Final project - INF01015
Network management as micro services

Lucas S. Hagen[1], **Lucas A. Tansini**[2]

[1]Instituto de Informática – Universidade Federal do Rio Grande do Sul (UFRGS)
CEP – 91.501-970 – Porto Alegre – RS – Brazil

{lucas.tansini,lucas.hagen}@inf.ufrgs.br

Abstract. *This paper presents the implementation of a reverse proxy-based network management microservice established with Docker containers. This work carries out three microservices: Wordpress running with Apache Web Server, MYSQL Database and Varnish Cache. Our results show that a reverse proxy managed network can improve the overall user experience and server load balance with a diverse set of resource caching.*

1. Introduction

Utilizing a microservice design can bring very flexible solutions to the network, the pieces become smaller and self-contained. The network gets bigger, busier, and hence, indispensable. As we utilize microservices for network management, our deploy time, monitoring time and setup time shrink. To implement these kinds of microservices, the platform as a service solution named as Docker is very well used. Docker can summarize the setup of each application as a microservice (further explained on future sections).

The microservice implementation on this paper is an reverse proxy-based network monitoring/logging, utilizing Varnish Cache. Varnish is an HTTP accelerator designed for content-heavy dynamic web sites as well as APIs, and it can reduce drastically the server load balance, thus reducing the amount of servers needed to host applications and drastically increase the number of simultaneous requests to the server itself.

2. Architecture and Network Topology

As seen in Figure 1. common network topologies implement a direct and simple client-server architecture, leading to direct access to the server itself. Our network topology implements Varnish as an external shell to communicate with potential clients, depicted in Figure 2.

As shown in the figure, Varnish Cache funnels all the access through its service, meaning that any user request must pass through Varnish. Varnish work is relatively simple, the administrator establishes a set of rules to determine what sections and pages need caching and those who don't need as well - those configurations are represented on a configuration file (further explained).

Varnish rules can be quite embracing, and some of them are implemented on this proof of concept. For example, we can determine that Varnish doesn't cache any login-related page, so we write the piece of code inside a Varnish Subroutine, that establishes a well-defined behavior, making the access through those pages go directly to the back-end, and also ignore already set cookies on Wordpress, as depicted in Figure 3.

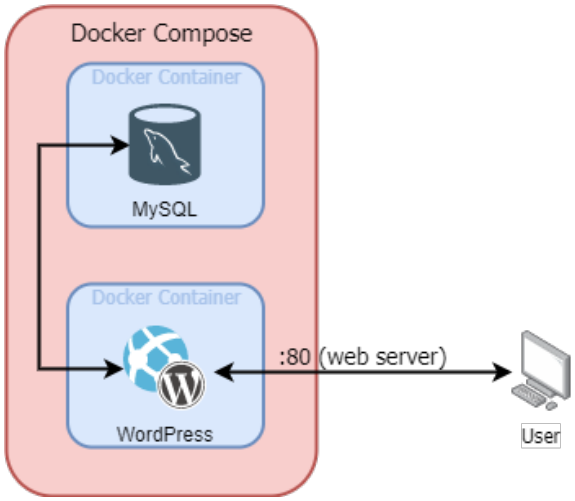

Figure 1. Network topology with without Varnish Cache solution

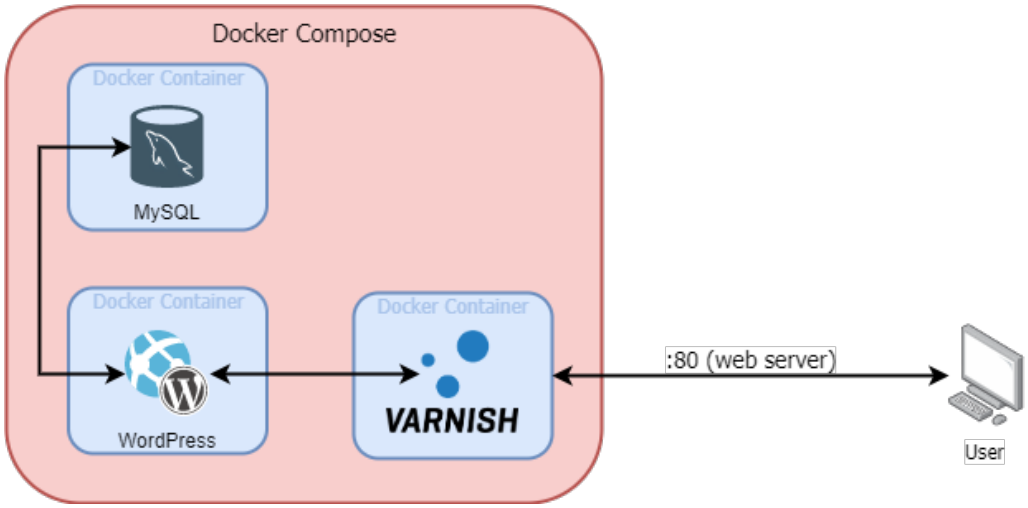

Figure 2. Network topology with Varnish cache solution

3. Environment configuration and Setup tutorial

As imposed above, the overall configuration can be quite embracing. This section will provide some guidelines on how to run the Docker compose file and give an overview of the components.

Starting with the root folder, containing the docker-compose.yml file, this is where the environment is built. As depicted in Figure 4, the main compose file holds up three containers, as previously shown. The first container is the database container, and this file describes some environment variables and configurations. Next, the Apache web-server Wordpress is defined, also with some vital configurations. Finishing the configuration file, we have the Varnish container configuration, and the overall configuration is located on `/config/varnish` folder, inside `default.vcl` file, shown on Figure 3.

The configuration listed above is a well-structured network topology meant to be easy to scale, configure, and deploy. It's important to note that the command used to run the environment is: `docker-compose up -d`, with the terminal inside the

```
sub vcl_recv{
  if(req.url ~ "wp-(login|admin)")
  {
    return(pass);
  }

    set req.http.cookie = regsuball(
                            req.http.cookie,
                            "wp-settings-\d+=[^;]+(; )?",
                            "");
    set req.http.cookie = regsuball(
                            req.http.cookie,
                            "wp-settings-time-\d+=[^;]+(; )?",
                            "");
    set req.http.cookie = regsuball(
                            req.http.cookie,
                            "wordpress_test_cookie=[^;]+(; )?",
                            "");
    if (req.http.cookie == "")
    {
        unset req.http.cookie;
    }
}
```

Figure 3. Varnish default.vcl archive configuration

root folder. After the environment is set, the client can access the webserver on the `localhost:80` URL on any browser.

4. Network Monitoring and Results

Varnish Cache offers a powerful logging tool, known as Varnishlog. Varnishlog can be accessed inside Varnish container from the command `varnishlog -i VCL_call,VCL_return,ReqURL`. The command uses as arguments the tags that the user wants to monitor, and this example only monitors the VCL calls, returns, and the requested URL. VCL calls can be interpreted as the action that the system is requesting to Varnish cache, and the VCL return is the state after Varnish executes that action. As seen in Figure 7, the output is really easy to understand, and Varnish offers lots of different flags to output thousands of options.

Our microservice topology implements real-time monitoring to analyze network traffic and Varnish outputs, such as Cache miss, hit and even the amount of back-end bypasses made by Varnish.

For the following result analysis, a scenario with three simulated clients accessing the webserver simultaneously was performed, each one performing different requests - from login to content-only pages. As depicted in Figure 5, this implementation can have a great number of Hits on Varnish Cache, meaning that the server will be less busy, as the

```
version: '3.3'
services:
    db:
        image: mysql:5.7
        volumes:
            - db_data:/var/lib/mysql
        restart: always
        environment:
            MYSQL_ROOT_PASSWORD: somewordpress
            MYSQL_DATABASE: wordpress
            MYSQL_USER: wordpress
            MYSQL_PASSWORD: wordpress
    wordpress:
        depends_on:
            - db
        image: wordpress:latest
        restart: always
        environment:
            WORDPRESS_DB_HOST: db:3306
            WORDPRESS_DB_USER: wordpress
            WORDPRESS_DB_PASSWORD: wordpress
            WORDPRESS_DB_NAME: wordpress
    varnish:
        depends_on:
            - wordpress
        image: varnish:latest
        ports:
            - "80:80"
        restart: always
        volumes:
            - ./data/varnish:/var/lib/varnish
            - ./config/varnish:/etc/varnish
volumes:
    db_data: {}
```

Figure 4. Varnish default.vcl archive receive subroutine configuration

important requests, shown as 59.4% can pass through the cache and access directly the back-end.

Furthermore, as seen in picture 6, Varnish does a great job when caching images, and non-critical content (e.g sensitive data relying on database access and return to front-end client). Results show that 36.2% can be cached, reducing server overloading.

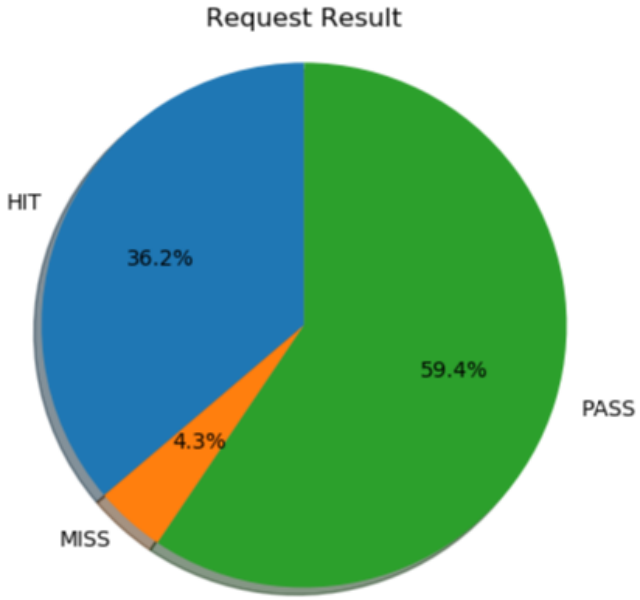

Figure 5. Percentage of Varnish return states.

5. Conclusion

After a thorough analysis of results, this work can conclude that developing a reverse proxy-based network management microservice established with Docker containers can be really helpful for building heavy request-bound applications. Microservices can be really helpful when deploying applications, thus reducing work time and focusing on essential tasks, such as network monitoring and improvement with all results presented on the monitoring task.

6. References

GitHub Repository - https://github.com/lucastansini/tfGerenciaRedes

Varnish - https://varnish-cache.org/docs/trunk/reference/vcl.html

Docker - https://www.docker.com/

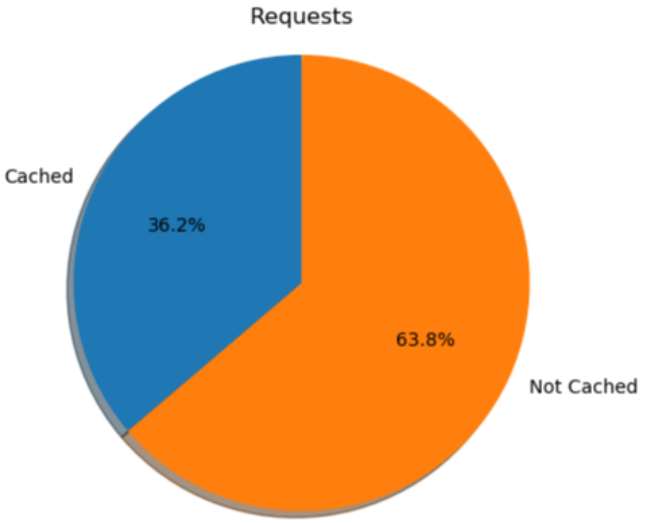

Figure 6. Percentage of Varnish cache miss and hit.

```
*    << Request  >> 65588
-    ReqURL         /
-    VCL_call       RECV
-    VCL_return     pass
-    VCL_call       HASH
-    VCL_return     lookup
-    VCL_call       PASS
-    VCL_return     fetch
-    VCL_call       DELIVER
-    VCL_return     deliver

*    << BeReq    >> 65591
-    VCL_call       BACKEND_FETCH
-    VCL_return     fetch
-    VCL_call       BACKEND_RESPONSE
-    VCL_return     deliver
```

Figure 7. Varnishlog ouput.