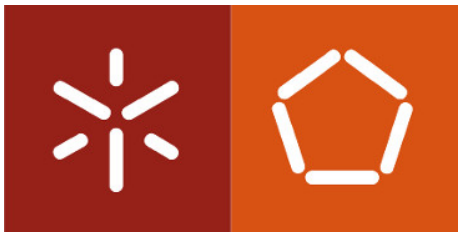
System Deployment and Benchmarking

Grupo nr.	5
a85370	Hugo Matias
a84802	João Nuno Abreu
a84577	José Pedro Silva
a84783	Pedro Rodrigues
a84485	Tiago Magalhães



Mestrado Integrado em Engenharia Informática
Universidade do Minho

Contents

1 Introdução

Cada vez mais o ser humano encontra-se conectado e dependente da tecnologia, sendo que esta tem de ser capaz de se adaptar e responder com sucesso às necessidades exigentes de qualquer tipo de cliente.

Nesse sentido, este trabalho tem como objectivo não só consolidar os conhecimentos obtidos na unidade curricular System Deployment and Benchmarking, nomeadamente no planeamento, configuração, análise de desempenho e operação de infraestruturas de elevada disponibilidade e desempenho, como também implementar um serviço escalável e de elevada disponibilidade de infraestruturas computacionais para a plataforma *GitLab*.

Ao longo dos próximos capítulos será apresentada a abordagem feita pelo grupo para cumprir com os requisitos acima descritos.

2 Gitlab

2.1 Descrição

O GitLab é um gerenciador de repositório de *software* baseado em git. GitLab é similar ao GitHub, no entanto o GitLab permite que os desenvolvedores armazenem o código em seus próprios servidores, ao invés de servidores de terceiros.

2.2 Arquitetura e Componentes

A famosa aplicação *Gitlab* contém inúmeros componentes já referidos na apresentação intermédia, no entanto, nem todos estes componentes são necessários para o correto funcionamento da aplicação e nem todos tem a necessidade de serem isolados como micro-serviço. Posto isto, e tal como foi sugerido na proposta de arquitetura, optamos apenas por isolar como micro-serviço a **base de dados** (*postgresql*), o ***redis*** e o **load balancer** (*HAProxy*), dado que podem ser vistos como diferentes camadas e também para serem escaláveis, resilientes e possibilitar uma melhor performance. O uso de *containers* possibilita uma rápida instalação e migração, além de que se um serviço de um *container* falhar basta reinicializar o próprio *container*, invés de ter que reinicializar o serviço e todas as suas dependências,

Nesta arquitetura, facilmente reparamos que o principal ponto crítico do sistema será a aplicação em si, uma vez que será o principal alvo de pedidos (HTTP como *downloads*, ficheiros e *push/pull requests*), sendo esta obrigada a computar cada um desses pedidos. Assim sendo, houve a preocupação em replicar a componente que contém a aplicação, partindo daí a necessidade da existência de um componente *load balancer*, sendo este responsável por gerir as conexões para os vários nodos replicados da aplicação. Este processo de replicação não é complicado e não implica cuidados extras, uma vez que os dados persistentes estão inseridos na base de dados, isto faz com que não seja necessário sincronização entre as várias réplicas.

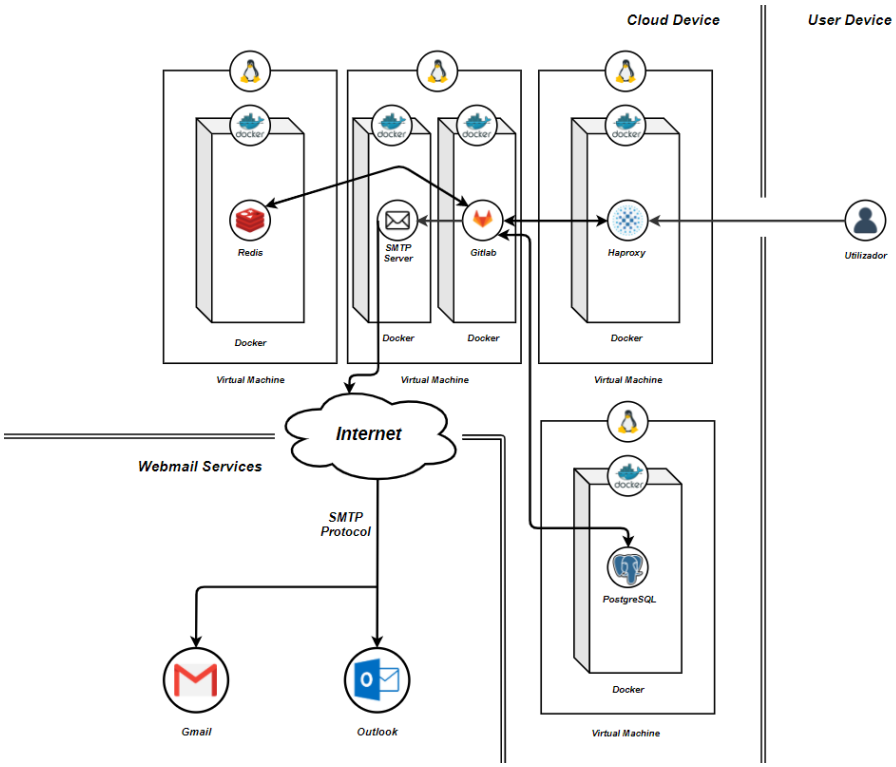


Figure 1: Arquitetura do sistema

3 Configuração e Deployment

De forma a automatizar o processo de *deployment* foi utilizada a ferramenta **Ansible**, introduzida na aula. Antes de mais, é pertinente reforçar a importância desta ferramenta e das suas componentes como *roles*, *variáveis*, *templates* e *inventários dinâmicos*.

Estas componentes permitem que várias *tasks* e/ou *handlers* sejam organizados e agrupados para determinada situação, podendo ser reutilizáveis através do uso de variáveis e templates.

No *deployment* do Gitlab, cada serviço (base de dados, *redis*, load-balancer e aplicação Gitlab) contém um role associado, permitindo que estes sejam facilmente configuráveis e estruturados.

3.1 Serviços

3.1.1 Database

O *role* responsável pela base de dados é composto pela diretoria *tasks* e *templates* e, tal como os nomes indicam, a primeira contém as *tasks* da base de dados e a segunda todos os templates necessários em formato *jinja*.

Como apresentado na Figura 1, todos os serviços correm dentro de *docker containers*. Dessa forma, a pasta *templates* contém os ficheiros *Dockerfile* e *docker-compose* necessários para iniciar um servidor *postgresql*.

Posto isto, as *tasks* para a base de dados são apenas criar a pasta `/postgresql` e copiar para a mesma os ficheiros necessários para correr o *docker*. Finalizado o processo de cópia de ficheiros, é iniciado o *docker container* produzido pelo ficheiro *docker-compose*.

Com o servidor *postgresql* a correr, são inseridos os dados necessários para que seja compatível com a aplicação Gitlab, ou seja, é criado um utilizador e base de dados. O nome e password do utilizador e nome da base de dados são variáveis definidas no *playbook* ao referenciar este role.

3.1.2 Redis

Tal como o servidor *postgresql*, o *redis* corre dentro de um *docker container* e, para isso, é também criada a pasta `/redis` e copiado para a mesma o ficheiro *docker-compose* com as configurações necessárias.

Após a cópia do ficheiro, é criado um *docker-container* a partir do mesmo, não havendo necessidade de configurações adicionais, o *redis* estará a escutar na porta definida no playbook

3.1.3 GitLab

A role *gitlab* corresponde ao serviço da aplicação em si, sendo que, tal como os serviços anteriores, corre dento de um *docker container*. Assim, da mesma forma que o *redis* e a base de dados, é criada a pasta `/gitlab` para onde

será copiado o ficheiro *docker-compose* responsável por criar o *container* com a aplicação.

Neste role é onde está mais presente a importância do uso de variáveis e template, juntamente com o inventário dinâmico, de forma a que a aplicação seja configurável e a receita *ansible* seja reutilizável.

3.1.4 HAProxy

Componente responsável pelo balanceamento da carga, dividindo a carga pelas inúmeros instâncias da aplicação, sendo a mesma tolerante a faltas, ou seja, caso uma falhe, ignora-a até que recupere outra vez.

Tal como todos os outros serviços corre dentro de um *docker container*, onde apenas é necessário especificar a configuração e as portas. Sendo assim, as portas são definidas no *playbook* como variáveis e a configuração está inserida como um *template* no formato *jinja* permitindo a inserção de todas as instâncias da aplicação.

No ficheiro de configuração do HAProxy foi necessário configurar uma *cookie*, de modo a garantir persistência de sessões, uma vez que sem esta os *tokens*, como por exemplo para definir a *password* do administrador não eram guardados para cada sessão do HAProxy.

3.2 Inventário dinâmico

A noção de inventário dinâmico é fundamental para a automatização do processo de *deploy* uma vez que, sem o mesmo, seria necessário criar manualmente as máquinas virtuais no ambiente *Google Cloud* e criar um inventário com as mesmas. No entanto, este processo é repetitivo e pode ser automatizado através dos inventários dinâmicos.

Para isso, é utilizado o *plugin* `gcp_compute` que permite autenticar-nos na conta da *Google Cloud* e automatizar todas as operações necessárias como criar instâncias e definir *firewall rules*. Sendo assim, é criado um role *gcp* que irá correr em ambiente local, por forma a adicionar *hosts* ao inventário.

No *playbook* são definidas as instâncias e portas a serem abertas e através dos módulos disponíveis pelo `gcp_compute` são realizadas essas operações de forma automática, sendo depois, estas instâncias, adicionadas ao *inventário* através do módulo `add_host`, com a tag definida no *playbook*.

Em suma, a inserção de inventários dinâmicos na nossa receita, permitiu-nos criar máquinas virtuais na *Google Cloud* e guardar/organizar a sua informação de forma a que possam ser usadas para as restantes roles.

3.3 Vagrant

Apesar da automatização do processo de criação de instâncias na *Google Cloud*, sentimos necessidade de criar uma máquina virtual em ambiente local para correr a receita *ansible* explicada acima. Para isso, utilizamos a ferramenta *Vagrant*, lecionada nas aulas.

O *Vagrant* é responsável por criar apenas uma modesta máquina virtual, copiar todos os ficheiros relativos às receitas *ansible* para essa mesma máquina e copia a pasta *ssh* que contém a chave pública e privada necessárias para aceder às instâncias da *Google Cloud*. Apesar de ser pouco recomendável a passagem de chaves privadas, esta chave é apenas usada para acesso à *Google Cloud* e, por isso, apenas deve ser passada a pessoas com permissões para tal.

3.4 Ansible Vault

O Ansible Vault permite cifrar informação sensível, tal como *password's* ou ficheiros, assim não é preciso manter esta informação em *plaintext*. Neste caso *password's* para a base dados foram cifrados com esta ferramenta. Torna-se fundamental que o ficheiro que guarda o segredo que a ferramenta utilize tenha os devidos mecanimos de controlo de acesso.

4 Monitorização

A monitorização torna-se fundamental neste contexto, a fim de se conhecer os principais *bottlenecks*, garantir o correto funcionamento e também para controlo de custos do serviço de *Cloud*.

De forma a tornar a monitorização possível, são precisas 3 ferramentas distintas: *metricbeat*, *elasticshearch* e *kibana*.

Todos os componentes da aplicação contém um serviço *metricbeat* responsável por enviar a monitorização desse mesmo componente ao *elasticsearch*, que analisa os dados e apresenta no *kibana*. Tal como todas as outras componentes, o *elasticsearch* e o *kibana* correm dentro de um *docker-container*.

A informação referente à monitorização do sistema estará presente no host do *elasticsearch*/*kibana* com a porta definida no *playbook* como variável.

Nesta página é aberto um painel onde pode ser selecionada a secção das métricas, apresentado resultados como na Figura 2. As métricas disponíveis neste painel são:

- CPU Usage
- Memory Usage
- Load
- Inbound Traffic
- Outbound Traffic

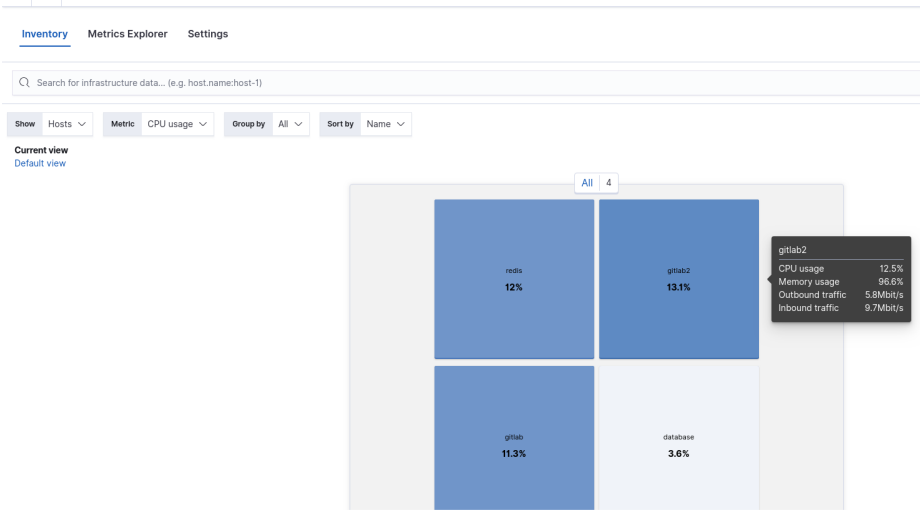


Figure 2: Exemplo de apresentação de monitorização no *kibana*

Como podemos verificar pela Figura 2, existe balanceamento de carga entre as duas instâncias do *Gitlab*.

5 Avaliação

De modo a avaliar o desempenho da nossa aplicação recorremos a testes de carga utilizando o ***JMeter***. Utilizando GUI do *JMeter* criámos testes que tentam simular um pouco do que será o funcionamento do *GitLab*.

Os testes foram corridos no ambiente de linha de comandos de modo a poupar recursos das nossas máquinas, sendo que as limitações de hardware e as limitações relativas à **JVM** foram os principais obstáculos ao aumento da carga.

Existiram dificuldades em conseguir usar o *JMeter*, devido ao *website* apresentar protecção contra falsificação de solicitação entre sites (CSRF - *Cross Site Request Forgery*). Para ultrapassar este problema foi necessário configurar o *JMeter*, começando-se por adicionar um *Regular expression extractor*, para obter o valor do *csrf token*, devido ao facto do seu valor ser dinâmico. O valor detectado pela expressão regular é atribuído a uma variável *JMeter* e depois passado no parâmetro de pedidos (*Request Parameter*).

De seguida, estarão presentes os nossos resultados dos testes de carga, comparando valores com e sem o serviço de load balancing para possuirmos um termos de comparação para a implementação da nossa aplicação. Em vez de fazermos os pedidos para o load balancer, fazemos diretamente para a camada aplicacional.

5.1 Teste 1

O Teste 1 visa representar o cenário mais básico: um utilizador aceder à página inicial do GitLab. Para este teste irão ser utilizadas mais *threads* do que nos restantes dado que num cenário real, o acesso à página inicial do GitLab é o mais comum. Ao correr este teste é testado o seguinte método:

1. GET /

O teste foi corrido com 100, 500 e 1000 *threads* (utilizadores), sendo que recorremos aos relatórios gerados pelo *JMeter* para apresentar os seguintes resultados.

Threads	Erro	Tempo Resposta Médio (ms)	*Throughput (Trans./s)*
100	0.00%	1870.70	58.90
500	25.25%	6103.43	81.69
1000	31.25%	10613.66	89.80

Table 1: Sumário do Teste 1 - Com Load Balancer

Threads	Erro	Tempo Resposta Médio (ms)	*Throughput (Trans./s)*
100	0.00%	4230.16	30.74
500	39.23%	7127.11	47.02
1000	46.28%	12118.93	47.76

Table 2: Sumário do Teste 1 - Sem Load Balancer

Analisando os dados da tabela, podemos verificar que com 100 não obtemos nenhuma percentagem de erro. Ao aumentar o número de threads, verificámos também o aumento do *throughput* da aplicação começou a estagnar. O tempo de resposta médio aos pedidos por sua vez não estagnou, subindo para valores de maneira proporcional.

5.2 Teste 2

O Teste 2 visa representar o cenário em que um utilizador acede à página inicial e efetua login no sistema, sendo redirecionado para a página principal. Ao correr este teste são testados os seguintes métodos:

1. GET /
2. POST /users/sign_in
3. GET /

O primeiro GET leva o utilizador para "/users/sign_in" se este não tiver efetuado o login primeiro, ou seja, seria o mesmo que fazer "GET /users/sign_in".

O teste foi corrido com 100 e 250 *threads*, sendo que recorremos aos relatórios gerados pelo JMeter para apresentar os seguintes resultados. O processo de login é constituido pelos 3 pedidos apresentados, sendo que o conteúdo do POST que é feito possui o endereço de email e a password da conta utilizada no teste. Estes 3 pedidos foram encapsulados num processo mais generalizado que representa o login como um todo. Optámos por esta visão de modo a representar de modo mais fidedigno o ato de efetuar login, dado que não é possível efetuar login se a segunda operação falhar. Outra atenção que tivemos, foi o cancelamento de uma *thread* no caso de um dos pedidos falhar.

Threads	Erro	Tempo Resposta Médio (ms)	*Throughput (Trans./s)*
100	0.00%	8129.51	18.96
250	30.28%	8894.58	30.11

Table 3: Sumário do Teste 2 - Com Load Balancer

Threads	Erro	Tempo Resposta Médio (ms)	*Throughput (Trans./s)*
100	0.00%	11013.52	14.56
250	35.96%	8526.46	22.99

Table 4: Sumário do Teste 2 - Sem Load Balancer

Com os resultados obtidos as soluções para permitir escalar o sistema seriam aumentar a memória RAM, como podemos observar pelo *kibana* que é alto ou aumentar o número de instâncias da aplicação.

6 Tolerância a faltas

Para que seja possível garantir que um serviço possua uma elevada disponibilidade mantendo-se com um bom desempenho no atendimento de pedidos, torna-se necessário perceber quais são os **pontos críticos de falha e desempenho** do nosso sistema.

O principal cenário em que um sistema pode falhar é aquele em que um dos componentes que o constitui (ou mais) falhar, não existindo mais do que uma instância de cada um destes elementos. Isto pode acontecer quer por falhas de software/hardware, quer por falhas de energia. No nosso caso em concreto, com a nossa arquitetura inicial, possuímos 2 elementos: a **Base de Dados** e o **Redis**. Estes elementos não possuem qualquer réplica, o que significa que na eventualidade de um ou mais falhar, todo o sistema irá ficar inoperacional.

Outro dos principais pontos de falha e desempenho de uma aplicação são os ***bottlenecks***. Os *bottlenecks* são situações em que um componente da aplicação limita todas as outras, diminuindo a performance do sistema. No nosso caso, o principal *bottleneck* que identificámos é a **Base de Dados**. Dado que a nossa base de dados não está preparada com mecanismos de alta disponibilidade, prevemos que esta poderá a vir ser um ponto crítico do sistema.

A camada aplicacional apresenta vários nodos/réplicas, que através do *load balancer*, irá distribuir a carga aumentando a performance, e irá garantir a disponibilidade do serviço, ou seja, caso um nodo falhe o cliente será reencaminhado para outro nodo, garantido a resiliência desta camada.

Quanto ao ao *load balancer* este é um ponto de falha único, na medida em que sem a sua disponibilidade a performance irá ficar comprometida, bem como a camada aplicacional irá perder resiliência.

Com estes pontos, ficámos com uma melhor noção do que poderá vir a ser um problema no nosso sistema.

7 Conclusão

Numa primeira fase tínhamos como *load balancer* o NGINX, no entanto este já se encontra disponível na camada da aplicação, posto isto uma das recomendações presentes na documentação do GitLab era a utilização do HAProxy como *load balancer*, este permitiu balancear a carga, de modo a se obter uma melhor performance bem como tornar a aplicação resiliente, uma vez que usa um mecanismo de *Health Check* , isto é, se um nodo se encontrar indisponível ele irá detetar e reencaminhar a ligação.

O uso do HAProxy em relação ao NGINX, enquanto *load balancer*, traz mais benefícios, entre os quais: é um serviço *open-source*, com o NGINX não era possível adquirir resiliência, pois seria necessária uma versão *premium*, o NGINX tem como principal uso o de *web server*, no entanto o HAProxy tem como único próposito o *load balacing* o que na nossa arquitetura permitir obter uma melhor performance.

Apesar do NGINX já se encontrar na camada aplicacional, o GitLab permitia o seu *deployment* num servidor externo, porém tivemos dificuldades nesta configuração. O facto do NGINX se encontrar na aplicação também faz sentido pois atua como *web-server* nessa camada e permite estabelecer conexões seguras.

Quanto ao serviço de email, configuramos um servidor SMTP(*Simple Mail Transfer Protocol*) que se encontra ligado à aplicação e que recebe os emails da aplicação, no entanto este não consegue reencaminhar os emails por exemplo para a google, uma vez que o serviço de *Cloud* bloqueia a porta padrão (25), perante esta impossibilidade tentamos usar portas temporárias, bem como abrir outras, mas sem sucesso. Uma solução para este problema seria usar serviços de terceiros como SendGrid, o Mailgun, Mailjet ou o Google Workspace, que possuem documentação, bem como API's compatíveis com a Google Cloud.

Para melhorias futuras seria de considerar tornar a camada de Base de dados e o serviço Redis tolerantes a falhas, por exemplo através de réplicas para garantirem a sua disponibilidade. No caso da base de dados iria existir uma maior complexidade em atingir tolerância a falhas, já que o dados são persistentes daí ser preciso uma sincronização.
Algumas funcionalidade do GitLab também poderiam ser usadas para aumentar a escalabilidade e desempenho, tais como: **Database Load Balancing**, que permite balanceamento de *queries* de leitura em diferentes instâncias da base de dados. Isto resulta numa diminuição da carga na instância principal da base de dados e, consequentemente, aumenta a responsividade. **GitLab Geo** que cria um clone completo do servidor GitLab, apenas com privilégios de leitura, que se mantém sempre sincronizado. Este *add-on* aumenta a velocidade das operações `clone` e `fetch`. A GitLab publicou um vídeo, na plataforma de partilha de vídeos *Youtube*, onde explica e exemplifica os benefícios deste serviço.