Webserver benchmarking with Lancet : end-to-end application latency for HTTP-based applications

Table of Contents

1 Introduction

The aim of this project is to measure the end-to-end application throughput and latency for HTTP-based applications using open loop experiments in order to obtain more realistic results that reflect the targeted deployment environment.

2 HTTP protocol

HTTP protocol, created by Tim Berners-Lee at CERN in the early 90's, is a client-server communication protocol to exchange HTML code whose acronym stands for "Hypertext Transfer Protocol". Since then, HTTP has known many updates and various versions of the protocol are used nowadays. A HTTP server uses by default port 80. HTTP protocol works in a request-response scheme, where there are many types of request to specify what task has to be performed, such as GET to simply retrieve the content of a webpage, HEAD to get the meta data of the webpage only, POST which is mainly used to interact with the website and more such as OPTIONS, CONNECT, TRACE, PUT, PATCH and DELETE, some of these operations requiring a particular authorization.

The figure below (Figure 1) illustrates a GET request example to access the website called www.example.com formatted according to HTTP protocol.

```
GET / HTTP/1.1
Host: www.example.com
```

Figure 1 – GET request example with HTTP version 1.1

A HTTP request or response contains "headers" which are parameters to specify the purpose of the request and how it has to be performed. Each header is written on a new line, and is formatted as following: First, it has a parameter name (for example "Host", "Content-Length", "Connection", …) followed by a colon with the parameter value. The headers order is not relevant, even though it is good practice to regroup them logically by showing first the general headers and then the request headers. More specifically, a HTTP response contains a status code which is a number whose purpose is to inform the receiver on how the request was interpreted. Concretely, this number summarizes if the request has been successful (200 OK), or if the target webpage was not found (404 Not Found), if the request was valid but the server refused it (403 Forbidden) or, as a last example, if the request was simply not formatted correctly (400 Bad Request). Below is an example of how headers are formatted from a response to a successful GET response (Figure 2).

```
HTTP/1.1 200 OK
Server: nginx
Date: Mon, 21 Jan 2019 13:09:32 GMT
Content-Type: text/html; charset=UTF-8
Content-Length: 89222
Connection: keep-alive
Vary: Accept-Encoding
Accept-Ranges: bytes
Vary: Accept-Encoding
Cache-Control: max-age=0, public
Expires: Mon, 21 Jan 2019 13:09:32 GMT
X-Powered-By: WP Rocket/2.11.6
X-Powered-By: PleskLin
```

Figure 2 – GET response example

3 About Lancet

3.1 Lancet structure

Lancet is a distributed tool which is modular. There are different types of entities, each having a special role.

First there is a coordinator (C) whose task is to manage agents, which send requests to the target server in order to measure the latency. More precisely, there are three types of agents: throughput agents (TA) whose only purpose is to load the target server, latency agents (LA) which are lightly loaded, and their aim is to measure the latency and finally, symmetric agents (SA), which can do both simultaneously.

At the beginning of the deployment, parameters are given to the coordinator to specify the protocol to use as well as other specifications such as the number of requests sent per second per agent or the number of threads.

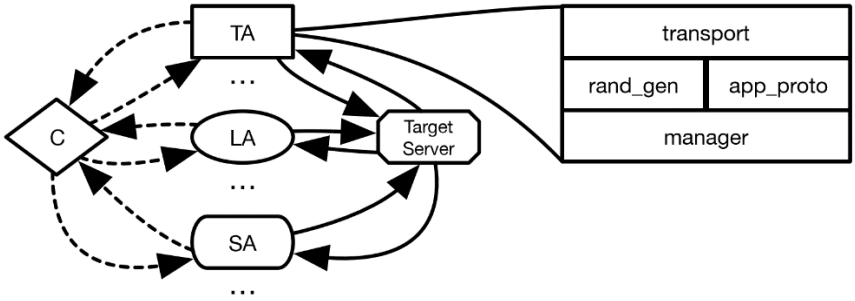

Figure 3 - Lancet structure

Each agent has the inner structure shown in figure 3 in the table which contains, among others, a transport layer and an application layer, called “app_proto”. When the protocol is specified to the coordinator once Lancet is launched, the “app_proto” entity is initialized according to the chosen protocol passed to the coordinator. This protocol could be for example echo, synthetic or, in this case, HTTP.

3.2 Interaction between the application layer and the transport layer

Adding a new protocol to Lancet can be done by only coding at the application layer without having to modify the transport layer since they are decoupled. The remaining questions is thus: *How do the application layer and the transport layer interact?*

The steps of an agent routine answer to this and are the following:

First, the transport layer establishes the TCP connection with the target server. Then, the application layer transfers the request – in this case, a HTTP one – to the transport layer which is responsible for sending it to the server and receiving the answer. Then, the transport layer gives the response back to the application layer whose final task is to process it (Figure 4).

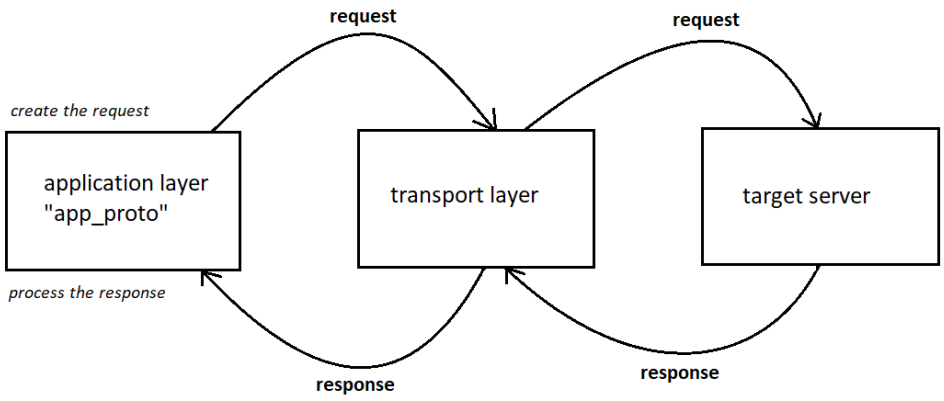

Figure 4 - Interaction between application layer and transport layer

Hence my task consisted of adding HTTP protocol to the existing agent infrastructure in the code section dedicated to the application part “app_proto”.

3.3 Application layer structure

Figure 5 shows the `application_protocol struct` I have been working with. It needs a protocol type, in this case HTTP. The `application_protocol` entity also has two functions, `create_request` and `consume_response`, specific to each protocol. There is also a generic pointer in the `app_proto struct, arg,` to which some additional information can be passed.

```
struct application_protocol {
    enum app_proto_type type;
    void *arg;
    int (*create_request)(struct application_protocol *proto,
                          struct request *req);
    struct byte_req_pair (*consume_response)(struct application_protocol *proto,
            struct iovec *response);
};
```

Figure 5 - application_proto struct

All the available protocols have these two functions as well as a last one to initialize the `struct` correctly called *"protocol_name*`_init`*"*.

4 What I have implemented

I have implemented the 3 following methods that characterize HTTP protocol:

- **`int http_create_request`**`(struct` **`application_protocol`** `*proto, struct` **`request`** `*req)`
- `struct` **`byte_req_pair http_consume_response`**`(struct` **`application_protocol`** `*proto,struct` **`iovec`** `*response)`
- `static` **`int http_init`**`(`**`char`** `*proto, struct` **`application_protocol`** `*app_proto)`

My code allows the user to generate GET and POST requests. There is also the possibility randomly alternate between GET and POST requests according to a chosen ratio. The function **`http_consume_response`** is looking for three points: The status code, the HTTP version and the Content-Length header.

The pointer `void *arg` of the `application_protocol struct` has been used to provide some useful precisions about the requests that needed to be created and to this extent, a new `http_req_elems struct` had to be implemented. This new type of structure contains all the fields of the parametrizable request, such as the HTTP version we want to use, the host of our target, the URI and eventually a body according to the type of request we want to send along with its length in bytes and this is this new `struct` that has been passed to the `arg` pointer.

5 Results

Lancet has been deployed to test the performances of NGINX, a HTTP server, serving a small html page. The measurements went the following:

NGINX thread pools was enabled to scale throughput and allow multithreading.

On Lancet side, the asymmetric mode was used, meaning that the experiment was led with 3 throughput agents and 1 latency agent. Every agent had 8 threads and 64 connections.

The latency graph (Figure 6) shows how the latency evolves according to certain throughputs, giving a performance appreciation of the server.

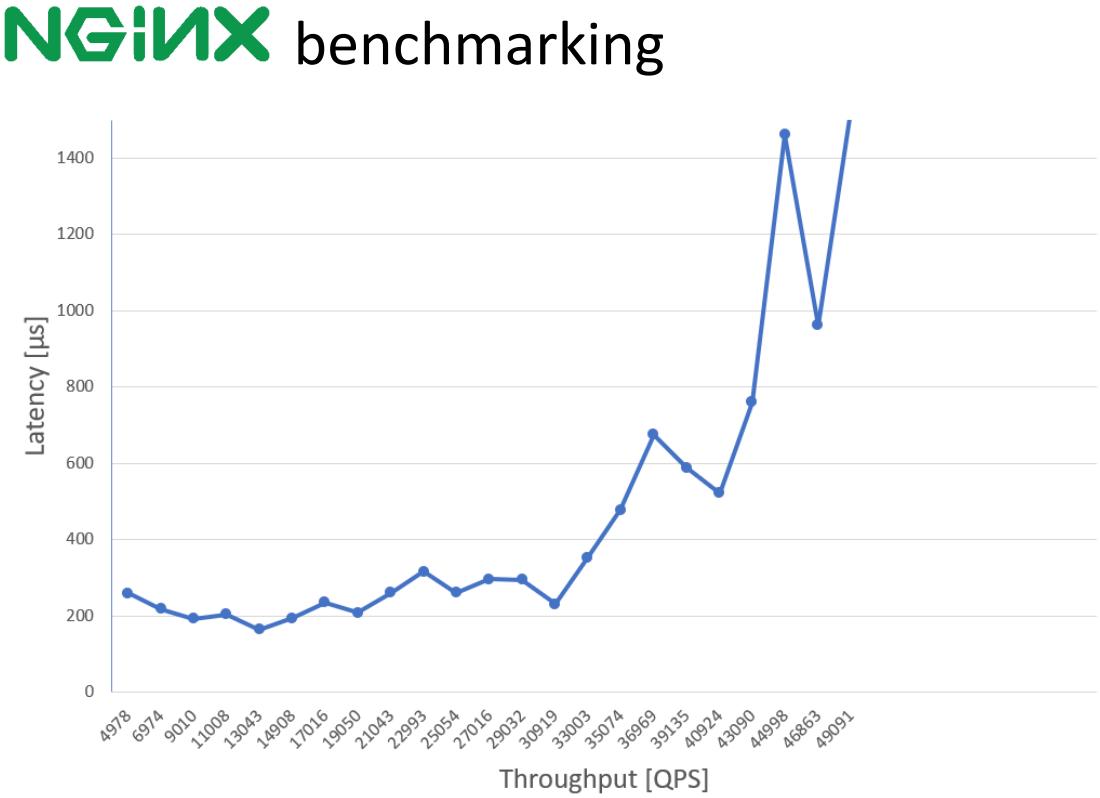

Figure 6 - NGINX server latency vs queries per second (QPS)

6 What I have learnt

Not only did this project teach me to code faster and more efficiently in C, but it has also brought me other knowledge and new tools to use. First, I learnt how we differentiate the ways a webserver is benchmarked: open vs closed loops experiments.
Moreover, even though I have always enjoyed learning more about computer networks, I never had the opportunity to study HTTP protocol in such details such as how to format a HTTP request.

Additionally, I did know Wireshark, but I never really used it. Thanks to this project, I have been able to do so to supervise HTTP packets sent to test my code. For the same purpose, I have also been using tcp_dump. Another tool I discovered is Advanced REST client to format my HTTP requests correctly.
I also had the opportunity to discover some new features in Git, such as the rebasing process and how to fork.
Furthermore, I could get a first insight of the Go programming language even though I did not concretely write a program with it.
On top of that, this project was also enriching since I could get more familiar with Linux command line.
And of course, adding my code to an already existing and complex project was also a challenge since it required me to understand the structure of Lancet and produce some code that would fit in.