Universidade do Minho

Mestrado Integrado em Engenharia Informática

System Deployment and Benchmarking

Fase 2

Provisionamento, *Deployment* e monitorização da aplicação GitLab

Grupo 4

João Alves (A77070)
Gonçalo Raposo (A77211)
Alexandre Dias (A78425)
Hugo Oliveira (A78565)

26 de Dezembro de 2018

Resumo

O presente documento diz respeito à fase 1 do trabalho prático da unidade curricular *System Deployment* e *Benchmarking*.

Este documento consiste numa análise da aplicação *open source GitLab*. Começaremos com uma breve explicação sobre a sua funcionalidade e, posteriormente, será descrita a arquitetura e os diferentes componentes que constituem a aplicação.

Analisar-se-á ainda o ficheiro de configuração do *GitLab* que servirá de complemento aos componentes da arquitetura do *GitLab*.

Por fim serão analisados os componentes críticos e as diferentes implementações da arquitetura do *GitLab* que fazem dele uma aplicação altamente disponível.

Conteúdo

Lista de Figuras

1 Introdução

Numa primeira fase, vamos apresentar a aplicação *GitLab* e os serviços que nos são oferecidos. Posteriormente, será demonstrada toda a arquitetura da aplicação, começando com uma comparação desta com um escritório físico. Mais tarde serão descritos todos os componentes que fazem parte da arquitetura do *GitLab*.

Para completarmos a informação relativa aos componentes da arquitetura do *GitLab*, será apresentado o ficheiro de configuração da aplicação que permite configurar cada um dos diferentes componentes.

Por fim, serão expostos os componentes críticos do *GitLab* bem como as diferentes arquiteturas do *GitLab* que tornam a aplicação altamente disponível.

2 *GitLab*

Antes de podermos descrever toda a arquitetura e funcionamento do *GitLab*, é necessário perceber o que é o ***git***.

2.1 O que é o *git*?

O *git* é um sistema *open source* que permite controlar versões de ficheiros de um projeto de uma forma rápida e eficaz.

O uso de um sistema como este, permite, em projetos que possam envolver vários contribuidores, criar e editar ficheiros em simultâneo, bem como retroceder a versões mais antigas destes. Assim, existe a possibilidade de analisarmos toda a evolução de um projeto.

2.2 O que é o *GitLab*?

O ***GitLab*** é uma plataforma grátis que permite hospedar projetos em servidores locais ou remotos, utilizando o ***git*** para fazer o controlo de versões. O *GitLab* é um concorrente *open source* direto aos serviços *GitHub* e *Bitbucket*.

Para além desta sua principal funcionalidade, a empresa *GitLab* disponibiliza o código da sua aplicação para que qualquer utilizador possa a usar livremente num outro ambiente. Recentemente, foi disponibilizado uma imagem *Docker* que permite instalar e correr o *GitLab* facilmente. Para uma utilização que permita uma alta disponibilidade da aplicação, é necessário alterar e configurar os diferentes componentes da *stack* aplicacional do *GitLab*.

2.3 *GitLab Software Delivery*

Existem atualmente dois tipos de distribuição do *GitLab*, a versão CE (*Community Edition*) e a versão EE (*Enterprise Edition*).

- ***Community Edition***: é uma versão *open source* que não necessita de qualquer apoio externo no processo de *delivery* ou *deployment*. Apresenta funcionalidades reduzidas.
- ***Enterprise Edition***: ao contrários da versão anterior, esta inclui vários serviços para além dos incluidos na versão CE, entre estes, destacamos o suporte técnico especializado 24/7, a gestão adicional de servidores, desempenho e segurança, ferramentas de estatística. Esta versão apresenta diferentes preçários mensais.

3 Arquitetura do *GitLab*

A alta **disponibilidade** e **escabilidade** são, hoje em dia, condições importantes e obrigatórias em qualquer aplicação. Se estes requisitos não são cumpridos, os utilizadores podem querer migrar para outra aplicação idêntica.

Um dos principais aspetos para a alta disponibilidade, consiste no acesso de várias instâncias da aplicação a uma base de dados. Isto é, para prevenirmos que uma aplicação se torne indisponível, são criadas várias réplicas da mesma. Estas réplicas são, no entanto, invisíveis ao utilizador, dado que este apenas vê a aplicação como uma única instância.

Seguindo este raciocínio, o *GitLab* foi desenhado sobre um padrão de várias camadas a trabalhar entre si. Como atualmente é cada vez mais difícil tornar uma aplicação completamente *stateless*, devido à necessidade de armazenamento de informações, é importante separarmos os componentes com estado (como uma base de dados ou sistema de ficheiros) dos componentes sem estado, meramente aplicacionais. Como seria de esperar, isto gera complexidade no desenho e configuração da aplicação.

O *GitLab* possui cerca de 7 camadas aplicacionais distintas, projetadas e configuradas para assegurarem a alta disponibilidade. A imagem em baixo, demonstra a arquitetura do *GitLab*.

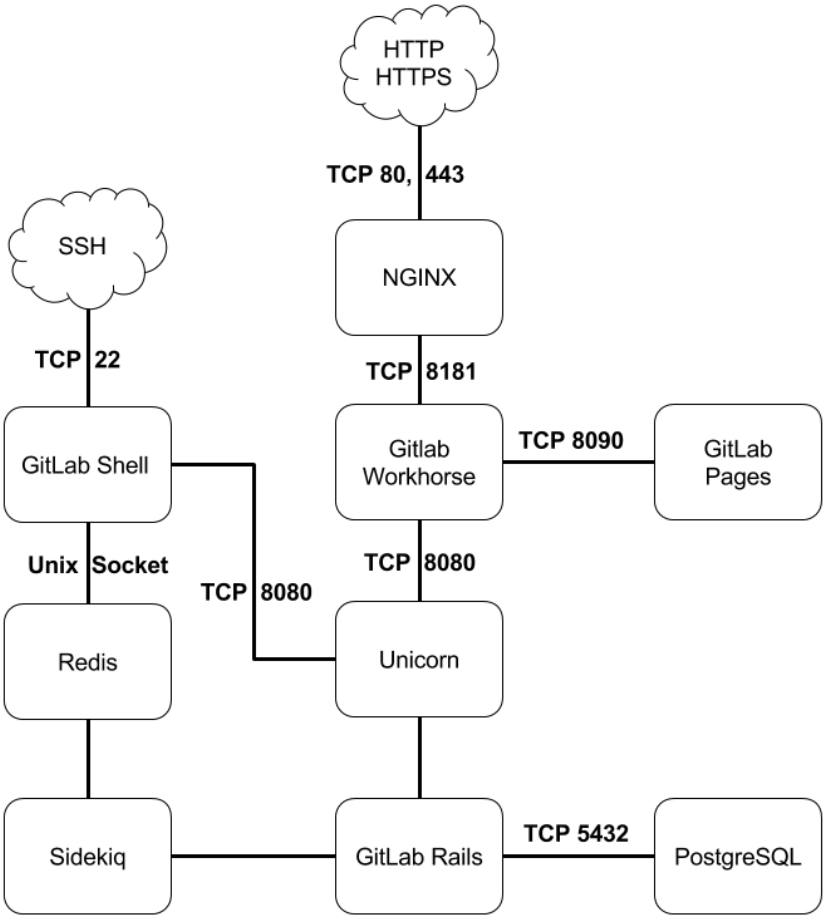

Figura 1: Arquitetura do *GitLab*.

3.1 A arquitetura vista como um escritório físico

Toda a estrutura do *GitLab* pode ser comparada como um escritório físico. Vejamos alguns componentes na seguinte lista.

- **Os repositórios** serão os bens que o *GitLab* gere. Estes armazenam as chamadas *codebases*, que são constituídas por todo o código fonte que é usado para construir uma aplicação;
- ***Nginx*** pode ser visto como uma secretaria, onde os utilizadores se dirigem para requisitar certas operações, que serão realizadas pelos funcionários que lá trabalham;
- **Armazenamento de dados** consiste numa série de armários que contêm ficheiros com informações sobre:
  - os bens que são armazenados;
  - os utilizadores que se dirigem à secretaria.
- ***Redis*** é visto como uma espécie de quadro, que contém uma série de tarefas, para serem executadas pelos funcionários;
- ***Sidekiq*** pode ser comparado como um trabalhador, que recolhe as suas tarefas do quadro (*Redis*), tratando principalmente do envio de emails;
- ***Unicorn worker*** é um operário que trata de tarefas rápidas/gerais, coletadas pelo quadro (Redis). Estas consistem principalmente em:
  - verificar as permissões dos utilizadores , confirmando a sessão de cada um no *Redis*;
  - criar tarefas para o *Sidekiq*;
  - recolher dados presentes no armazenamento de dados.
- ***GitLab-shell*** pode ser visto também como um funcionário que recebe ordens (por *SSH*), comunica com o *Sidekiq* por via do *Redis* e faz alguns pedidos rápidos aos *Unicorn Workers* ou diretamente, ou por via do *Nginx*;
- ***Gitaly*** corresponde a uma espécie de escritório secundário que gere todas as operações feitas através do *git*, desde monitorizar a sua eficiência, até manter cópias dos resultados de operações mais custosas.

Com esta simples introdução aos componentes da arquitetura do *GitLab*, conseguimos visualizar o seu funcionamento.

Nas secções a seguir, serão descritos todos os componentes presentes na arquitetura do *GitLab*.

3.2 Componentes do *GitLab*

A aplicação *GitLab* apresenta uma interface *web* que utiliza várias tecnologias *open source*. A sua aplicação *web* foi desenvolvida utilizando a *framework Ruby on Rails*, baseada num padrão *MVC*. Esta aplicação permite não só, a gestão de repositórios *git*, como também a gestão de comentários, que complementam os diferentes projetos.

Relativamente à estruturação da aplicação *web*, é importante destacarmos os seguintes componentes da camada aplicacional:

- ***NGINX***: é um servidor *web* utilizado para servir conteúdo *web* aos clientes;
- ***GitLab Shell*** e ***GitLab Workhorse***: lidam com os comandos *git* enviados pela a aplicação *web* ou por conexões via *ssh*. Ou seja, comandos como *clone, fork, push* ou *pull*, são redirecionados para um destes componentes, onde serão executados;
  - o *GitLab Shell* permite ainda modificar a lista as chaves autoritativas *ssh* dos utilizadores, verificar um determinada chave pública, limitar comandos *git* aos utilizadores e, por fim, copiar entre o cliente e servidores *Gitaly*;
  - o *GitLab Workhorse* funciona como um *reverse proxy* que trata de pedidos *http* (p.e transferências de arquivos, *git pull/push*, etc). É responsável por assumir uma ligação direta com a aplicação *GitLab*, tentando sempre que possível, diminuir as comunicações entre estes dois (por exemplo ficheiros *Javascript* e *CSS* são diretamente passados ao cliente); assume-se então que os servidores *Nginx* ou *Apache* enviam os pedidos ao *Workhorse* e este ao *backend* do *GitLab*.
- ***Sidekiq***: ferramenta responsável pela gestão e execução de processos *Ruby* da aplicação *GitLab* em *background*. O *Sidekiq* foi introduzido devido a fugas de memória da aplicação *GitLab*. Ou seja, por forma a evitar as fugas de memória, o *Sidekiq* é reiniciado a cada intervalo de tempo previamente definido, permitindo que a aplicação esteja disponível;
- ***Unicorn***: tal como o *Sidekiq*, o *Unicorn* foi introduzido para gerir o uso de memória entre pedidos *web http* e pedidos *git* via *http*. É um *deamon* que corre a aplicação *GitLab* (*master*) e contém um conjunto de *workers*. Cada um é responsável por executar uma tarefa durante um determinado período (*timeout*), caso contrário, a tarefa é atribuída a outro *worker*. O *master* nunca lida com os pedidos recebidos. Tal como referido anteriormente, o *GitLab* apresenta fugas de memória. Ao fim de algum tempo, os processos começam a apresentar falhas, que podem ser controladas com a interrupção do processo (*kill*).

Os componentes descritos em cima fazem parte da camada aplicacional *GitLab*, no entanto, não traduzem a verdadeira essência da aplicação *GitLab*. Isto é, a aplicação ou várias instâncias desta não guardam qualquer tipo de informação a longo prazo. O *frontend* do *GitLab* faz pedidos à camada lógica da aplicação para que possa ser guardado um dado estado ou informação num dos outros componentes da *stack*. O único estado que é guardado na aplicação, é o ficheiro de configuração que abordaremos mais à frente.

3.3 *Redis*

O *Redis* funciona como uma estrutura de armazenamento de dados guardada em memória. É geralmente utilizado para fazer *caching* de dados que, na maioria das vezes, permanecem em memória num curto período de tempo.

Dentro da camada aplicacional do *GitLab*, o *Redis* funciona como um serviço que guarda as sessões ativas de utilizadores e uma lista de tarefas a serem realizadas pelo *Sidekiq*. Isto é, a informação da sessão de um utilizador e as tarefas que estão a ser executadas são guardadas numa ou várias instâncias *Redis*.

Para além destas funcionalidades, o *Redis* permite que os restantes componentes da camada aplicacional guardem o seu estado e o partilhem com o resto dos componentes, por exemplo, através de trocas de messagens. O *Redis* permite que a informação seja consistente e facilmente partilhada com as várias instâncias do *GitLab*.

Uma forma de mantermos a alta disponibilidade do *Redis* consiste na integração de um *Redis Master*, vários *Redis Slaves* e ainda do *Redis Sentinel* (que controla falhas de instâncias *Redis* começando o processo de seleção e atribuição de um *master*).

3.4 Base de dados - *PostgreSQL*

O *Postegres* é um sistema de base de dados relacional *open source*, que permite o controlo e persistência de dados.

Os dados que são armazenados incluem dados do utilizador (como *username*, *email*, *password*, definições de conta, etc), permissões dos repositórios, comentários, problemas encontrados e todos os outros dados que não são diretamente controlados pelo *git*. Todos os dados devem, uma vez mais, estar disponíveis e consistentes para poderem ser recolhidos pelo *GitLab*.

Uma das funcionalidades do *Postgres*, é a existência de um *master* e de um ou mais *slaves*(réplicas). As diferentes réplicas são configurados para se manterem *up to date* e prontas a atuarem como *master* no caso de uma falha do servidor principal. A alta escalabilidade e disponibilidade do *Postgres* é mantida através da utilização do *consul*, *repmgr* e *pgbouncer*.

3.5 Sistema de armazenamento de ficheiros

Em versões antigas do *GitLab*, era utilizado um único sistema de ficheiros local ou distribuído (como o *NFS*), para que pudessem ser armazenados dados e para que o *git* conseguisse controlar repositórios e gerir e versões de ficheiros. Atualmente, o serviço *git* é efetudo pelo *Gitaly*, falado mais à frente. No entanto, todos os restantes dados continuam a ser armazenados num sistema de ficheiros.

Num ambiente onde existem múltiplas instâncias do *GitLab*, é necessário que exista um sistema de ficheiros comum a todas as instâncias, para que estas consigam manipular dados.

O *NFS* é um exemplo de sistema distribuído de ficheiros e que é atualmente utilizado pelo *GitLab*. Neste sistema permitem armazenar dados essenciais ao *GitLab*: armazenamento das chaves públicas *SSH* dos utilizadores, *uploads* dos utilizadores, *GitLab pages*, etc.

3.6 *Gitaly*

O *Gitaly* é um serviço RPC (*Remote Procedure Call*), escrito em *Go*, que surgiu com o intuito de resolver problemas de escalabilidade. Isto é, depois o *GitLab* ter crescido como aplicação, existiu a necessidade de escalar horizontalmente quando verticalmente passou a ser insuficiente e dispendioso. Ainda assim, o *Gitaly* surge com a necessidade de remover falhas no acesso aos repositórios e serviços do *git*, que eram armazenados diretamente num sistema distribuído de ficheiros (*NFS*).

No contexto da aplicação *GitLab*, o *Gitaly* apresenta vários clientes distintos, como é o caso do *GitLab Shell*, *GitLab Workhorse* e *GitLab Rails*. Assim, nenhum outro componente conseguirá escrever ou ler dados do *git*. O diagrama em baixo pretende demonstrar a como é efetuada a comunicação entre clientes e o *Gitaly*.

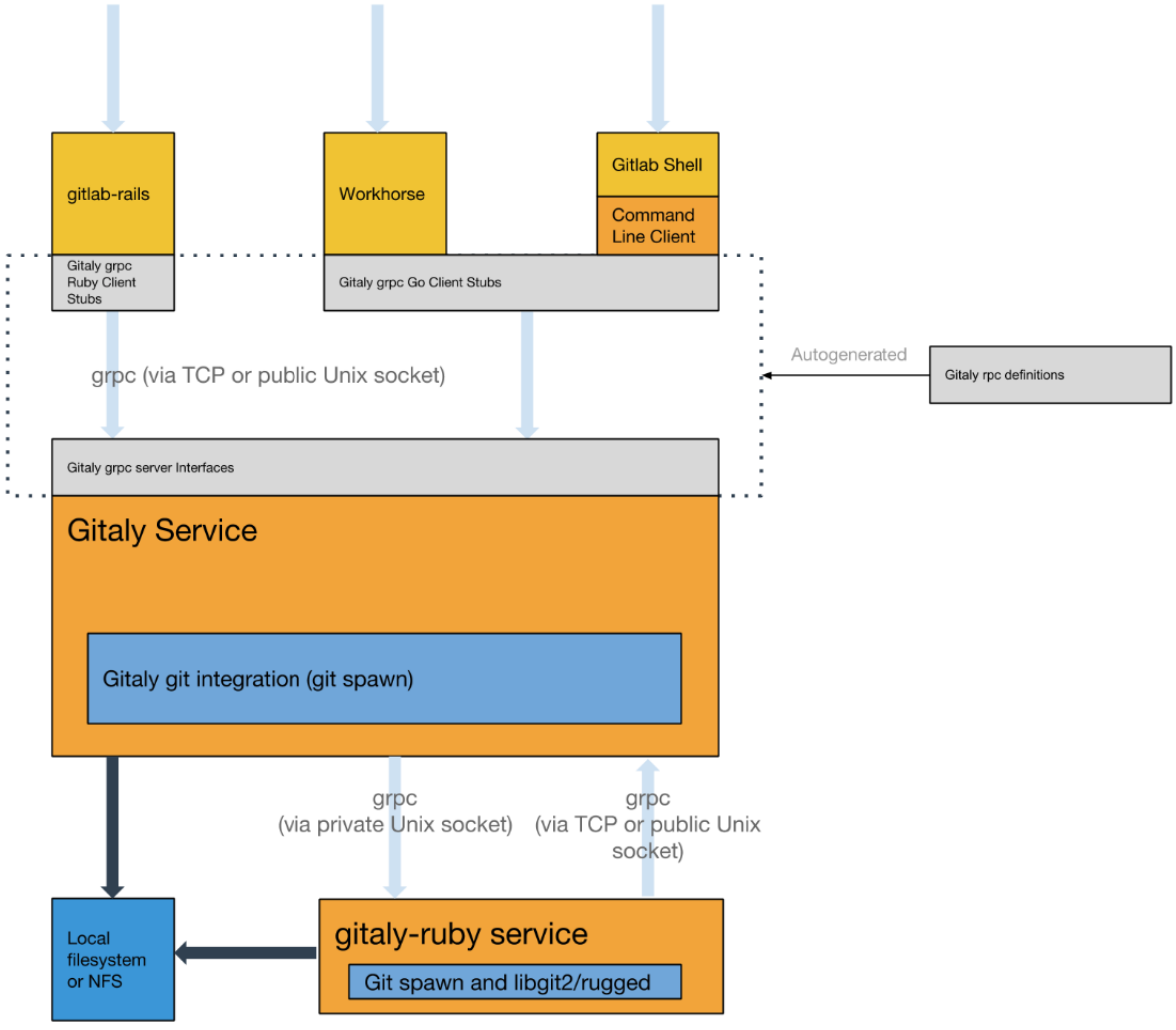

Figura 2: Estrutura do *Gitaly*.

4 Componentes críticos

Através da análise da arquitetura do *GitLab*, é possível detetar os componentes críticos sem os quais comprometeriam a disponibilidade e consistência da aplicação *GitLab*. Grande parte dos componentes referidos em baixo têm em vista a existência de várias réplicas do serviço *GitLab*, pelo que numa abordagem com apenas uma instância alguns dos componentes não seriam considerados à partida críticos.

Componente	**Motivo**	**Solução**
HAProxy	Tornar uma aplicação altamente disponível envolve a existência de múltiplas instâncias de um determinado serviço. O *HAProxy* permite balancear pedidos e serviços para cada uma das potenciais instâncias replicadas, verificando também o seu estado. A existência de uma única instância do *HAProxy* origina um ponto de falha central.	Uma das soluções passa por utilizar o ***keepalived***, que permite a partilha do mesmo endereço IP por diversas instâncias de servidores diferentes. Sempre que uma instância do *HAProxy* falhar, o *Keepalived* deverá ser capaz de atribuir o endereço IP a outra instância ativa.
Redis	A configuração básica de *master-slave* do *Redis* não permite, muitas das vezes, manter a alta disponibilidade de um serviço. Como tal, é necessária a adição de um serviço capaz de resolver problemas relacionados com falhas e replicações de instâncias *Redis*. A falha de uma instância *Redis* compromete a disponibilidade do *GitLab*.	O ***Redis Sentinel*** permite complementar a escolha das instâncias *master* e *slave* de *Redis*, agindo como a única fonte de verdade na hora da decisão. Garante assim a existência de uma instância *Redis*. Este permite ainda indicar aos clientes qual a instância correta que deverá ser utilizada.
Base de dados - *Postgres*	Se a instância da base de dados for comprometida ou estiver inacessível, os dados que os clientes armazenaram na mesma tornam-se indisponíveis, tornando o *GitLab* inutilizável.	Uma forma de ultrapassar a falta de uma instância passa por utilizarmos um *master* e vários *slaves*. A replicação deverá ser efetuada pelo ***repmgr***. Para verificar o estado das instâncias pode utilizar-se o ***consul***. Através do ***pgbouncer*** é possível distribuir os pedidos por diferentes instâncias da base de dados.
Sidekiq	A execução das tarefas em fila de espera no *Redis* é da responsabilidade do *Sidekiq*. Assim sendo, caso o Sidekiq falhe, as tarefas deixarão de ser executadas.	Utilizando várias instâncias do *Sidekiq* é possível evitar este ponto central de falha, uma vez que a aplicação deixa de estar dependente de uma única instância para correr.
Gitaly	O *Gitaly* é responsável por tratar de todos os acessos ao *git* e aos seus repositórios. É muito importante que este esteja disponível às várias instâncias do *GitLab*, dado que estas deixam de ter acesso ao serviço *git*.	Atualmente ainda não existem soluções para garantir a alta disponibilidade do *Gitaly*. No entanto, o *GitLab* tem investigado soluções nesse sentido.

Sistema de ficheiros distribuído - *NFS*	O *GitLab* foi desenvolvido em torno do protocolo *git*. Este protocolo armazena todos os dados diretamente num sistema de ficheiros. Qualquer indisponibilidade do sistema de ficheiros, comprometerá a experiência de utilização do *GitLab*. Este componente é, por si só, dos mais importantes entre os restantes anteriormente indicados.	Serviços como o ***GPFS*** e ***HighlyAvailableNFS***, permitem que os dados estejam sempre disponíveis. No caso do *GPFS*, é possível distribuir os dados em diferentes instâncias assegurando redundância e tempos leitura mais baixos. No entanto, ainda que seja utilizado este serviço, é necessário que existam pelo menos 2 nós a persistirem dados.

Tabela 1: Componentes críticos numa arquitetura distribuída do *GitLab*.

5 Arquitetura adotada

De seguida, é apresentada a arquitetura que foi adotada para a realização do provisionamento e *deployment* da aplicação *GitLab*.

Para que seja possível tornar o *GitLab* altamente disponível, adotamos uma arquitetura distribuída, semelhante à utilizada atualmente pela aplicação *GitLab*. Esta arquitetura apresenta um acentuado número de nós computacionais que requerem alguma gestão e monitorização. Não foram utilizados *containers* durante a fase de provisionamento e *deployment*.

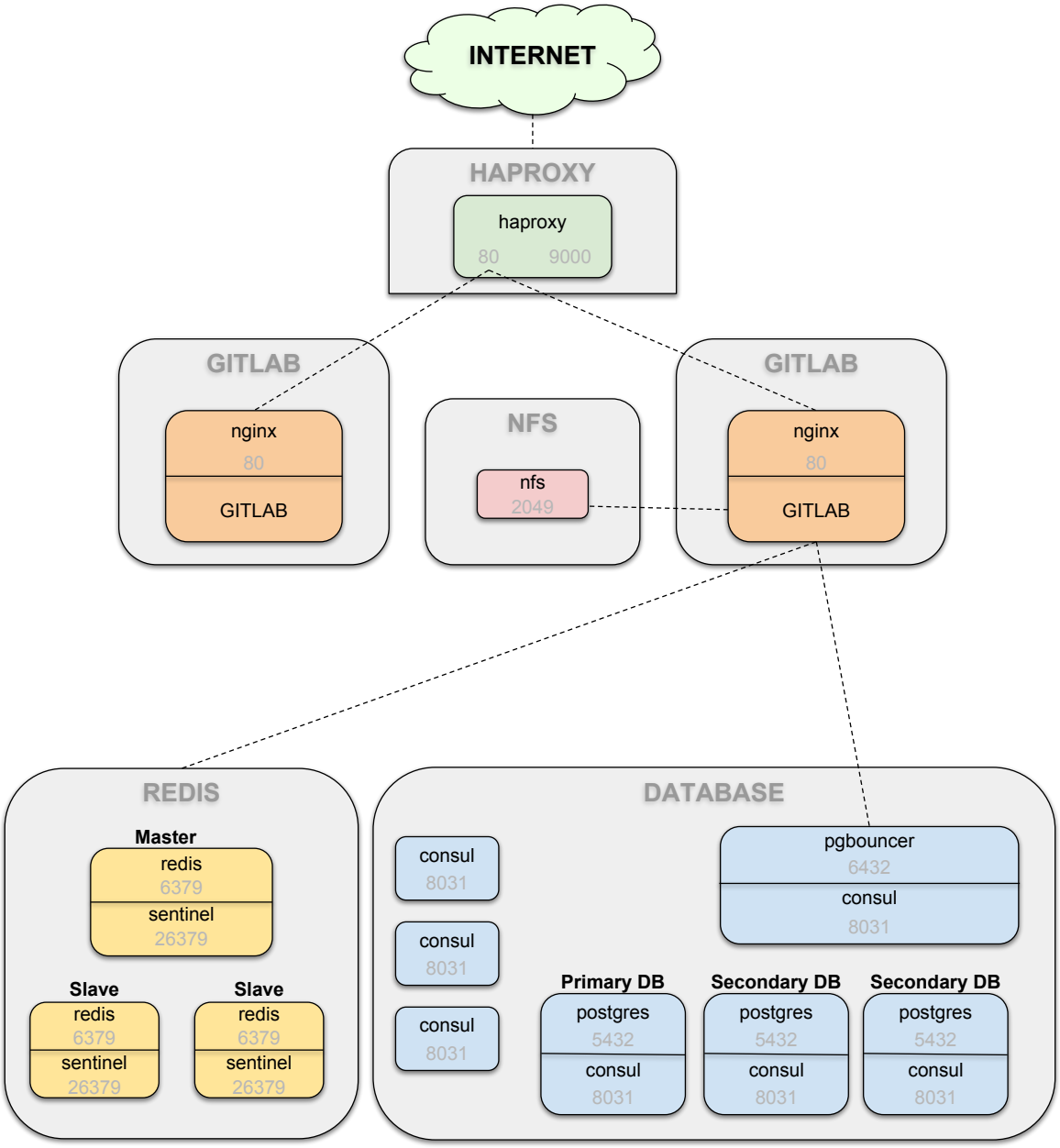

Figura 3: Arquitetura adotada.

Com base nos componentes do *GitLab* e na arquitetura adotada, vamos proceder à explicação de um possível provisionamento do *GitLab*.

5.1 *HAproxy*

A introdução de uma instância a realizar *load balancing* é uma das tarefas mais importantes quando se pretende tornar uma aplicação altamente disponível.

Como visto anteriormente, uma instância com o *HAproxy* é, por si só, um componente crítico e como tal deverá ser usada uma ferramenta como o *keepalived* que permite verificar o estado de uma máquina e automaticamente a colocar num estado de *standby* sempre que existe uma falha. A adição de um único servidor *HAproxy* com o *keepalived* não garante, no entanto, a alta disponibilidade. Para garantirmos isso, é necessário adicionarmos mais uma máquina a servir as mesmas necessidades. Para isso seria necessário existir um *floating ip*. Em baixo segue um exemplo que sugere a estrutura destes componentes.

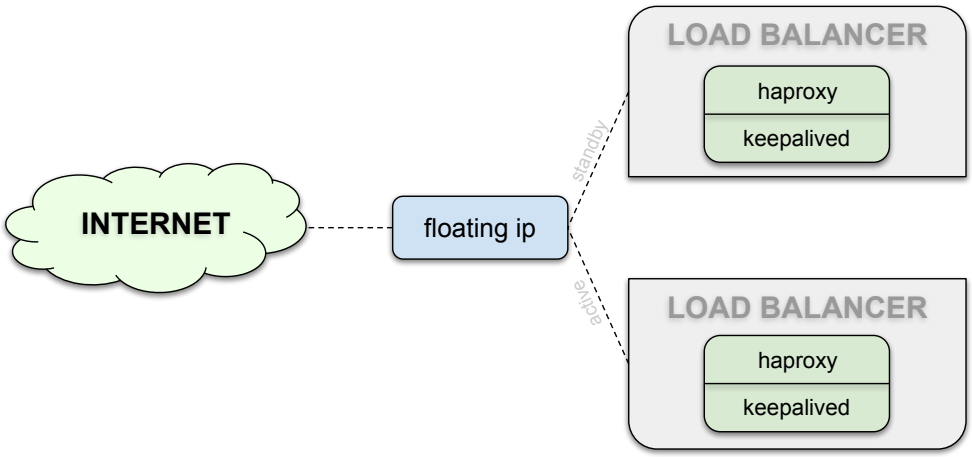

Figura 4: Estrutura ideal para *load balacing*.

Na estrutura adotada adicionamos unicamente uma instância a servir o *HAproxy*. A solução ideal para seria a apresentada em cima. Nesta estrutura não existe limite para o número de instâncias a serem adicionadas, garantindo a alta escalabilidade e resiliência.

5.2 Aplicação *GitLab*

A introdução de *load balancers*, como mostrado anteriormente, permite que instâncias que sirvam a aplicação *GitLab* não sofram um grande *overhead* caso exista um elevado número de clientes. O *HAproxy* possui um mecanismo de verificação do estado dos servidores aplicacionais, garantindo que não é encaminhado tráfego para servidores em baixo.

A estrutura adotada inclui duas instâncias a servir o *GitLab*, no entanto poderiam ser adicionadas mais instâncias, conforme as necessidades.

Como vimos anteriormente, existem inúmeros componentes que são vitais ao funcionamento do *GitLab*, como é o caso do *Nginx*, *Gitaly*, *GitLab Shell*, *GitLab Workhorse*, *Sidekiq*, *Unicorn*, *Redis* e base de dados.

À exceção da base de dados, do *Gitaly*, do *Redis* e do *Sidekiq*, nenhum dos outros componentes requer ser replicado ou até mesmo ser colocado numa instância isolada, uma vez que acabam por ser intrínsecos ao *GitLab*. No entanto, não existem indicações

de que é uma abordagem incorreta. Abordaremos mais à frente componentes como o *Gitaly*, *Redis* e base de dados.

Relativamente ao *Sidekiq*, seria favorável colocar este serviço em diferentes máquinas. Esta ação tornaria a aplicação *GitLab* mais rápida a processar pedidos. Estes pedidos estão "armazenados"temporariamente no *Redis* e são posteriormente tratados pelo *Sidekiq*, pelo que, quantas mais instâncias existirem a correr este serviço, mais rápido será o sistema. Em baixo segue um exemplo que sugere a estrutura destes componentes.

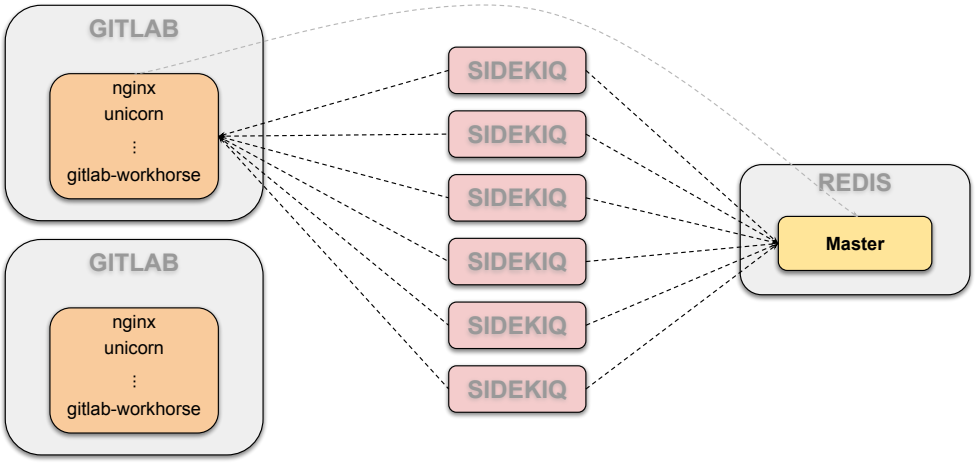

Figura 5: Estrutura ideal para executar pedidos rapidamente.

A vantagem da estrutura apresentada em cima, consiste na distribuição de diferentes tarefas, por cada uma das instâncias a servir o *Sidekiq*.

5.3 *Network File System*

A estrutura apresentada na figura 3, inclui unicamente uma instância a partilhar o sistema de ficheiros com a rede local. Esta máquina é, por si só, um ponto de falha grave, pois toda a aplicação ficará inconsistente com a sua falha, como mencionada anteriormente.

Uma das soluções para corrigir este ponto de falha, seria utilizar um *Highly Available NFS*. Isto é, permitir que um serviço como o *NFS* esteja sempre disponível aos utilizadores de *NFS*. Uma forma de fornecer um único endereço para os clientes para terem acesso ao *NFS*, passa por utilizar um endereço virtual juntamente com o serviço *heartbeat* que permite analisar o estado das instâncias. Para garantirmos a replicação dos dados podemos utilizar o serviço *DRBD*. Em baixo segue um exemplo que sugere a estrutura destes componentes.

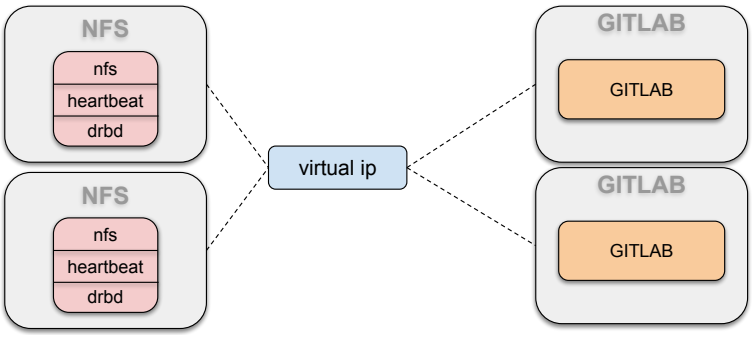

Figura 6: *High Available NFS*.

5.4 *Gitaly*

5.5 *Redis*

A estrutura adotada inclui três instâncias *Redis*, nomeadamente uma instância *master* e duas instâncias *slaves*. Juntamente com o serviço *Redis*, foi adicionado o *Redis Sentinel* que garante a alta disponibilidade do serviço. Este permanece à escuta de falhas das instâncias. Nesta situação inicia o processo de seleção de um novo *master*. Um exemplo desta estrutura encontra-se na figura 3.

5.6 Base de dados

O *GitLab* recomenda a utilização do *PostgreSQL* como sistema de gestão de base de dados. Nesse sentido, foram exploradas várias configurações que permitissem assegurar a elevada disponibilidade.

A figura 3 ilustra a arquitetura que o grupo planeou para uma configuração da base de dados que garante grande disponibilidade. Relativamente à arquitetura da base de dados, esta é constituída por um total de sete instâncias:

- três instâncias a servir o *postgres*, nomeadamente um *master* e dois *slaves*;
- três instâncias a servir o *consul*, que ficam à escuta de falhas das instâncias mencionadas anteriormente;
- uma instância a servir o *pgbouncer*; as instâncias *GitLab* conectam-se ao *pgbouncer* que encaminha os pedidos para a base de dados *master*;

É ainda importante referir, que nas instâncias a replicação entre os servidores *postgres* é efetuada pelo serviço *repmgrd*, pelo que se encontra ativo nestas máquinas, assim como o *consul* que serve de cliente para as instâncias principais.

No decorrer do provisionamento e *deployment* da base de dados, o *pgbouncer* tornou-se problemático, uma vez que a sua tentativa de inicialização resulta sempre num *timeout*. Numa tentativa de resolução do problema, foi encontrada um *board*[1] com erros semelhantes que, até ao dado momento, não possuem solução oficial da equipa do *Omnibus* do *GitLab*. Para solução deste problema, decidimos apenas utilizar uma instância a servir a base de dados.

[1] `https://goo.gl/a9uF5H`

6 Ferramentas de monitorização

No que concerne à monitorização das instâncias presentes na estrutura apresentada na figura 3, foram utilizadas, em conjunto, as seguintes ferramentas fornecidas pelo *elastic*[2]: *Elasticsearch*, *Kibana* e *Beats*.

- ***Elasticsearch***: aplicação *RESTful* que regista e fornece os dados fornecidos pelos diferentes *Beats*;
- ***Kibana***: aplicação *web* que permite exibir os dados recolhidos e analisados pelo *Elasticsearch* ao utilizador;
- ***Beats***: ferramenta que recolhe vários tipos de dados sobre uma dada instância, um dado serviço, um dado ficheiro, etc.

Numa arquitetura de dimensão semelhante à do *GitLab*, é indispensável a monitorização das diferentes instâncias. Só assim é possível identificar o estado dos componentes.

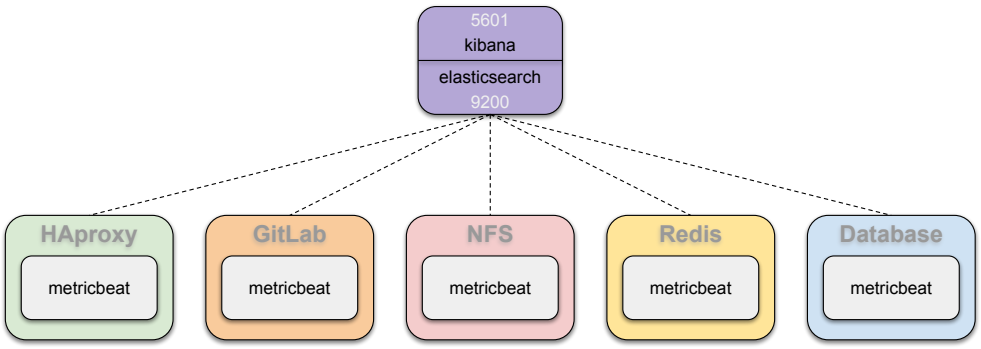

Figura 7: Estrutura das ferramentas de monitorização.

Assim sendo, para recolher informações sobre o estado de uma determinada instância, como a frequência do cpu, utilização de RAM, espaço em disco e tráfico I/O, utilizamos a ferramenta *metricbeat*. Esta, após instalada em cada uma das máquinas cujas as instâncias se pretende monitorizar, envia os dados e estatísticas recolhidas, relativas ao sistema operativo e à maquina, para um host configurável no ficheiro *metricbeat.yml*. No caso implementado, o host, corresponde a um servidor externo que recolhe informação de várias instâncias através da porta 9200, servidor este adicionado à arquitetura anteriormente desenhada do *GitLab* e cuja a única funcionalidade será a monitorização de todas as outras instâncias da aplicação. Assim, através do *Elasticsearch*, este servidor recolhe todos os dados pretendidos de todos os servidores monitorizados identificando e redirecionando estes, para a aplicação web *kibana* instalada na mesma maquina. Desta forma, qualquer problema da arquitetura poderá ser facilmente reconhecido uma vez que o utilizador tem acesso em tempo real ao estado de todas as máquinas da arquitetura de uma forma personalizada, permitindo assim uma analise eficiente de toda a informação recolhida pelos diferentes *beats*.

[2] `https://www.elastic.co/products`

7 Ferramentas de avaliação

Para avaliar o desempenho da aplicação implementada, irá ser utilizada uma ferramenta de geração de carga e medição de desempenho, o Apache JMeter. Através deste, é possível automaticamente gerar cargas para testar diversos tipos de serviços e analisar o desempenho dos mesmos.

Uma vez que o Apache JMeter permite a realização de testes que utilizem diferentes serviços, este pode ser utilizado para testar componentes individualmente, ou para avaliar a performance da aplicação como um todo. Por exemplo, uma vez que o postgres tem suporte JDBC é possível utilizar o Apache JMeter para gerar um conjunto de instruções SQL e alimentar a base de dados com os mesmos para avaliar o seu desempenho individualmente, como um componente isolado, no entanto, uma vez que esta ferramenta suporta gravação de comandos num browser, é possível simular um conjunto de ações via interface web, que possam originar o mesmo tipo de carga na base de dados, mas no entanto testam a aplicação como um todo avaliando o desempenho que seria percebido pelo cliente ao invés de avaliar o desempenho de um único componente.

8 Ferramentas de instalação automática

9 Conclusão

Numa primeira fase, tornou-se claro que, antes de qualquer *deployment* de uma plataforma com o *GitLab*, é necessário um estudo prévio da arquitetura por motivos de *High Avaibility*, conforme os objetivos futuros da implementação desta. Por estas razões, é necessário dissecar a arquitetura nos seus vários componentes e compreender as funcionalidades e objetivos destes, assim como as relações existentes entre os vários. Apenas desta forma, o utilizador irá conseguir criar uma plataforma *GitLab* que corresponda aos seus interesses.

Assim sendo, o *GitLab* é facilmente reconhecido como um sistema distribuído bastante complexo em constante desenvolvimento, com vista a poder responder a cada vez mais utilizadores. Estes podem ser desde simples programadores com projetos pessoais, até grandes empresas tecnológicas, onde cada componente pode ser reutilizado noutro tipo de plataformas, devido ao cuidado com que foi desenvolvido e à sua documentação.

Outro grande fator que deve motivar o estudo da arquitetura é o custo financeiro da própria escalabilidade, onde este pode escalar e ter um desempenho que não corresponda às expectativas de quem a está a implementar.

Para finalizar, após um estudo intensivo da arquitetura e após identificar os componentes críticos desta e tendo em consideração todos os tópicos descritos neste relatório, é seguro dizer que o processo de *deployment*, a realizar numa próxima fase deste projeto, irá respeitar o orçamento disponível para a equipa de trabalho.

10 Referências

Bertsche, Ryan. IBM Corp. (2017). GitLab: Highly Available Architecture.

GitLab Architecture Overview, GitLab Documentation. Acedido em 26 de Dezembro de 2018, em `https://goo.gl/thmr2N` e `https://goo.gl/iyR4zN`

GitLab High Availability, GitLab Documentation. Acedido em 26 de Dezembro de 2018, em `https://goo.gl/Dt93fF`

11 Anexos