

REST API service

1. Implementation based JBoss ReasEasy, which is a `JAX_RS 2.0` Implementation provided by JBoss AS and bundled with Wildfly.
2. Four HTTP GET APIs are implemented to provide a calculator service, in such forms:

```
http://52.42.56.228:8080/add?operand1=10&operand2=20
http://52.42.56.228:8080/substract?operand1=10&operand2=20
http://52.42.56.228:8080/multiply?operand1=10&operand2=20
http://52.42.56.228:8080/divide?operand1=10&operand2=20
```

3. To get a `HTTP 200` response, the APIs are case sensitive.
4. The successful respond message is plain text. The message is `error input` if no

valid operands are given. Valid operands are integers, floating point numbers in plain form or scientific form like `3.14`, `2E10`, `1.0e5`.
5. Project is managed and deployed using Maven to root folder of wildfly server.
6. The source code is called `CalculatorREST.java` which can be found in the folder `/calculator/src/main/java/com/sensorhound/calculator` with this report.

Client-side Design

Client test principles

1. To fully stress the server in concurrency requests, the client should simulate many users requesting the API simultaneously.
2. The users are simulated by multiple threads. Each thread connects to the server and requests repeatedly. To parallel the threads, the waiting of response must be non-blocked.
3. Above principles can be realized using `javax.ws.rs.client` in RestEasy client library and `java.util.concurrent`.

Client program user guide (Same as `Readme.txt` in `/calculatorClient/`)

1. The client can accept five optional parameters.
   (1) Number of users: an integer (A user is a standalone thread)
   (2) Requests per user: an integer
   The above two parameters should be provided together. Otherwise, 10 users and 10 requests are used by default.
   (3) Operator: `+`, `-`, `*`, `/`, `all`
   Operator is `all` if not provided. `all` means all the four APIs will be tested randomly.
   (4) Operand1: either integer or float point number
   (5) Operand2: either integer or float point number
   The two operands should be provided together. Otherwise, the operands will be a pair of random integers for each user.

   In fact, the best way to use the client is to provide only the first two parameters. The client will test all APIs with random operands automatically.

2. Build and execute the client
   (1) You need to have JDK 1.8.0 and Maven (better newest version) on your computer and configured correctly.

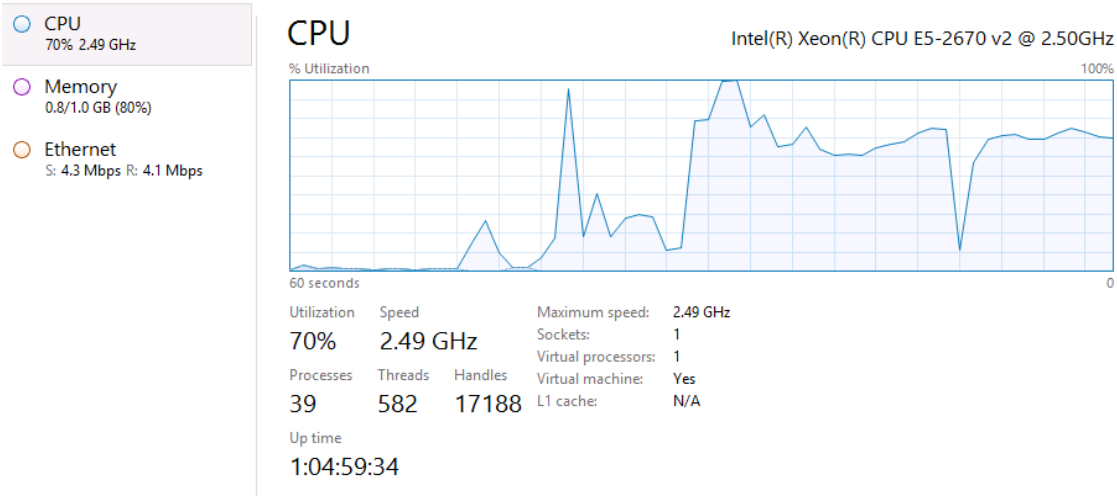
(2) Go to `/calculatorClient/` in command line.
(3) Execute `mvn clean compile`
(4) Execute `mvn exec:java -Dexec.mainClass="com.sensorhound.client.CalculatorClient"` to run the default 10 x 10 test.
(5) If succeed, you will find a `log10x10.csv` file in the current folder `/calculatorClient/`.
(6) Try larger parameters such as:
`mvn exec:java -Dexec.mainClass="com.sensorhound.client.CalculatorClient" -Dexec.args="1000 1000"`
(7) A new empty `logMxN.csv` will be created each time you execute the client. So if you want to keep file after execution, you must rename it or move it away.

Test Results

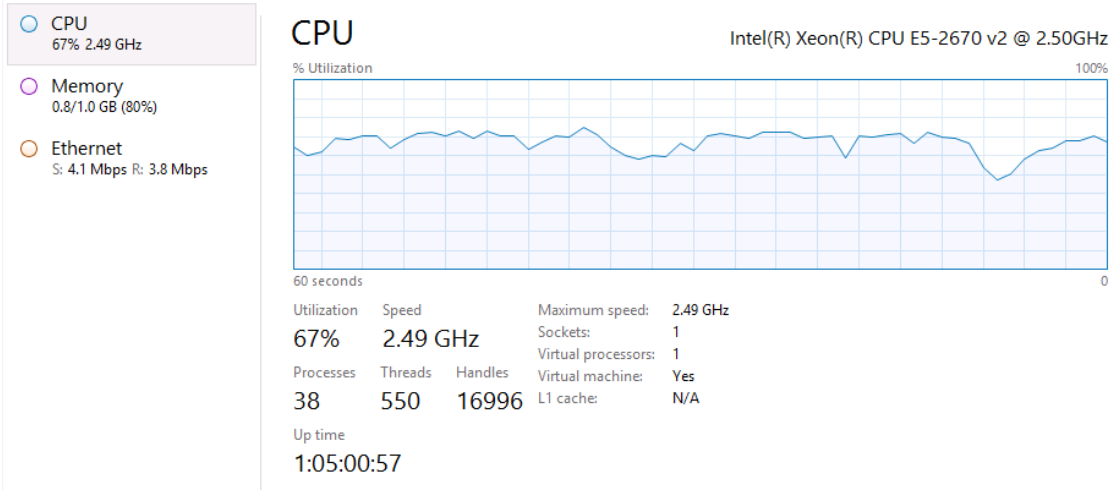
Test 1: 1000 x 1000

The first test is to simulate 1000 user requesting 1000 times.
The log file is `/report/TestData/log1000x1000.csv`.
The test takes about 495s.
(1) Response Delay

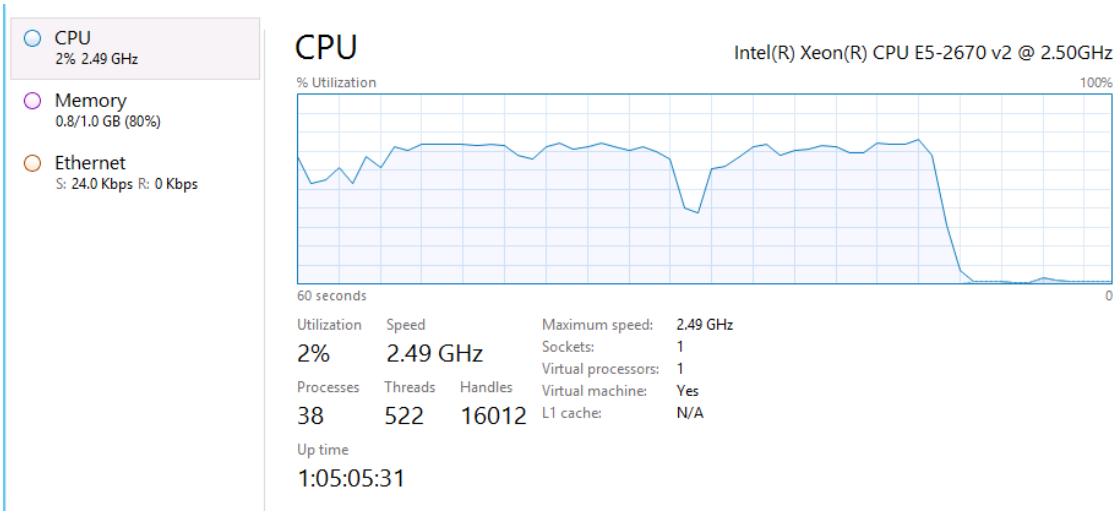
- The first 10s:
  When 1000 connections are invoked at the same time, the server has a large response delay.
  The first request of most threads are slightly above 1000ms. But no one takes longer than 20000ms which is the threshold of `TIMEOUT` in the client.
  The server CPU usage ramps to 100% in a short time.



- The middle 470s:
  In steady time, about 430ms response delay and the server has about 70% CPU usage.



- The last 15s:
  Threads finished their requests and stops one by one. The response delay drops gradually to 80ms, which represent the best the server can have. CPU usage drops down to 2%.

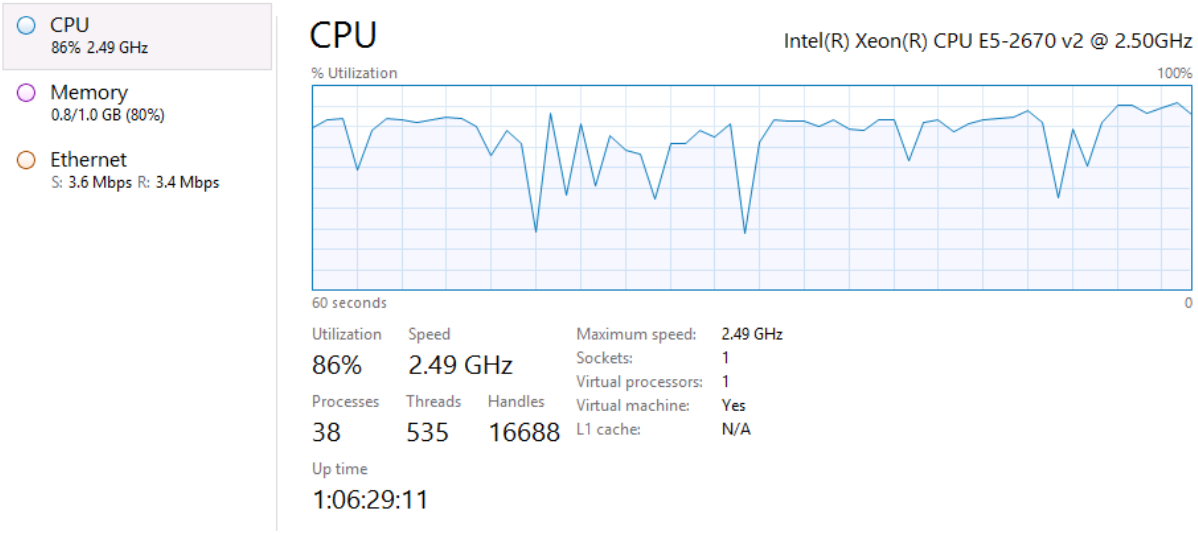


(2) QPS:
In about 495s, 1000,000 requests are tested and successfully responded. So the `average QPS` = 2020, which is approximately equal to the real time QPS in the second half time.
The column `Success QPS` in the file is computed based on successful requests before.
Which represent the servers concurrency response ability.

(3) Error type and Error Ratio
Two types of error are recorded, one is `Time Out` (large than 20s), the other is `Unable to invoke request`. In this case no error occurred.

Test 2: 2000 x 100

The second test is to simulate 2000 user requesting 100 times.
The log file is `/report/TestData/log2000x100.csv` .
The test takes about 116s.
The CPU usage is about 85% in steady state.



(1) Response Delay

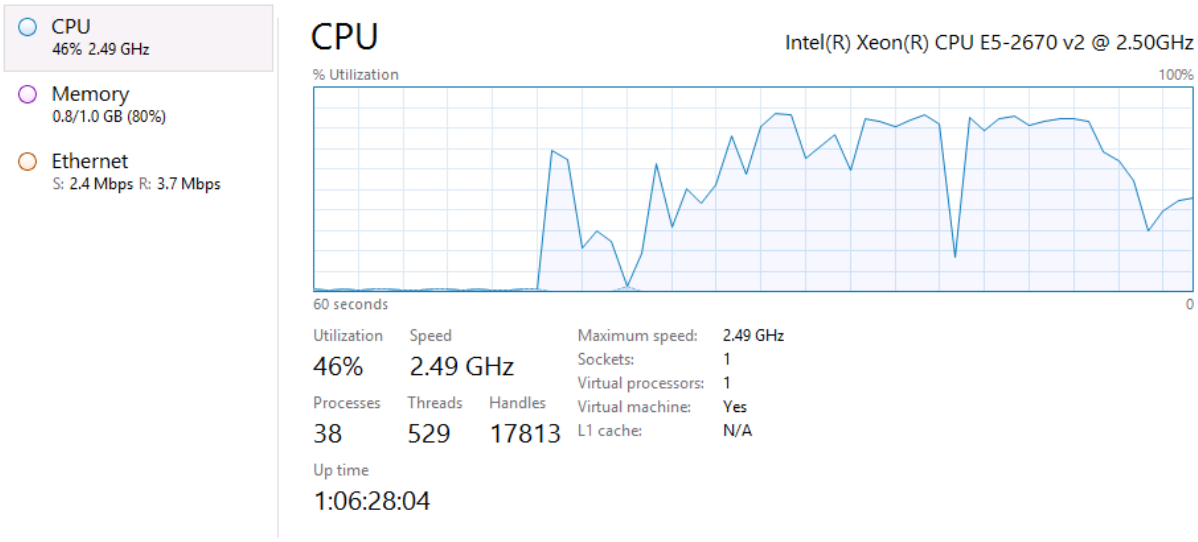
The response delay is large (2±1 s) and undulate heavily.

(2) QPS:
Even thought there are more threads, the QPS is smaller due to larger delay.
`Avg QPS` = 199241/116 = 1717

(3) Error type and Error Ratio
Only 11 thread met response `Time Out` (>20s) . No other errors.
`Error Ratio` is 11/200000.

Test 3: 10000 x 100

The last test is much more threads.
The log file is `/report/TestData/log10000x100.csv` .
My client PC goes to 100% so it contributes to the delay and failure too.
The test takes about 119s. The Server CPU burden goes up to 90% but highly unstable.



(1) Response Delay
The highest delay recorded is larger than 20s. This is due to Client side CPU burden in creating threads. After all threads came to run, there are still some delay > 10s.

(2) QPS
`Avg QPS` = 41531/119 = 349. Much lower than less threads.

(3) Error Type and Error Ratio
There are `Time Out` and `Unable to invoke request` errors.
In fact 9729 Threads met error in 41531 requests.
Error Ratio = 23.4%

Bottle Neck Analysis

The CPU speed of the server

Our REST API is CPU intensive only. The data transmitted per request is very little. It is totally stateless, with no database query or disk IO.
The only limitation of the server is the CPU ability. As we can see from the images above, when the server CPU usage rises up to 100%, the client response `Time Out` or can not open new connection.

The network speed and distance between the server and the clients

Besides CPU response time, the response delay in http request also comes from the distance (number of routers) between the server and clients.
In addition the server network bandwidth also determines the speed it receives and sends data.

The JVM performance and garbage collection

JVM has automatic garbage collection mechanism. But when providing too many services concurrently, the JVM will have to collect garbage frequently for the used objects.

Optimization

For our simple calculator API

Based on the bottle necks analyzed above, there are three things we can do to improve the high concurrency performance of the Calculator API.

- Distributed Multi-core Servers
- Deploy servers near the target clients
- Reuse objects to reduce garbage collection.

In fact we still need a server to load balance the requests. Compared to our simple API, the load balancing server has more work to do.

For REST Services in Reality.

In reality, a rest services may have such abilities:

- Return large size static data, such as HTML, images, PDF, videos.
  Optimize: Cache the data on CDN, which are distributed servers near users.

- Heavy reading of database on different servers
  Optimize: Cache the database in RAM (use Memcached or Redis)

- Heavy transaction business of database on remote servers
  Optimize Message Queuing or Stored Procedure for database operation