Performance test report

Object under test

Simple web server written on Go. It is deployed on localhost, listens TCP port 8080, and has the only handler that gives http 200 ‘Hello’ to every request for HTTP request on url localhost:8080/hello.

Host characteristics

IntelCore i5-1145G7 @2.60GHz x 8
Memory 16Gb
Disk SSD 512 Gb
OS Ubuntu 20.04 64-bit

Load generator

Phantom load generator with Yandex.Tank.
The load generator was working on the same host with the object under test.
Results were uploaded to Overload web service for better visualization
https://overload.yandex.net/

Test objective

To find saturation point of web server under test

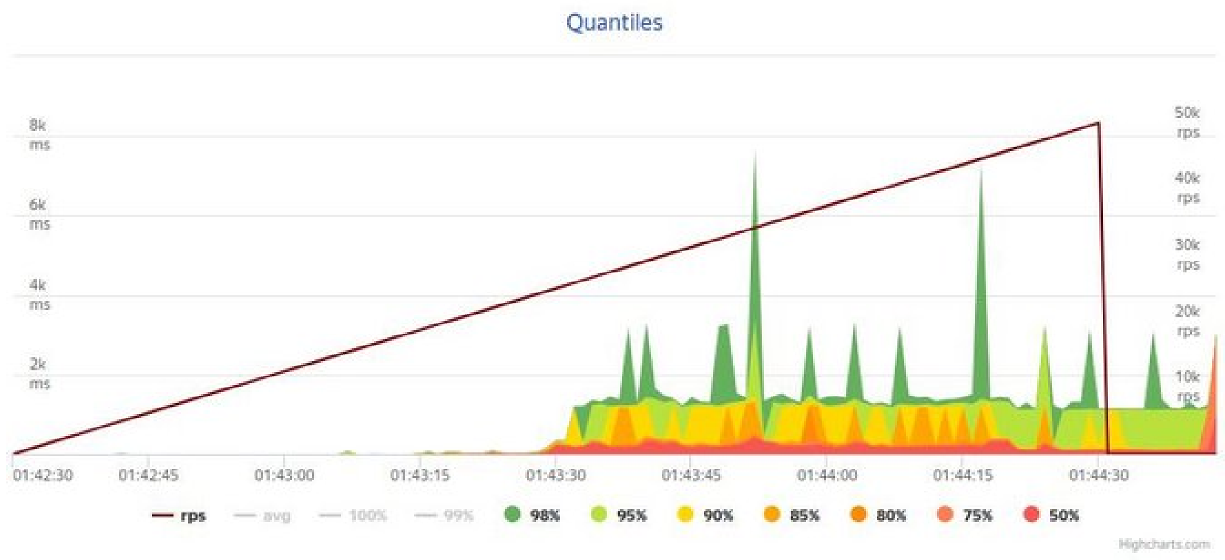
Test design and results

First test is performed with linearly increasing load from 1 up to 50k rps.
Test configuration: https://github.com/szypulka/pt-as/blob/master/load/test1_conf.yaml
Test results: https://overload.yandex.net/514441

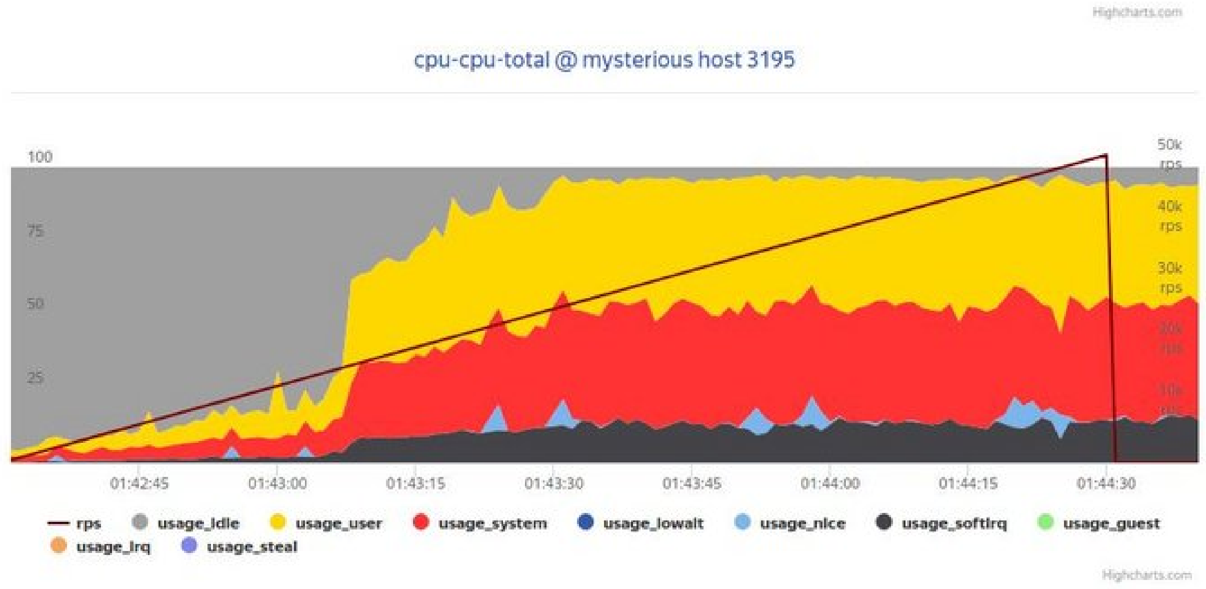
Second test is performed with 100 parallel instances for 5 minutes.
Test configuration: https://github.com/szypulka/pt-as/blob/master/load/test1_conf.yaml
Test results: https://overload.yandex.net/514437

Results analysis

The first test shows the behavior of SUT under permanently increasing load and allows us to detect the saturation point where the system under test behavior changes: response time begins to grow.
On this chart we can see that response times grow significantly at the load of approximately 21k rps.Then RT come to plateau and service responds under the heavy load up to 50k rps with the acceptable level of degradation.



The host monitoring charts show us the bottleneck: it's cpu usage



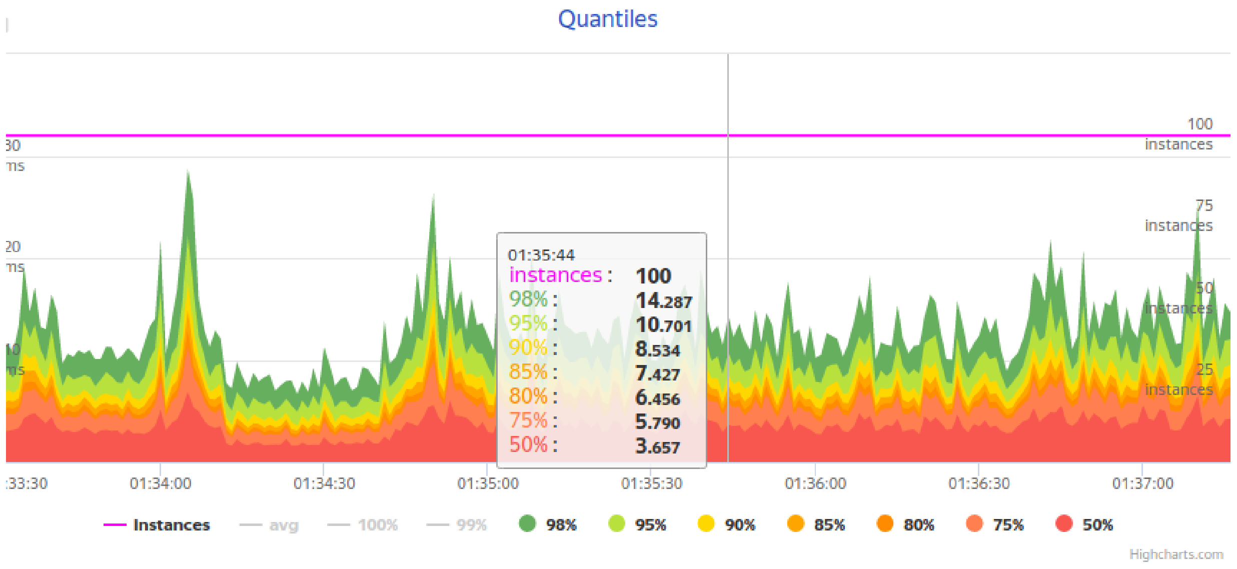
The other problem is found in server logs:

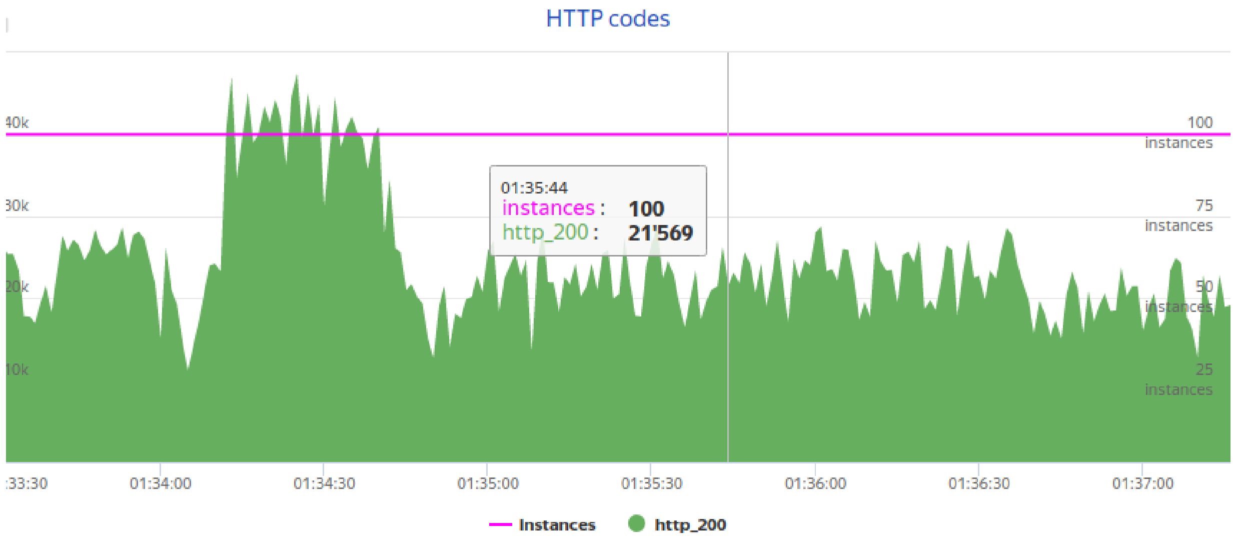
```
http: Accept error: accept tcp [::]:8080: accept4: too many open
files; retrying in 5ms
```

It’s the lack of file descriptors – the usage of FD grows linearly with the number of open connections, and by default it is not very high (1024 in our case).

The second test shows the behavior of SUT under permanent load. Load generator here is working with a limited number of 100 instances (we can consider them as VUs, with certain amount of convention – there are no sleeps here etc.).

Here the system under test comes to saturation and performs as fast as it can. On the screenshot we can see that it is possible to achieve about 21k rps with the appropriate response times (95% responses are within 10ms).





The possible solutions for increasing server performance:

- Tuning the host where SUT is located (increase file descriptors limit)
- Scaling – put several web servers under the balancer
- Add additional CPUs to web server host

Additional details regarding the test

As for the tool selected for the assignment, I've just taken the first open source tool I'm familiar with and that certainly was able to do the task. It of course can be solved with all performance test tools, but the easy setup and illustrative charts made my choice.

I've tried to do the same with k6 (you can see the skeleton of test scenario in my repository), but the default limit on 50 users is definitely not enough to show the saturation point of a simple Go web server.

As for the other questions:

The response time consists of several parts, namely initializing and opening the connection to server (can be done for the several requests at once, i.e. when Keep-Alive setting is used), network part (the time for net transfer of the request/response packages) and latency itself (the time for webserver to handle the request and give the response).
In this particular case the latency and connection part of the RT are approximately equal. That is because the handler is very-very simple, almost no work at webserver part – and connection here opens for every request and closes for every response, because HTTP/1.0 is used on load generator and it has Connection: Close setting by default. Switching to keep-alive definitely wiil help to improve the performance, especially in case with TLS requests (now it's simple HTTP, not HTTPS).
The network part is relatively small because the load generator and SUT are located on the same host and because the size of request/response is small.

The load test of course has impact on web application response time. Every performance test is designed with the consideration that load generator and system under test make the closed system where both depend on each other. For example, if the load generator is limited with instance (not enough VU) it wiil have no resources to create the scheduled load, because the sent requests will be waiting for responses and the instances will be occupied in waiting state.
In my case, the load generator and the system under test are located on the same host, so they will fight for resources (CPU time, for example, or memory).

The optimal application response time for modern web applications depends on the web application: online banking systems with heavy transactions can be long, but the user will wait for the result because it's important for him, and security and reliability are more important in

this case than the speed. And some microservice responsible for authentication/authorization must respond in milliseconds, because many other services depend on it and include its response time for their response times.
Traditionally we consider three numbers: 0.1s, 1s and 10 seconds as some etalon response time. 0.1 second is considered by user as instant reaction of system, usually the user doesn't notice it. One second is enough long to notice some interruption, but it is usually considered as acceptable. Any second longer is already a nuisance for the user, so we should really think and fight for the every second. After 10 second the user's attention is lost and it should be a really important web application for him to stay with it.

There are many ways to define acceptable load for the application depending on requirements for the service. But the main metrics that I consider as important are:

errors rate (the app should respond correctly to the 99% of requests, the percentile depends on reqs, of course);

response times (see the previous answer, say it should be 95% in 3 seconds);

CPU and other important host resources usage (there should be some reserve for peal load and one should be able to maintain the application in case of any troubles, say it should be 75% of CPU).