Infraestruturas de Tecnologias de Informação

TP1- Grupo 2

Trabalho 3 - Multiprocessamento

Francisca Gonçalves de Freitas Pereira Coelho
a95481
a95481@alunos.uminho.pt

Raúl José de Castro Pereira
pg54171
pg54171@alunos.uminho.pt

Diogo Froufe e Castro dos Santos Marques
pg53780
pg54780@alunos.uminho.pt

Paulo Jordão Xavier
pg54131
pg54131@alunos.uminho.pt

Miguel Gonzaga Morais de magalhães
pg54099
pg54099@alunos.uminho.pt

1. Balanceador de Carga (*Load Balancer*)

1.1 Estudo e Escolha

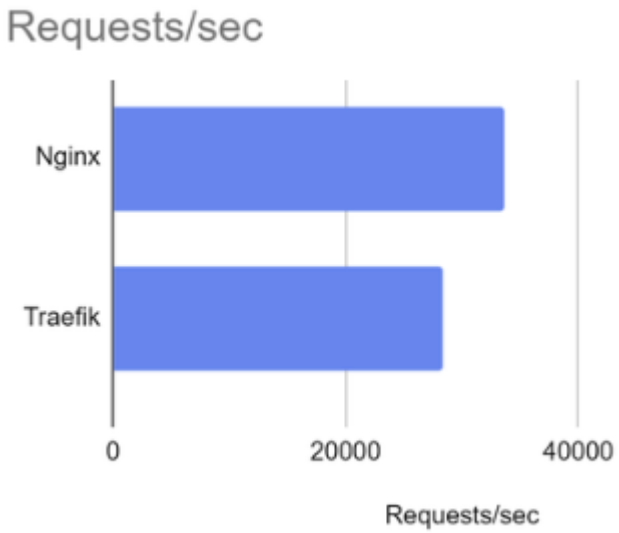
Um *load balancer* é um software ou dispositivo que distribui o tráfego de rede entre servidores diferentes, de modo a maximizar a velocidade e a utilização da capacidade e garantir que nenhum servidor fique sobrecarregado, podendo prejudicar o desempenho. Quando um servidor fica inativo o *load balancer* trata de redirecionar o tráfego de dados para os restantes servidores online. Quando um novo servidor é adicionado, o load balancer começa automaticamente a enviar solicitações para o mesmo. Existem vários tipos de *load balancer* tais como:

Tipos de *Load Balancer*	Definição	Exemplos
Load Balancer* de *software	O software pode ser instalado e configurado para distribuir o tráfego de rede entre servidores.	Nginx, Traefik, HAProxy
Load Balancer* de *hardware	É um dispositivo físico que é colocado à frente dos servidores para distribuir o tráfego, otimizando o desempenho e garantindo disponibilidade.	F5 BIG-IP, Citrix ADC
***Load Balancer* de aplicação**	Trabalha na camada de aplicação sobre o modelo OSI. Recebe solicitações e avalia as regras com base em informações específicas do protocolo de aplicação, como HTTP. Estes não distribuem apenas o tráfego, mas também oferecem funcionalidades como otimização de aplicações e SSL.	AWS Application Load Balancer, Azure Application Gateway
***Load Balancer* de nuvem**	São serviços ou dispositivos oferecidos para distribuir o tráfego entre instâncias de servidores virtuais na nuvem. Estes são projetados para serem escaláveis e flexíveis, ajustando dinamicamente a distribuição de carga conforme o necessário para garantir a disponibilidade e o desempenho das aplicações.	Amazon Web Services (AWS), Microsoft Azure, Google Cloud Platform (GCP)
***Load Balancer* de container**	São dispositivos ou softwares projetados para distribuir o tráfego entre várias instâncias de containers. A distribuição eficiente das cargas de trabalho em containers é garantida por esses balanceadores, o que aumenta a escalabilidade e confiabilidade das aplicações baseadas em containers.	Traefik, Nginx

Load Balancer	Nginx	Traefik
Definição	É um software open source popular para servidores web. Foi desenvolvido para resolver o problema C10k, que é definido como o problema de gerenciar dez mil conexões ao mesmo tempo. Inicialmente desenvolvido para resolver problemas de desempenho e escalabilidade associados ao tratamento de tráfego pesado em servidores web Apache tradicionais. Mas o Nginx é conhecido pelo seu alto desempenho, baixa utilização de recursos e capacidade de lidar com milhares de conexões simultâneas com carga mínima.	É um reverse proxy moderno open source e um load balancer. Projetado para lidar com o tráfego de microsserviços e aplicativos em contêineres . Desenvolvido para funcionar perfeitamente com plataformas de orquestração de contêineres , como Docker e Kubernetes. Além disso, a ferramenta descobre e roteia dinamicamente o tráfego para contêineres à medida que eles são criados ou destruídos.
Vantagens	- Grande e ativa comunidade de utilizadores e colaboradores que garante excelente serviço e atualizações regulares de software. - Capaz de lidar com mais de 10 mil conexões simultâneas com baixo consumo de memória. - Melhor no tratamento de conteúdo estático . - Recomendado para sites correndo em VPS . - O proxy da web e o proxy da base de dados são excelentes. - Suporta protocolos IMAP , POP e SMTP para proxy reverso . - Os recursos mais valiosos são o gateway e a capacidade de publicar em sites. - Utiliza recursos mínimos do sistema, o que ajuda a reduzir custos de hardware e melhorar o desempenho do servidor.	- Funciona em diversas plataformas, incluindo Docker e Kubernetes. - Projetado com o desempenho em mente, oferecendo roteamento de tráfego rápido e balanceamento de carga . Além disso, resultando em respostas mais rápidas e melhor desempenho do aplicativo. - Oferece suporte HTTPS integrado e gera automaticamente certificados SSL/TLS. - Não precisa adicionar manualmente novos serviços e rotear o tráfego quando novos contêineres são iniciados ou interrompidos. - Oferece um arquivo de configuração YAML simples para configuração rápida de serviços e roteamento de tráfego. - Projetado para oferecer suporte a microsserviços e ambientes em contêineres. Ou seja, é facilmente escalável com um número crescente de contêineres.
Desvantagens	- É necessária mais automação para facilitar ainda mais a manutenção. - A escalabilidade poderia ser melhorada.	- Comparado a outras ferramentas de gerenciamento de tráfego de rede , como o NGINX, o Traefik oferece menos recursos. - Voltado principalmente para

	- Lista menos extensa de módulos. - Embora o Nginx seja adequado para servir conteúdo estático, não é tão eficiente para servir conteúdo dinâmico, especialmente em ambientes de alto tráfego.	microsserviços e ambientes de contêiner. O que implica que, se os utilizadores não tiverem conhecimento destas tecnologias, poderão não utilizar o Traefik de forma eficaz. - Não oferece mecanismos de autenticação integrados, os utilizadores devem configurar a autenticação em nível de aplicativo ou usar ferramentas externas. - Em algumas situações, como durante uma atualização do Traefik, a configuração pode ser perdida, levando à perda de dados ou ao aumento do tempo de inatividade. - Projetado especificamente para funcionar com microsserviços e não oferece suporte para alguns recursos avançados, como suporte WebSocket

Com base nesta pesquisa, ambos os servidores web têm os seus pontos fortes, mas no fim o load balancer que escolhemos utilizar foi o Nginx uma vez que este é ligeiramente mais rápido que o Traefik. O Nginx está muito mais estabelecido e é conhecido pelo seu alto desempenho, estabilidade, rico conjunto de recursos e simplicidade na instalação e configuração, com o benefício adicional do baixo consumo de recursos.



Nginx vs Traefik proxing performance (Higher is better)

1.2 Implementação

Começámos por instalar o Nginx no docker.

```
PS C:\Users\raulj> docker pull nginx
Using default tag: latest
latest: Pulling from library/nginx
1f7ce2fa46ab: Pull complete
9b16c94bb686: Pull complete
9a59d19f9c5b: Pull complete
9ea27b074f71: Pull complete
c6edf33e2524: Pull complete
84b1ff10387b: Pull complete
517357831967: Pull complete
Digest: sha256:10d1f5b58f74683ad34eb29287e07dab1e90f10af243f151bb50aa5dbb4d62ee
Status: Downloaded newer image for nginx:latest
docker.io/library/nginx:latest

What's Next?
  1. Sign in to your Docker account → docker login
  2. View a summary of image vulnerabilities and recommendations → docker scout quickview nginx
PS C:\Users\raulj> docker run --name mynginx1 -p 80:80 -d nginx
34ec8db00a85144946b017868f905385a0cecd8a345fabf237a160b7e06acad7
```

De seguida, criamos três servidores pdfer localmente nas portas 8090, 8091 e 8092.

pdfer-container da88ba2be93b	pdfer-image	Running	0%	8090:8080	2 hours ago
pdfer-container2 8c075d12d635	pdfer-image2	Running	0%	8091:8080	2 hours ago
pdfer-container3 98fd429063d1	pdfer-image3	Running	0%	8092:8080	2 hours ago
mynginx1 34ec8db00a85	nginx	Running	0%	80:80	25 minutes ago

Copiamos o ficheiro default.conf para uma diretoria onde o poderíamos editar.

```
PS C:\Users\raulj> docker cp mynginx1:/etc/nginx/conf.d/default.conf C:/Users/raulj/nginx/default.conf
Successfully copied 3.07kB to C:\Users\raulj\nginx\default.conf
```

Ao abrir este adicionamos um upstream chamado pdfer contendo os três servidores pdfer e uma localização apontando para esse mesmo upstream.

```
upstream pdfer {
   server localhost:8090;
   server localhost:8091;
   server localhost:8092;
}

server {
    listen       80;
    listen  [::]:80;
    server_name  localhost;

    #access_log  /var/log/nginx/host.access.log  main;

    location / {
        root   /usr/share/nginx/html;
        index  index.html index.htm;
    }

    location /files {
        proxy_pass http://pdfer/files;
    }
```

Após isto copiamos o ficheiro default.conf de volta para a sua localização inicial, verificando se este tinha sido bem configurado e por último reiniciamos o nginx.

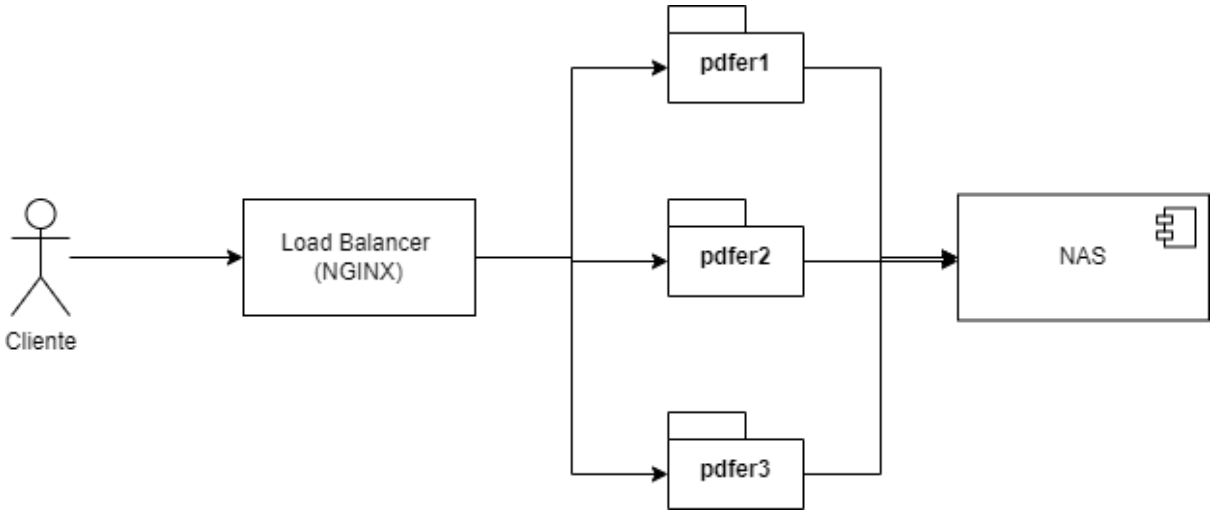
```
PS C:\Users\raulj> docker cp C:/Users/raulj/nginx/default.conf mynginx1:/etc/nginx/conf.d/default.conf
Successfully copied 3.07kB to mynginx1:/etc/nginx/conf.d/default.conf
PS C:\Users\raulj> docker exec mynginx1 nginx -t
nginx: the configuration file /etc/nginx/nginx.conf syntax is ok
nginx: configuration file /etc/nginx/nginx.conf test is successful
PS C:\Users\raulj> docker exec mynginx1 nginx -s reload
2023/11/25 15:04:36 [notice] 57#57: signal process started
```

De seguida, conseguimos verificar, tal como se vê na seguinte imagem, que estávamos a aceder corretamente ao *loadbalancer* implementado com o nginx para medir as métricas. Podendo passar assim para a fase seguinte, que se tratava em configurar um NAS para guardar os dados remotamente.

```
Server Software:        nginx/1.25.3
Server Hostname:        localhost
Server Port:            80
```

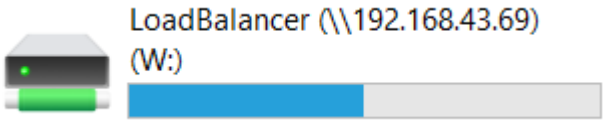
2. Implementação da arquitetura em rede

Considerando que cada elemento é uma máquina própria e todos estão conectados na mesma rede, a arquitetura utilizada foi a seguinte.



2.1 Configuração do SMB, para aceder à NAS

Esta fase foi feita semelhantemente ao que tinha sido feito no 2º trabalho, por isso não vamos entrar em detalhes. O que fizemos diferente foi que em vez de termos 3 discos para implementar um raid, usamos apenas 1 disco, o que implica então que não usamos uma arquitetura raid, de modo a simplificar a complexidade desta entrega, pois o foco para esta não era esse. Deste modo, acabamos por apenas configurar o protocolo SMB, e por mapear a respetiva unidade de rede:



Após isto, este disco foi montado e prosseguimos para a medição das métricas.

```
raulpereira@GigaRal:~$ cd /mnt
raulpereira@GigaRal:/mnt$ sudo mkdir w
[sudo] password for raulpereira:
raulpereira@GigaRal:/mnt$ sudo mount -t drvfs W: /mnt/w
raulpereira@GigaRal:/mnt$ ls
c  w  wsl  wslg  y  z
```

Após a implementação do NAS, configuramos um docker-compose para correr o pdfer em rede, e para guardar os dados no novo NAS que acabamos de criar.

```yaml
version: '3.7'
networks:
  bridge:

services:
  pdfer:
    image: pdfer-image
    container_name: pdfer
    ports:
      - "192.168.43.191:8080:8080"
    networks:
      - bridge
    volumes :
      - ./store:/mnt/w
```

Para a conexão em rede escolhemos utilizar um telemóvel a fazer hotspot por ser a única alternativa viável a uma switch física. Inicialmente, fizemos ping para verificar a comunicação entre os computadores.

```
Ping statistics for 192.168.43.191:
    Packets: Sent = 4, Received = 4, Lost = 0 (0% loss),
Approximate round trip times in milli-seconds:
    Minimum = 7ms, Maximum = 77ms, Average = 27ms
```

```
Ping statistics for 192.168.43.233:
    Packets: Sent = 4, Received = 4, Lost = 0 (0% loss),
Approximate round trip times in milli-seconds:
    Minimum = 73ms, Maximum = 275ms, Average = 125ms
```

```
Ping statistics for 192.168.43.212:
    Packets: Sent = 4, Received = 4, Lost = 0 (0% loss),
Approximate round trip times in milli-seconds:
    Minimum = 8ms, Maximum = 82ms, Average = 41ms
```

Alterámos o default.conf do nginx para estes mesmos ip's:

```
upstream pdfer {
   server 192.168.43.191:8080;
   server 192.168.43.233:8080;
   server 192.168.43.212:8080;
}

server {
    listen       80;
    listen  [::]:80;
    server_name  localhost;

    #access_log  /var/log/nginx/host.access.log  main;

    location / {
        root   /usr/share/nginx/html;
        index  index.html index.htm;
    }

    location /files {
        proxy_pass http://pdfer/files;
    }
```

Por fim, medimos as métricas, utilizando a arquitetura descrita anteriormente:

```
Concurrency Level:      1
Time taken for tests:   196.212 seconds
Complete requests:      1000
Failed requests:        67
   (Connect: 0, Receive: 0, Length: 67, Exceptions: 0)
Total transferred:      320957 bytes
HTML transferred:       161957 bytes
Requests per second:    5.10 [#/sec] (mean)
Time per request:       196.212 [ms] (mean)
Time per request:       196.212 [ms] (mean, across all concurrent requests)
Transfer rate:          1.60 [Kbytes/sec] received
```

3. Reavaliação de métricas

Descrição do ambiente de testes:

	Computador (Trabalho 1 e 2)	Computador (Trabalho 3 - localhost)
Marca	Microsoft	Gigabyte
Modelo	Surface Book 3	G5 KF
Versão SO	Windows 11	Windows 11
Processador	Intel(R) Core(TM) i7-1065G7 CPU @ 1.30GHz 1.50 GHz	12th Gen Intel(R) Core(TM) i5-12500H CPU @ 3.30GHz 4.50GHz
Memória RAM	16.0 GB	16.0 GB

Tal como no trabalho 1 e 2, medimos outra vez as métricas:

	Trabalho 1	Trabalho 2	Trabalho 3 - hotspot	Trabalho 3 - localhost	Trabalho 1	Trabalho 2	Trabalho 3 -hotspot	Trabalho 3 - localhost
	A - 1000				B - 5000			
Requisições por segundo	203.78	188.14	5.10	971.62	199.45	191.31	-	863.24
Tempo por solicitação	4.907	5.315	196.212	1.029	5.014	5.227	-	1.158
Solicitações Falhadas	993	856	67	0	4089	418	-	0

	Trabalho 1	Trabalho 2	Trabalho 3 -hotspot	Trabalho 3 - localhost	Trabalho 1	Trabalho 2	Trabalho 3 -hotspot	Trabalho 3 - localhost
	C - 10000				D - 20000			
Requisições por segundo	194.17	185.83	-	583.83	196.94	185.22	-	539.36
Tempo por solicitação	5.150	5.381	-	1.712	5.078	5.399	-	1.854
Solicitações Falhadas	8037	956	-	0	16451	1831	-	0

	Trabalho 1	Trabalho 2	Trabalho 3 -hotspot	Trabalho 3 - localhost
	E - 25000			
Requisições por segundo	197.94	184.76	-	551.66
Tempo por solicitação	5.052	5.412	-	1.812
Solicitações Falhadas	21169	1880	-	0

Devido à impossibilidade de obter um switch, para fazer a ligação entre os diferentes computadores recorremos ao uso de um hotspot de um telemóvel. Esta solução, embora tenha permitido a continuação dos testes que foram realizados, apresentou alguns desafios inesperados que interferem na medição das métricas. Infelizmente, devido às limitações do hotspot do telemóvel, só foi possível realizar a medição das métricas com 1000 pedidos.

Esta restrição inclui a capacidade limitada de conexão do hotspot, resultando numa menor largura de banda disponível para a execução dos pedidos. Além disso, a estabilidade da conexão foi comprometida, o que levou a interrupções que prejudicaram a recolha dos dados.

Para conseguirmos comparar as métricas dos 3 trabalhos, decidimos medir as métricas em localhost, uma vez que nos trabalhos 1 e 2 também o fizemos.

3.1 Avaliação das Métricas

3.1.1 Avaliação das Métricas usando a coluna “Trabalho 3 - hotspot”

Requisições por segundo e Tempo por solicitação:

Através da tabela acima, conseguimos observar que para 1000 pedidos, as requisições por segundo e o tempo por solicitação têm valores muito elevados neste último trabalho em comparação com os anteriores. Isto está relacionado com o facto de anteriormente estar tudo em apenas um computador enquanto que neste, estavam interligados 4 computadores diferentes. Também o facto de termos utilizado o hotspot de um telemóvel contribuiu para a estes resultados menos positivos.

Solicitações falhadas:

Porém, relativamente às solicitações falhadas, houve uma grande diminuição deste número relativamente aos trabalhos 1 e 2, ou seja, a utilização de um *loadbalancer* contribuiu para uma diminuição significativa de solicitações falhadas. Além disso, o uso de um hotspot de um telemóvel, embora tenha afetado outras métricas, não parece ter sido um fator significativo para as falhas.

3.1.2 Avaliação das Métricas usando a coluna “Trabalho 3 - localhost”

Requisições por segundo:

Do trabalho 1 para o trabalho 2 as requisições por segundo diminuíram ligeiramente devido a um trade off entre o tempo de resposta e a tolerância a falhas. Deste modo, o nosso objetivo para esta fase era contrabalançar os tempos de resposta mais lentos, enquanto mantendo a tolerância a falhas. Com isto, através do uso de um load balancer implementado com o Nginx conseguimos melhorar substancialmente as requisições por segundo, mais notavelmente em um número menor de pedidos, mas também significativamente em um grande número de pedidos.

Tempo por solicitação:

Quanto ao tempo por solicitação, tal como na métrica anterior, queríamos ganhar a velocidade de resposta que foi perdida do trabalho 1 para o trabalho 2, devido a ter uma tolerância a falhas superior. Para isto, através do mesmo load balancer implementado com o Nginx, obtivemos tempos por solicitação consideravelmente mais rápidos. Tal como visto na primeira métrica avaliada, um número menor de pedidos leva a tempos de resposta ligeiramente melhores do que um maior número de pedidos.

Solicitações falhadas:

No caso das solicitações falhadas, estas tinham diminuído efusivamente do trabalho 1 para o 2, e agora tínhamos como objetivo manter esta capacidade de tolerância a falhas. Neste caso acabamos até por ir mais além anulando por completo as solicitações falhadas, isto deve-se a alguns dos seguintes fatores: implementação de um load balancer por via do Nginx, implementação de uma arquitetura com um NAS incorporado, e o ambiente de testes desta última fase ter sido executado por um computador com uma taxa de processamento consideravelmente superior, e com o triplo dos cores.

Por fim, através do uso de um load balancer como o Nginx conseguimos distribuir a carga de trabalho uniformemente pelas três instâncias do pdfer, aumentando assim os tempos de resposta por forma a anular a velocidade perdida devido à tolerância a falhas que se ganhou no último trabalho.

4. Estratégia para garantir a escalabilidade e redundância

A principal vantagem de incluir aplicações num ambiente de containers é a facilidade de replicar e multiplicar instâncias. Ao criar um ambiente virtual com todas as dependências e condições para a execução do código, é garantido que esse ambiente (imagem) será executado em qualquer máquina. Esta abordagem também permite que várias instâncias sejam criadas conforme a necessidade. Além disso, a utilização de uma infraestrutura de nuvem, como a Amazon Web Services (AWS) ou a Google Cloud Platform (GCP), permitiria a utilização de funcionalidades de escalabilidade automática, como o escalonamento automático, mas para isto implicaria que desde o início do desenvolvimento do nosso trabalho, tivéssemos em conta o automatismo como uma prioridade do sistema.

Outra estratégia importante seria a utilização de um sistema de gestão de contentores, como o Kubernetes, para garantir que os contentores são distribuídos uniformemente e para automatizar a escalabilidade e a redundância.

Além disso, é importante garantir que os dados sejam replicados e armazenados em várias regiões geográficas para assegurar a disponibilidade e a tolerância a falhas.