Assignment 9 Design Documentation

Introduction:

In this assignment, we tried following features: **Markdown rendering, load test, the database interface and HTTPS encryption**. Also, our web server support **HTTP/1.1 (keep-alive)**. We successfully finished Markdown rendering, load test and HTTPS encryption, but did not finish the database interface. But there is still something to demo for the database interface and more details will be in that section.

1.Markdown (success):

Demo link: http://35.166.145.87/markdown/README.md

Markdown is implemented with the help of Cpp-Markdown Markdown text-to-HTML translator. We treat md file in a very similar way of handling other types of file in static handler. When static handler detecting a md suffix, the only difference is that instead of sending md files directly into output buffer, the handler will input the Markdown text into the translator and add the rendered output into the body of response. Also, since Cpp-Markdown is licensed under MIT, we also added MIT license in our repository. Even though this project is under private repository, adding proper license could prevent some trouble for potential future release.

2.Database Interface (tried and has something to demo):

Demo link: http://35.166.145.87/database/index.html

The way we chose to implement the database interface is to use html as the interface for user and web server and then embed php language inside html to function as the interface for web server to database.

The database we installed is mysql.

The topic for database we chose is Soccer Land, which enables user to add soccer player/coach/team information and also query soccer player/team information.

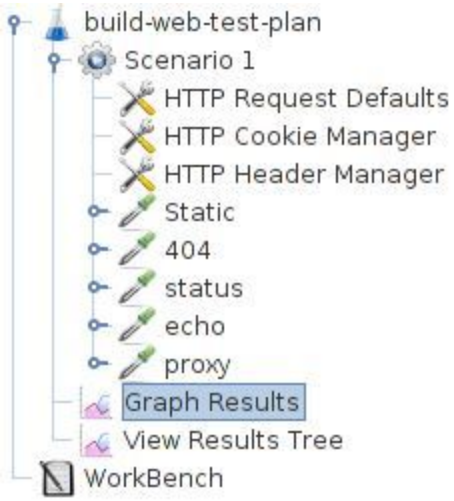
Also, we add bootstrap for the front end.

Our database interface worked well on apache server or server that supports php. However, we did lots of research and could not find a way to make our c++ web server supports php. As a result, only the interface with php code displayed can be demonstrated in above link.

3.Load Test (success):

We chose to use JMeter as the load test tool for our web server. JMeter has many features, one is the ability to load and performance test the HTTP web server. After downloading and installing the JMeter application, we followed the Web Test Plan instructions (http://jmeter.apache.org/usermanual/build-web-test-plan.html) to do the load test and results are shown in following graphs. Each user will send http requests to echo handler, static handler, status handler, proxy handler and not found handler.

Our layout of test plan in JMeter:



Key labels of graphs:
Blue line: average response time.
Purple line: median response time.
Green line: throughput (# of responses/minute)

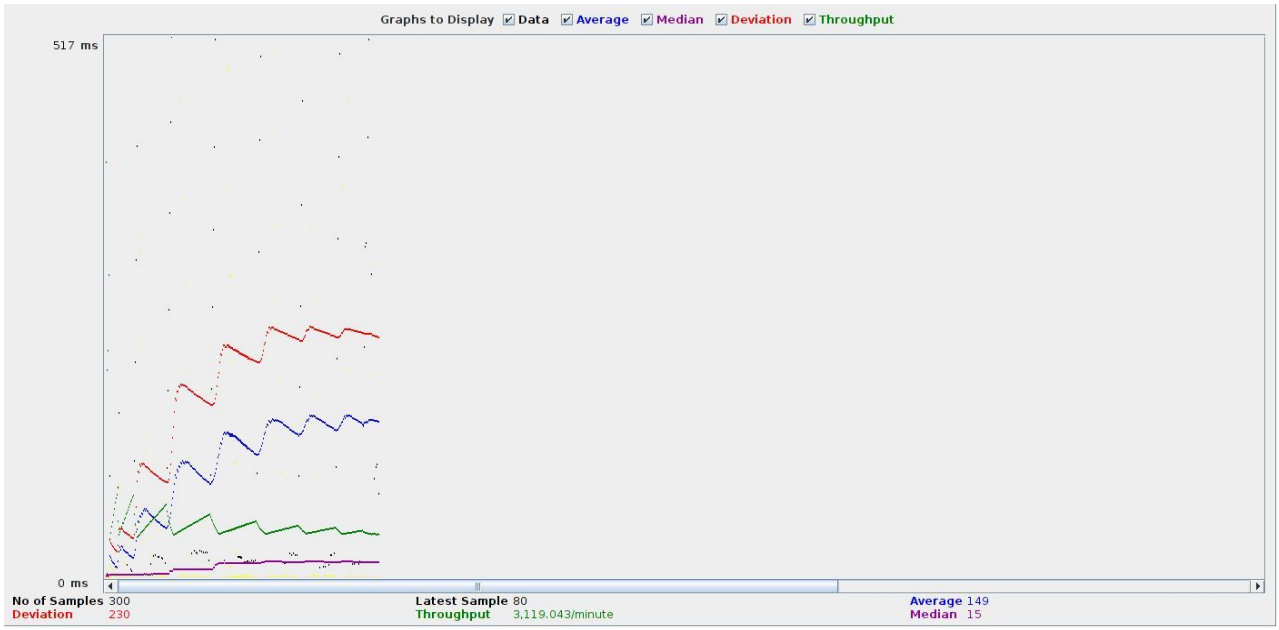


Figure 1. Load test of 10 users.

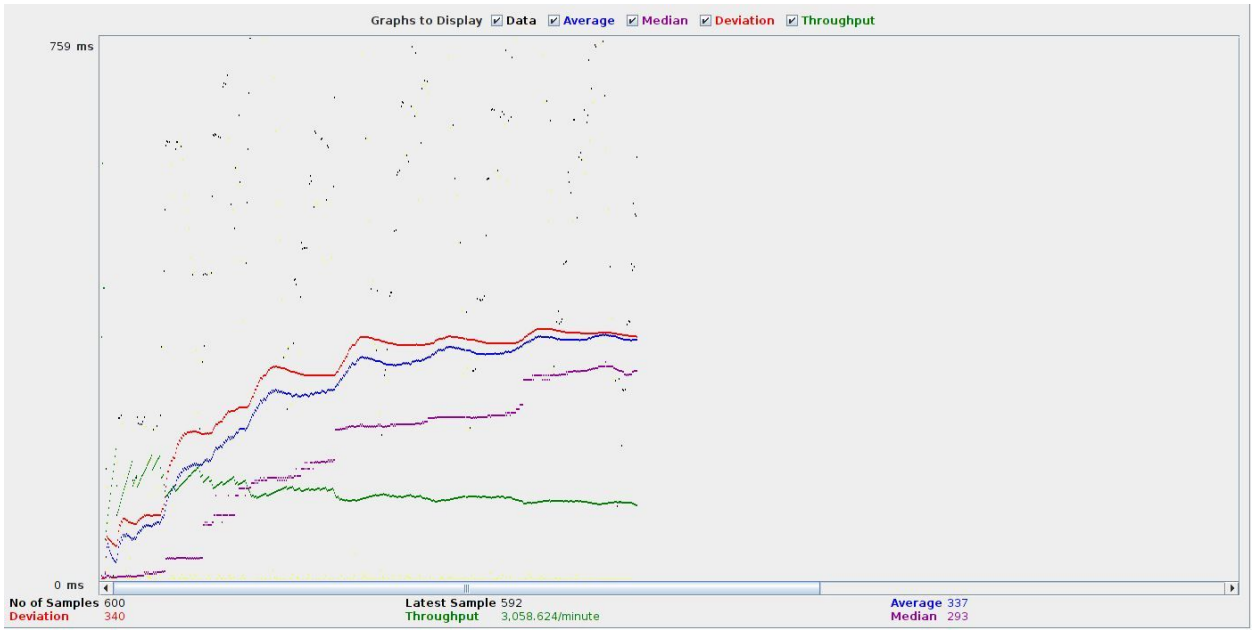


Figure 2. Load test of 20 users.

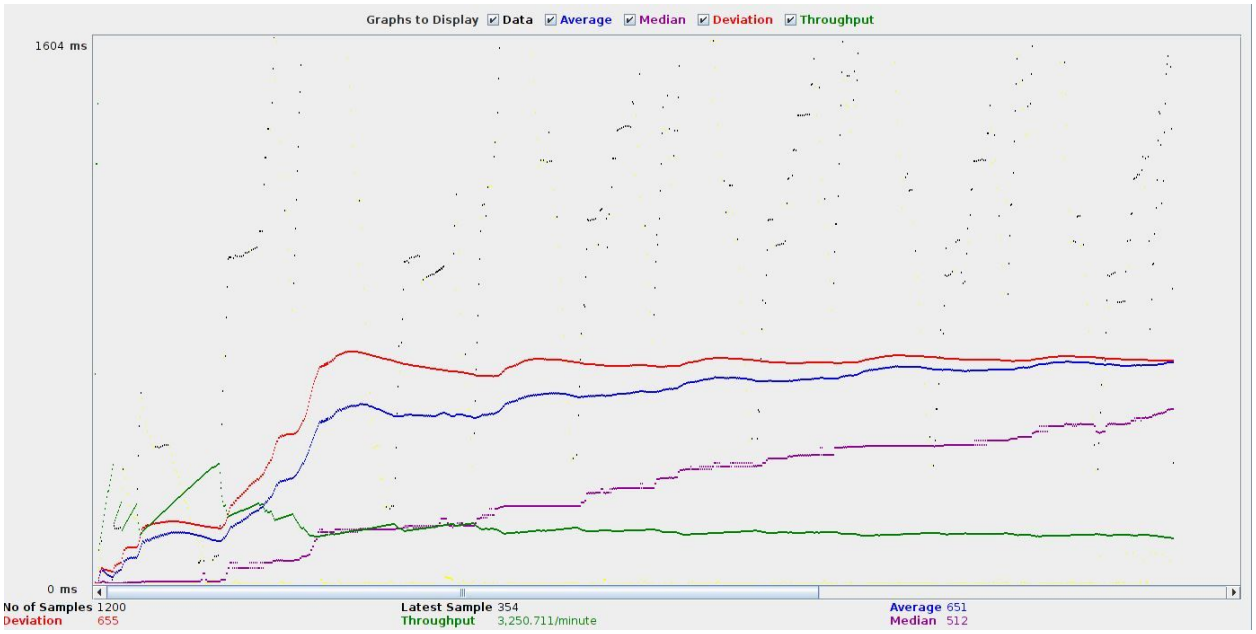


Figure 3. Load test of 40 users.

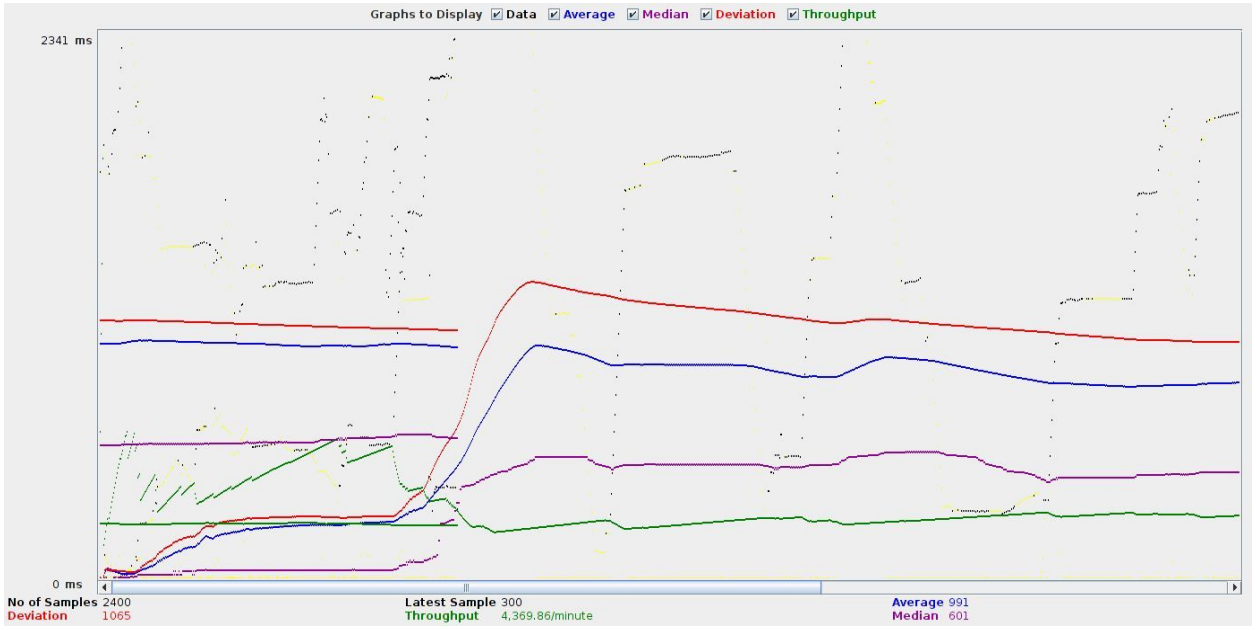


Figure 4. Load test of 80 users.

From figure 1 to 4, as number of users increases, average response time increases from 149ms to 991ms and median response time increases from 15ms to 601ms. These response times are still not bad. And our server finishes all request without failure.

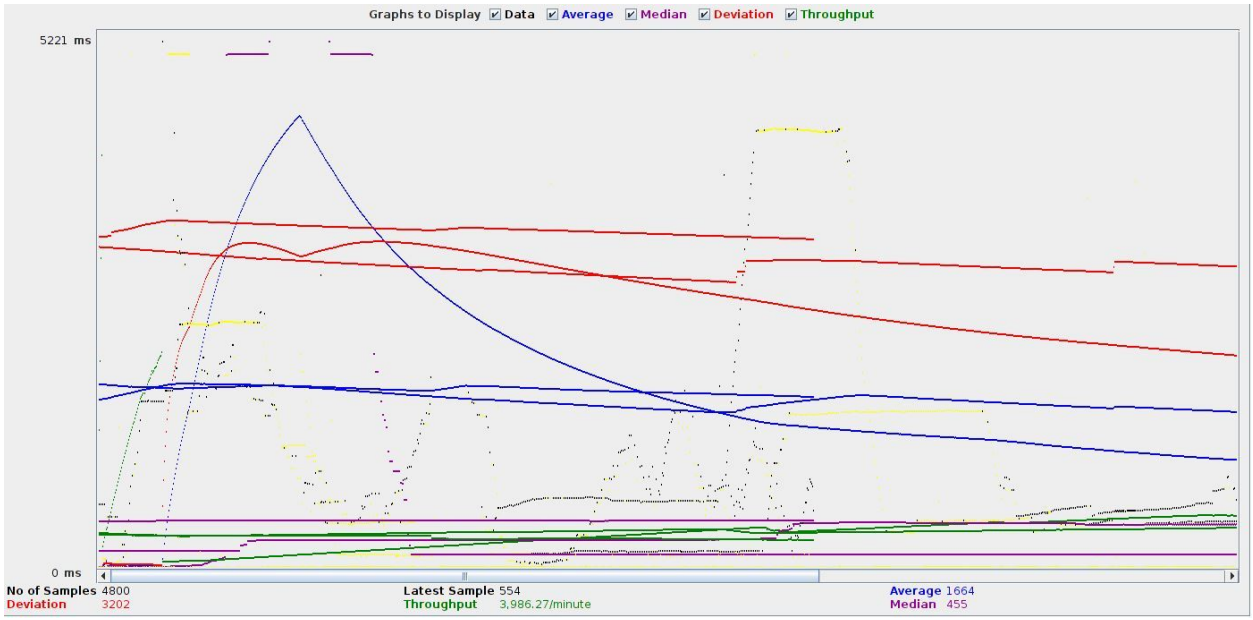


Figure 5. Load test of 160 users.

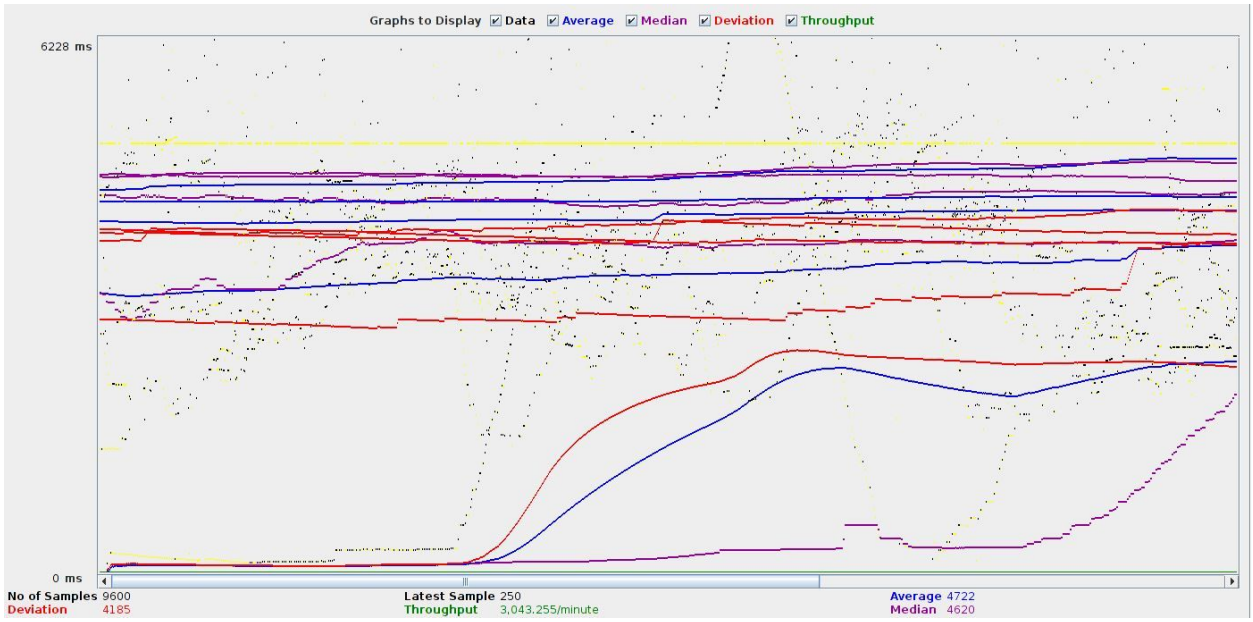


Figure 6. Load test of 320 users.

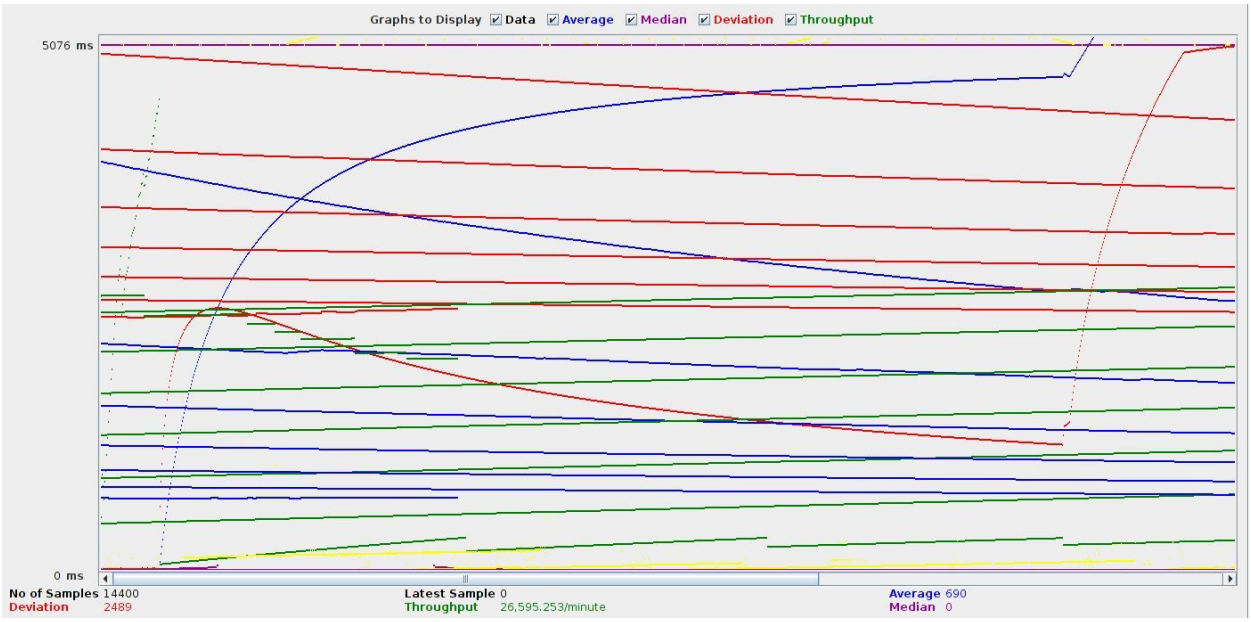


Figure 7. Load test of 480 users.

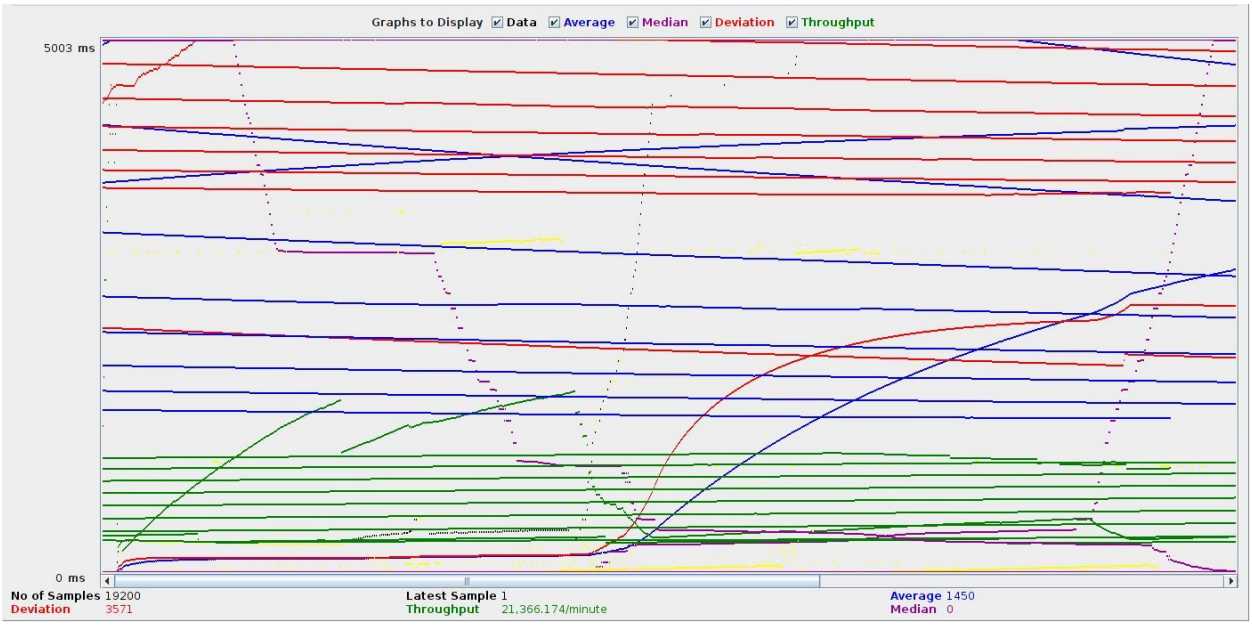


Figure 8. Load test of 640 users.

Our server still finishes all requests without failure under load test of 160 users (figure 5) and 320 users (figure 6). However, the performance under load of 320 users becomes dramatically terrible. As we can see from the figure 5, the average response time is 4722ms and the median response time is 4820ms, which will give users obvious feeling of delay of response. Also, the throughput becomes much less than that of load under 160 users or load under less than 160 users.

As number of users become 480, our servers shut down during middle of test and only finished part of requests. Same thing happened to load test under 640 users. We can see this from the horrible figure 7 and figure 8.

In conclusion, our web server works perfectly for 160 users or less, and still works but with terrible performance for 320 users. When it comes to 480 users or more, our servers will fail during the plenty of requests.

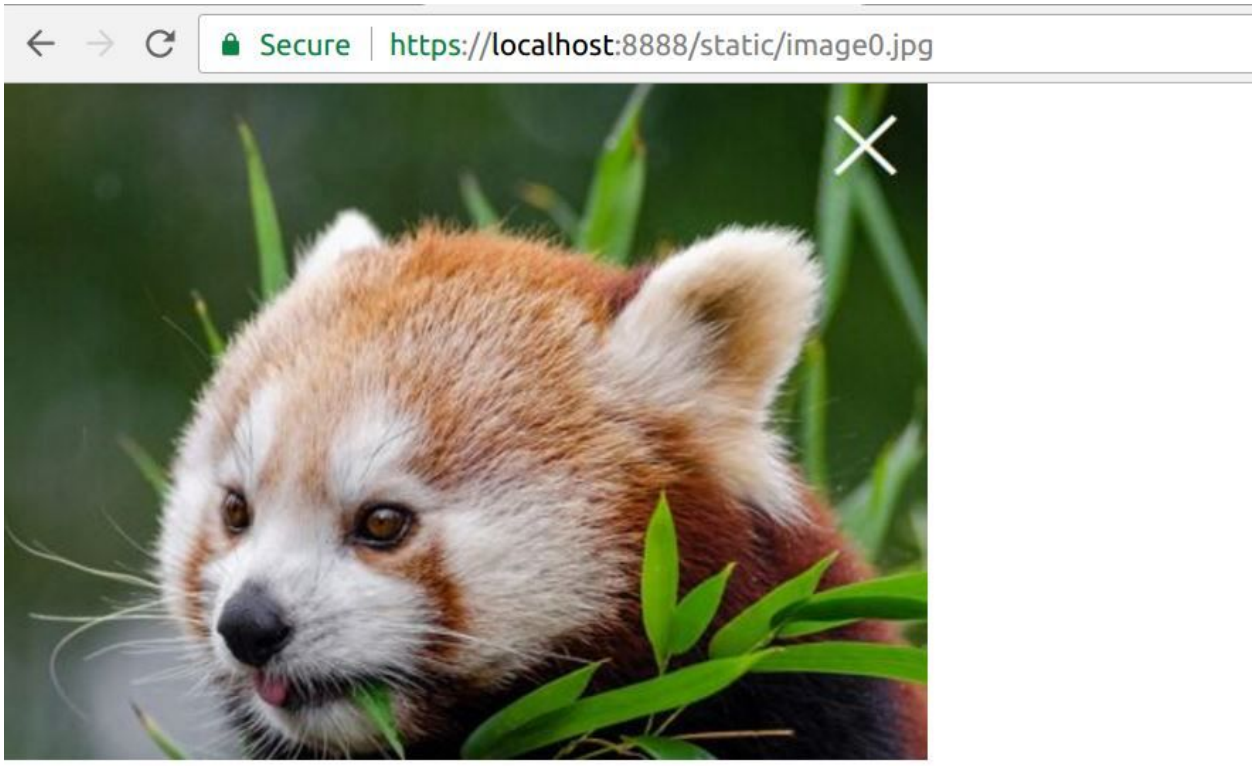
4.HTTPS Encryption (success on localhost):

Demo:
make webserver_https
./webserver_https demo_config
shuang94 (for “Enter PEM pass phrase”)
link: https://localhost:8888/static/image0.jpg

We implement HTTPS using boost’s ssl library. The connection uses an ssl stream socket instead of the plain tcp socket. In the process of accepting a connection, a handshake was added. Private key and certificate are generated using openssl.

The demo is readily working on local machine. However, when we try to deploy this on aws, we found out that the busybox docker image does not include libssl.so, which is necessary to link for ssl encryption. As a result, the we’ll demo it on localhost.

The green https lock can be seen in the upper left corner of following image:



5.HTTP/1.1

Demo:
curl -Iv http://35.166.145.87/static/index.html 2>&1 | grep -i '#0'

Our web server also supports HTTP/1.1 which keeps connection alive.