Universidade do Minho

Mestrado Integrado em Engenharia Informática

System Deployment and Benchmarking

Fase 1

Deployment* da aplicação *GitLab

Grupo 4

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

9 de Novembro 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* onde não apoio de uma qualquer empresa 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, *performance* 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 *clustering* da aplicação. Isto é, para prevenirmos que uma aplicação se torne indisponível, são criadas várias réplicas da mesma. Estas réplicas são, no entanto, invisiveis ao utilizador, dado que este apenas vê a aplicação como uma única instância.

Seguindo este raciocinio, o *GitLab* foi desenhado sobre um padrão de várias camadas a trabalhar entre si. Como hoje em dia é impossível tornar uma aplicação *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*.

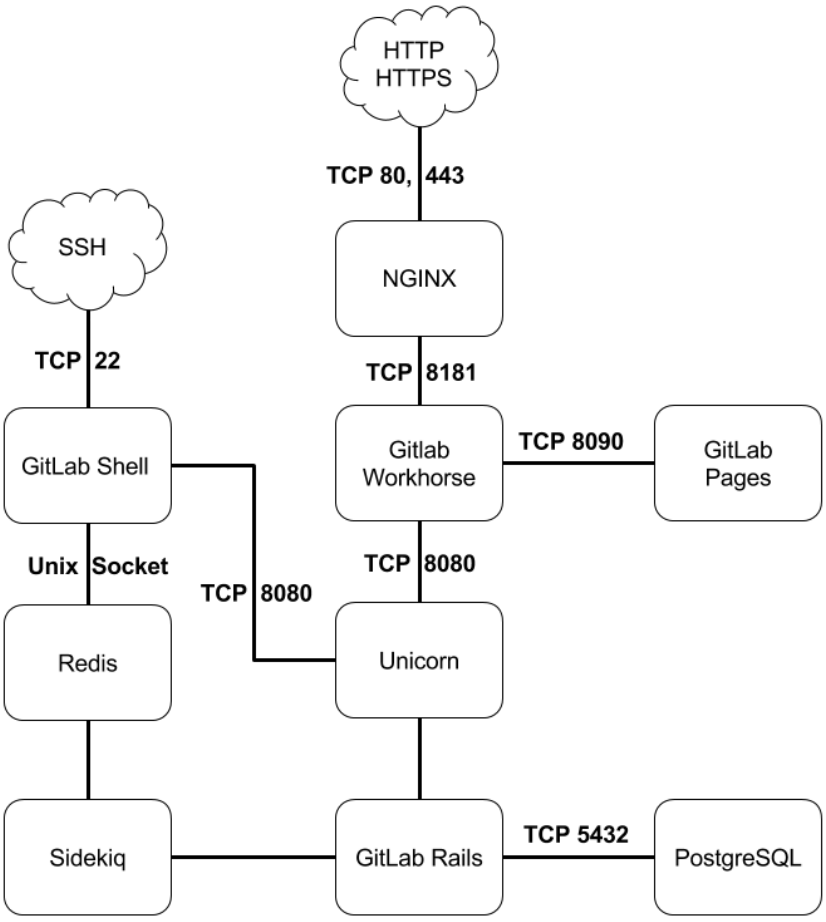


Figura 1: Arquitetura do *GitLab*.

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 *GitLab frontend*

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 *memory leaks* 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 e uso de memória entre pedidos *web http* e pedidos *git* via *http*. É um *deamon* que corre a aplicação *GitLab* (*master*) e contém um conjuntos 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 *memory leaks*. Ao fim de algum tempo, os processos começam a apresentar falhas, que podem ser controladas com a interropção 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.

3.2.1 *HAProxy*

O *HAProxy* (*High Performance TCP/HTTP Load Balancer*) é um servidor *proxy* para *web* que oferece soluções fiáveis como: alta escalabilidade, balanceamento de carga e *proxying* para aplicações que usam *TCP* e *HTTP*.

No contexto da aplicação *GitLab*, o *HAProxy* tem várias funcionalidades e funciona como um *gateway* que distribui tráfego e pedidos pelos vários servidores que correm a aplicação *GitLab*. Em baixo, encontra-se uma imagem que pretende demonstrar o funcionamento do *HAProxy*, destacado como *load balancer* que recebe vários pedidos *SSH* e *HTTP* e encaminha-os para os servidores que correm o *GitLab*.

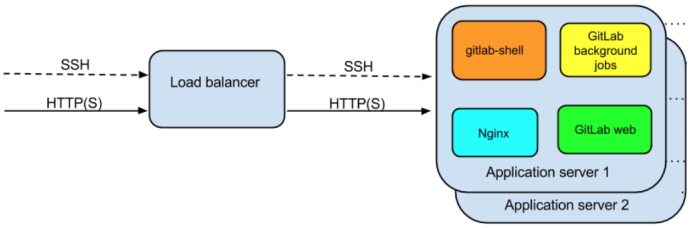


Figura 2: Exemplo de funcionamento do *HAProxy*.

Para além das funcionalidades descritas em cima, o *HAProxy* funciona como um *edge-node* entre a rede externa (*internet*) e a rede interna do *GitLab*. Isto é, o *HAProxy* irá negar qualquer tentativa de acesso (através de *sockets unix*) aos servidores *frontend* do *GitLab*.

3.2.2 *Keepalived*

Juntamente com o *HAProxy*, existe a aplicação *Keepalived* que garante que o serviço *GitLab* está sempre disponivel. Isto é, permanece ativas duas instâncias idênticas do *HAProxy*, garantido assim, que pelo menos uma delas se encontra configurada para um determinado IP, tornando o *HAProxy* eficiente. Se uma instância falhar, outra é automaticamente configurada para o IP anterior (único).

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. Tal como alguns dos componentes anteriores, é também importante que toda a informação seja consistente e facilmente partilhada com as várias instâncias do *GitLab*.

3.3.1 *Redis Sentinel*

O *Redis Sentinel* tem uma funcionalidade muito semelhante ao *Keepalived*. Para garantirmos uma alta disponibilidade, é necessário que para além de existir um *Master* e um *Slave* do *Redis*, exista algo à escuta de falhas de instâncias *Redis*, começando automaticamente o processo de *failover*, que consiste em atribuir uma nova instância aquando uma falha.

Com o *Redis Sentinel*, consegue-se garantir a existência e disponibilidade de uma instância *Redis* para um cliente, tornando abstratas todas as falhas.

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 exibidos e validados corretamente para a aplicação *web* do *GitLab*.

Num cluster onde possam existir várias instâncias do *Postgres* (como o *master* ou *slaves*/réplicas), existe também uma instância do *HAProxy*. Tal como vimos anteriormente, o *HAProxy* permite o balanceamento de carga e permite transmitir os pedidos a um grupo de nós previamente conhecidos. Ou seja, qualquer instância da aplicação *GitLab* irá comunicar com este *HAProxy* acreditando que este é a base de dados.

Os diferentes *slaves* são configurados para se manterem *up to date* e prontos a atuarem como *master* no caso de uma falha do servidor principal. A alta escalabilidade e disponibilidade do *Postgres* é mantida através da utilização de instâncias *HAProxy* e *Patroni*.

3.4.1 *Patroni*

O *Patroni* é uma *framework* escrita em *python* usada para facilitar e automatizar processos como replicação e falhas de instâncias do *PostgreSQL*.

Ainda que o *Postreges* suporte mecanismos de *master/slave*, a adição ou remoção de novos nós que contêm réplicas dos dados é uma tarefa dificil e que requer configurações que podem induzir a erros. Com a integração do *Patroni*, é possivel utilizar aplicar estes mecanismos de replicação automaticamente sem necessidade dos habituais processos manuais de configuração. Além disso, após uma falha, o *Patroni* é capaz de resolver o processo de recuperação, selecionando um novo *master*.

Esta *framework* corre em conjunto com as instâncias do *Postgres*. Para que seja possível reconhecer e atualizar o estado de cada uma destas instâncias, o *Patroni* recorre ao ***Consul***, *ETCD* ou ***Zookeeper***.

```
+---------------+--------+-------------+--------+----------+-----------+
|    Cluster    | Member |     Host    |  Role  |  State   | Lag in MB |
+---------------+--------+-------------+--------+----------+-----------+
| pg-ha-cluster |  pg-1  | 10.142.0.11 |        | running  |       0.0 |
| pg-ha-cluster |  pg-2  | 10.142.0.10 | Leader | running  |       0.0 |
| pg-ha-cluster |  pg-3  | 10.142.0.09 |        | running  |       0.0 |
+---------------+--------+-------------+--------+----------+-----------+
```

Figura 3: Exemplo de uma tabela com o estado do *Patroni*.

3.5 Sistema distribuído de ficheiros

No núcleo do *GitLab* encontra-se o *git*. Como vimos anteriormente, o *git* é uma ferramenta que permite o controlo de versões de ficheiros. O protocolo do *git* utiliza um sistema de ficheiros para guardar ficheiros. Como seria de esperar, todas as instâncias do *GitLab* utilizam o mesmo sistema de ficheiros.

Para além do sistema de ficheiros utilizado pelo *git*, existem ainda outras pastas partilhadas que permitem guardar outros ficheiros necessários ao funcionamento do *git* e do *GitLab*. Incluem-se: a pasta que armazena as chaves *SSH*, recursos estáticos que são carregados (como imagens, ficheiros *Javascript, CSS*) e outros ficheiros que necessitam de estar presentes em todas as instâncias do *GitLab*.

3.6 *ETCD*

Ferramentas como ***ETCD***, ***Consul*** e ***Zookeeper*** permitem guardar informações num estilo *key-value* dentro de um *cluster*. Ou seja, podemos guardar o estado de cada uma das instâncias do *Postegres* nestas ferramentas.

As ferramentas mencionadas em cima são muito semelhantes. Optamos por escolher o *ETCD* para descrever o funcionamento básico de um componente deste tipo. Este aprensenta um *time to live* (configurável) para o armazenamento dos dados. Isto é, ao fim de um tempo préviamente estabelecido, todos os dados sobre os estados das máquinas, é removido. Isto garante que as instâncias *Patroni* verificam continuamente o estado e o atualizam. Com a utilização deste recurso, o *Patroni* consegue reconhecer facilmente quem são as máquinas disponíveis bem como o líder atual.

Como podemos verificar, componentes como *Consul*, *ETCD* e *Zookeeper* são extremamente importantes na estrutura do *GitLab*, pelo que devem existir várias instâncias destes.

3.7 *Gitaly*

O *Gitaly* é um serviço RPC (*Remote Procedure Call*), escrito em *Go*, que trata todas as chamadas ao *git*, feitas pelo *GitLab*, ou seja, é responsável por fazer todos os acessos aos repositórios *git*.

Os componentes *GitLab-Shell* e *GitLab-Workhorse* funcionam como clientes para o *Gitaly*.

O diagrama em baixo pretende demonstrar a como é efetuada a comunicação entre clientes e o *Gitaly* para um acesso ao serviço *git*.

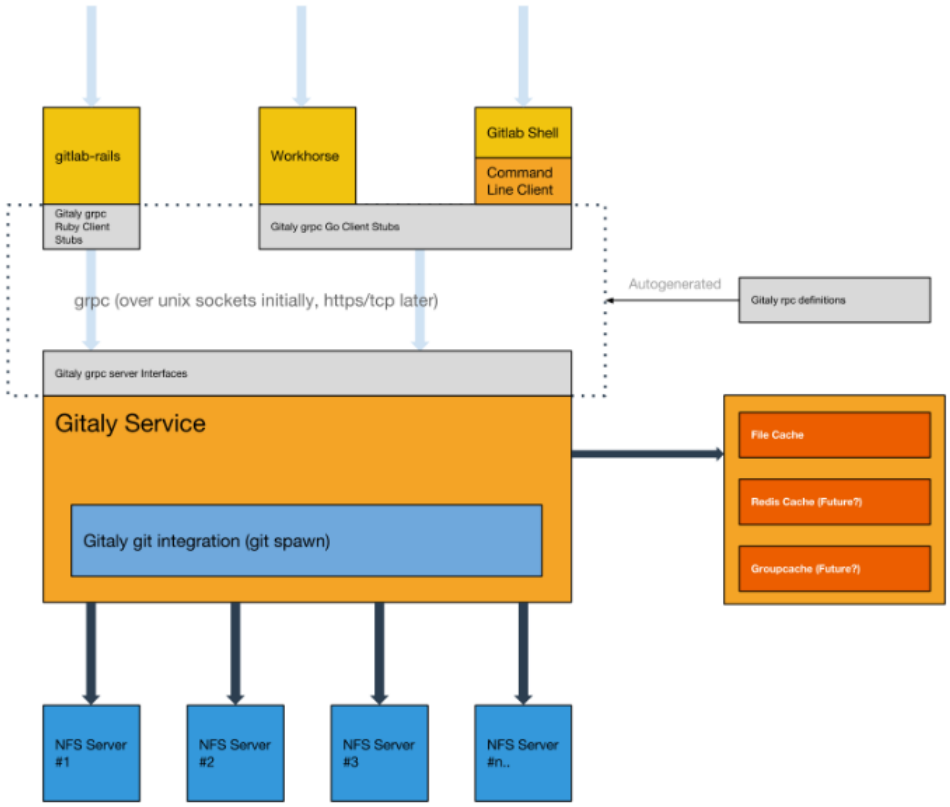


Figura 4: Funcionamento do *Gitaly*.

O *Gitaly* irá atuar como uma passagem para os repositórios. Possui ajudantes que permitem agilizar todo o processo de troca de informações. Estes são conhecidos por ocasionalmente terem fugas de memória, pelo que o *Gitaly* irá reiniciar estes ajudantes quando estes excederem uma certa quantidade máxima.

Para possibilitar a comunicação entre os ajudantes e o servidor *Go*, o *Gitaly* usa a framework *gRPC*, que permite aos ajudantes chamarem métodos sobre objetos no servidor, e executa-los como se o método fosse local, possibilitando assim uma implementação distribuída da aplicação.

Finalmente, deve ser considerada a criação de alternativas ao *Gitaly* na implementação, uma vez que no caso de haver falha de rede, este constituirá um ponto central de falha, e os pedidos não serão atendidos.

4 Ficheiros de configuração

Para uma melhor interpretação do ficheiro de configuração, utilizamos o *Omnibus*, um *package* de instalação do *GitLab* para podermos configurar os parâmetros da aplicação *GitLab* diretamente.

O ficheiro principal de configuração é o `gitlab.rb`. Cada instância do *GitLab* irá possuir o seu ficheiro de configuração. Este ficheiro encontra-se na seguinte diretoria:

```
/etc/gitlab/
```

Sempre que for efetuada qualquer alteração neste ficheiro, devemos executar o seguinte comando:

```
$ gitlab-ctl reconfigure
```

De seguida, iremos abordar algumas das possíveis configurações que são permitidas no *GitLab*.

4.1 Endereço *url* para acesso ao *GitLab*

Para que os utilizadores consigam aceder remotamente à aplicação *web* do *GitLab*, é necessário definir o *url*. De notar que, caso pretendam um acesso externo, é necessário registar este endereço *url* para que seja possível a resolução de nomes (DNS), ou seja, associar este *ip* ao nome.

Para isto, basta adicionar a seguinte linha ao ficheiro de configuração:

```
external_url "http://gitlab.example.com"
```

É ainda possível definir um *url* relativo:

```
external_url "http://example.com/gitlab"
```

4.2 Integração com a base de dados

Como foi abordado anteriormente, o sistema de base de dados recomendado é o *PostgreSQL*.

Caso se pretenda utilizar uma máquina diferente para correr um sistema de base de dados, é necessário efetuar os seguintes passos:

- `db_host`: representa o *ip* ou nome do nó/máquina;
- `db_port`: representa a porta por onde está a ser fornecido o serviço *postgres*;
- `db_database`: representa o nome do nome dado à base de dados;
- `db_username` e `db_password`: representa os dados de acesso à base de dados;
- `postgresql['enable']`: opção que permite desativar o *postgres* nativo da aplicação, permitindo que seja outra instância a servi-la.

```
gitlab_rails['db_adapter'] = "postgresql"
gitlab_rails['db_host'] = "10.0.0.101"
gitlab_rails['db_port'] = 5432
gitlab_rails['db_database'] = "gitlab-prod"
gitlab_rails['db_username'] = "gitlab-user"
gitlab_rails['db_password'] = "gitlab-password"

postgresql['enable'] = false
```

Figura 5: Exemplo de configuração para o *PostgreSQL*.

O *package omnibus*, por *default*, não irá importar o *schema* da base de dados do *GitLab* nem irá criar um administrador, necessário para a aplicação.

Como tal, é necessário proceder à criação da base de dados. É importante referir que este processo deverá ser apenas efetuado uma única vez, pelo que, os seguintes excertos de código apenas deverão ser adicionados num único ficheiro de configuração de um instância *GitLab*.

```
# especificar uma password de utilizador root
gitlab_rails['initial_root_password'] = 'nonstandardpassword'
```

Por fim, executar o comando `$ gitlab-rake gitlab:setup`.

No caso de existirem vários servidores *GitLab* a partilharem a mesma base de dados e for necessário efetuar a migração de dados de uma base de dados, é importante limitar o número de máquinas que irão efetuar este processo. Para isso devemos, para cada máquina que corre o *GitLab*, adicionar o seguinte:

```
# ativar ou desativar a migração automática
gitlab_rails['auto_migrate'] = false
```

4.3 Integração com um servidor SMTP

O *GitLab* permite que seja utilizado um servidor de *emails* externo, que esteja a correr noutro nó computacional.

Os parâmetros necessário à configuração são muito semelhantes aos da configuração de um servidor *SMTP* ou *IMAP*:

```
gitlab_rails['smtp_enable'] = true
gitlab_rails['smtp_address'] = "smtp.mailtrap.io"
gitlab_rails['smtp_port'] = 2525
gitlab_rails['smtp_user_name'] = "username"
gitlab_rails['smtp_password'] = "password"
gitlab_rails['smtp_domain'] = "smtp.mailtrap.io"
gitlab_rails['smtp_tls'] = true

gitlab_rails['smtp_openssl_verify_mode'] = 'none'
gitlab_rails['smtp_enable_starttls_auto'] = false
gitlab_rails['smtp_ssl'] = false
gitlab_rails['smtp_force_ssl'] = false
```

O exemplo em cima está apenas a utilizar o protocolo de segurança *TLS*, no entanto é possível utilizarmos também o protocolo *SSL*.

O *GitLab* disponibiliza *online* vários exemplos de configuração de servidores *SMTP* de vários *providers*.

4.4 Integração com o *Nginx*

Por *default*, o *package* de instalação *omnibus* instala um *bundle Nginx*. Para ser possível utilizar um conjunto de servidores externos *Nginx* ou *Apache*, é necessário os adicionar ao grupo de servidores *web* `gitlab-www`.

Para se juntar novos servidores ao grupo descrito em cima, é necessário os adicionar ao grupo principal e, se necessário, desativar o *bundle Nginx*. De notar que, para servidores que corram *Debian/Ubuntu* o grupo será `www-data`. Para servidores que corram *RHEL/CentOS* o grupo será `nginx`.

Em baixo encontra-se um exemplo de configuração a ser adicionado ao ficheiro `gitlab.rb`.

```
# desativar o bundle
nginx['enable'] = false

# adicionar o nó ao grupo externo
web_server['external_users'] = ['www-data']

# adicionar o servidore aos servidores proxies confiáveis (IPv4 e/ou IPv6)
gitlab_rails['trusted_proxies'] = ['192.168.1.0/24', '192.168.2.1']
```

Por *default* o *Nginx* aceita pedidos vindos de qualquer endereço *ip*. No entanto, podemos especificar a lista de endereços válidos.

```
# receber pedidos de todos os endereços IPv4 e IPv6
nginx['listen_addresses'] = ["0.0.0.0", "[::]"]
```

Caso uma instância do *GitLab* esteja a correr atrás de um *reserve proxy*, ou seja, os pedidos são encaminhados do *proxy* para o servidor *GitLab* e vice versa, é necessário alterar a porta *default* (80 para *http* e 443 para *https*) para uma porta válida. Em baixo segue um exemplo.

```
nginx['listen_port'] = 8081
```

4.5 Integração com o *GitLab-Shell*

O *GitLab-Shell* permite lidar com as sessões *SSH* do *git* e, mais importante ainda, alterar a lista das chaves autoritativas. O ficheiro de configuração permite várias alterações no que diz respeito ao funcionamento do *GitLab-Shell*. Entre as quais destacamos: alteração da diretoria onde são mantidas as chaves *SSH* e diretoria de ficheiros *log* e ainda, alteração das diretorias dos certificados *SSL*.

```
gitlab_shell['auth_file'] = "/var/opt/gitlab/.ssh/authorized_keys"
gitlab_shell['log_directory'] = "/var/log/gitlab/gitlab-shell/"
```

```
gitlab_shell['log_format'] = 'json'
gitlab_shell['http_settings'] = {
    user: 'username',
    password: 'password',
    ca_file: '/etc/ssl/cert.pem',
    ca_path: '/etc/pki/tls/certs',
    self_signed_cert: false
}
```

4.6 Integração com o *GitLab-Workwhorse*

Como vimos anteriormente, o *GitLab-Workwhorse* funciona como um servidor de *proxy*, pelo que deverá ter acesso ao *backend* da aplicação *GitLab*. O ficheiro de configuração permite alterar vários parâmetros sobre o funcionamento dp *GitLab-Workwhorse*, entre os quais: alterar o endereço do servidor *backend*; limitar o número de pedidos recebidos e definir um *timeout* para esses; etc. Em baixo apresentamos um exemplo de configuração.

```
gitlab_workhorse['auth_backend'] = "http://localhost:8080"
gitlab_workhorse['api_queue_limit'] = 0
gitlab_workhorse['api_queue_duration'] = "30s"
```

4.7 Integração com o *Unicorn*

O *GitLab* disponibiliza um conjuntos de parâmetros que permitem alterar os valores *default* do *Unicorn*. Entre estes, destacamos os seguintes:

- ***listen address***: é possível definirmos a interface por onde estão a ser enviadas as tarefas;
- ***listen port***: tamem é possível definirmos o *socket unix*;
- ***timeout***: tempo máximo permitido para a execução de uma tarefa; geralmente varia entre os 15 e 30 segunos para que possa ser possível outras tarefas serem executadas;
- ***worker_processes***: número de processos que são lançados; geralmente este número resulta na soma do número de *cores CPU* com 1; especial cuidado para máquinas com pouca memória *RAM* disponível;

```
unicorn['listen'] = 'localhost'
unicorn['port'] = 8080
unicorn['worker_processes'] = 4
unicorn['worker_timeout'] = 30
```

4.8 Integração com o *Sidekiq*

No que concerne à configuração do *Sidekiq*, o *GitLab*, tal como com o *Unicorn*, disponibiliza vários parâmetros de configuração desta ferramenta de gestão de processos do *Ruby*.

Em baixo segue um exemplo simples de configuração.

```
sidekiq['listen_port'] = 8082
sidekiq['listen_address'] = "localhost"

# Timeout máximo (segundos) de execução de tarefas
sidekiq['shutdown_timeout'] = 8

# Número de threads criadas pelo Sidekiq
sidekiq['concurrency'] = 25
```

4.9 Integração com o *Redis*

Caso se pretenta utilizar um instância do *Redis* diferente daquela fornecida pelo *omnibus*, o *GitLab* permite que sejam adicionadas as informações necessárias ao ficheiro de configuração.

Devemos primeiramente desativar o *bundle* fornecido pelo *omnibus*.

```
redis['enable'] = false
```

De seguida devemos indicar a instância *Redis* que queremos integrar na nossa aplicação.

```
# Redis via TCP
gitlab_rails['redis_host'] = "10.0.1.201"
gitlab_rails['redis_port'] = 6379

# Redis via sockets Unix
gitlab_rails['redis_socket'] = '/tmp/redis.sock'
```

Podemos no entanto utilizar o *omnibus* para criar instâncias *Redis*, que podem ou não apresentar diferentes funcionalidades, como: servir como *cache*; armazenamento de filas de espera ou partilha de estado.

```
external_url 'https://gitlab.example.com'

# Ativar unicamente o Redis
redis_master_role['enable'] = true

redis['port'] = 6379
redis['bind'] = '10.0.0.1'

# Se pretender, escolher uma das 3 opções em baixo
gitlab_rails['redis_cache_instance'] = REDIS_CACHE_URL
gitlab_rails['redis_queues_instance'] = REDIS_QUEUES_URL
gitlab_rails['redis_shared_state_instance'] = REDIS_SHARED_STATE_URL
```

4.10 Integração com o *Gitaly*

O *Gitaly* pode ser configurado através de um ficheiro *TOML* próprio. No entanto, podemos configurar alguns parâmetros diretamente no ficheiro de configuração do *GitLab*.

O *Gitaly* pode ser executado num servidor individual. Neste caso é necessário desativar o *bundle default* atribuido e configurado pelo *omnibus* atribui um *token*. É ainda necessário definir um endereço e uma porta para poderem ser efetuados pedidos ao *Gitaly*.

```
gitaly['enable'] = false
gitlab_rails['gitaly_token'] = 'abc123secret'
gitaly['listen_addr'] = "10.0.2.10:8075"
```

Podem ainda ser definidos os caminhos para os repositórios *git*.

```
gitaly['storage'] = [
  { 'name' => 'default', 'path' => '/mnt/gitlab/default/repositories' },
  { 'name' => 'storage1', 'path' => '/mnt/gitlab/storage1/repositories' },
]
```

5 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*. Em baixo mostramos os elementos considerados críticos.

Componente	**Motivo**	**Solução**
HAProxy	Utilizado para balancear tráfego entre os diferentes componentes como o PostgreSQL ou pedidos como o Redis, originando um ponto de falha central	Uma das soluções passa por usar o *Keepalived* para possibilitar a partilha do endereço IP por diversas instâncias de servidores diferentes.
Keepalived	Responsável por manter sempre duas instâncias de HAProxy ativas.	Quantas mais instâncias do servidor de HAProxy estiverem ativas maior será a disponibilidade.
Redis Sentinel	Ferramentas de automação que atualizam os ficheiros de configuração poderão causar erros ou até mesmo a falha do sistema.	É importante que caso sejam usadas ferramentas de automação para criar os ficheiros iniciais, estas não atualizem constantemente o ficheiro, uma vez que por defeito a maioria das ferramentas tenta manter o ficheiro consistente e o *Redis Sentinel* necessita que alterações vão ocorrendo.
Sistema de Ficheiros Distribuído	Numa aplicação como o git, se todos os restantes componentes estiverem disponíveis e não tiverem acesso ao sistema de ficheiros, a aplicação está basicamente inutilizável.	GPFS é um sistema de ficheiros paralelo responsável por assegurar redundância e tempos de leitura eficientes da base de dados.

5.1 Disponibilidade e escalabilidade da aplicação *GitLab*

Como já foi referido anteriormente, aplicações como o *GitLab* requerem uma alta disponibilidade, ou seja, não podem existir falhas que comprometam a atividade do sistema, quer seja pelos custos envolvidos quer seja pela confiança dos utilizadores.

Na estruturação de uma arquitetura que sustente o *GitLab*, é necessário ter em conta que o sistema pode e vai falhar, pelo que devem ser tomadas medidas que contornem esta situação.

Dependendo da tolerância máxima a falhas, a alta disponibilidade está diretamente relacionada com soluções mais complexas que irão necessitar de mais trabalho no processo de instalação e manutenção.

Nas seguintes secções, iremos abordar três arquiteturas que garantem escalabilidade e disponibilidade da aplicação *GitLab*. Imagens descritivas das arquiteturas encontram-se nos anexos.

5.1.1 Arquitetura horizontal

É uma solução mais simples que requer menos servidores e é indicada para a maioria dos utilizadores. No entanto, quando o número de clientes aumenta, podem ocorrer situações que provocam um congestionamento da aplicação, como: clonagem de repositórios com grande quantiade de ficheiros; uso eleveado da *API* do *GitLab*; lista de tarefas do *Sidekiq* cheia; etc.

Um exemplo desta arquitetura seria:

- 2 nós a servir o *PostgreSQL*;

- 2 nós a servir o *Redis*;
- 3 nós a servir o *Consul/ETCD/Zookeeper* e *Redis sentinel*;
- 1 nó a servir o *Gitaly* e *NFS*;
- 2 ou mais nós a correr a aplicação *GitLab* com componentes distribuídos entre si (*Unicorn*, *Workhorse*, *Sideqik*).

5.1.2 Arquitetura híbrida

Um dos problemas da arquitetura anterior consistia na possível sobrecarga de determinados componentes. Um arquitetura híbrida permite que determinados componentes sejam executados em nós computacionais dedicados.

É uma ótima arquitetura a ser implementada quando se prevê casos de elevada carga de trabalho e congestão.

Uma arquitetura híbrida é muito semelhante à anterior. Basta adicionarmos 2 novos nós computacionais responsáveis por servir o *Sidekiq*, pelo que têm de ser desativados na aplicação *GitLab*. Idealmente dever-se-á adicionar um novo nó a servir o *Gitaly* e *NFS*.

5.1.3 Arquitetura completamente distribuída

Esta arquitetura pode escalar para centenas de milhares de utilizadores e projetos. É atualmente uma arquitetura muito semelhante à utilizada pelo *GitLab* que estamos habituados a utilizar. É uma estrutura com um número elevado de nós computacionais que requerem elevada gestão e monitorização.

Um exemplo desta arquitetura seria:

- 2 nós a servir o *PostgreSQL*;
- 4 ou mais nós a servir o *Redis*;
- 3 nós a servir o *Consul/ETCD/Zookeeper*;
- 3 nós a servir *Redis sentinel*;
- 2 ou mais nós a servir o *Gitaly* e *NFS*;
- vários nós a servir o *Sidekiq*;
- 2 ou mais nós a servir o *git* (por *ssh* e/ou *http*);
- 2 ou mais nós a servir a *API* do *GitLab*;
- 2 ou mais nós a servir a aplicação *web* do *GitLab*.

6 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.

7 Referências

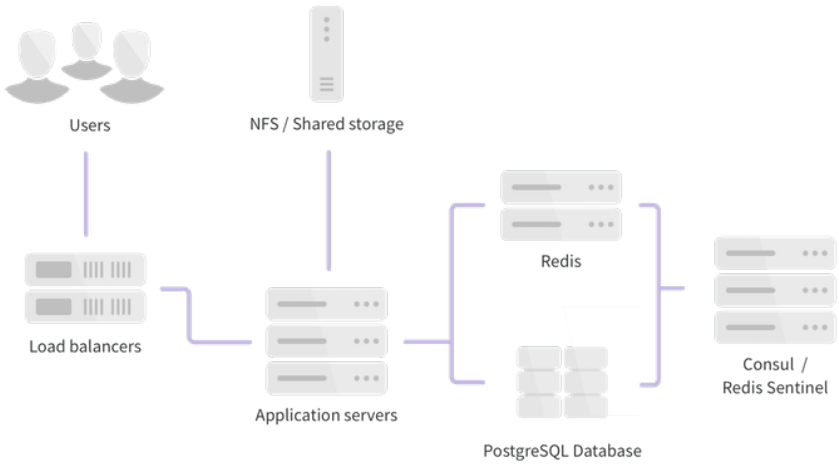
Bertsche, Ryan. IBM Corp. (2017). GitLab: Highly Available Architecture.

GitLab Architecture Overview, GitLab Documentation. Acedido em 9 de Novembro de 2018, em `https://goo.gl/thmr2N` e `https://goo.gl/iyR4zN`

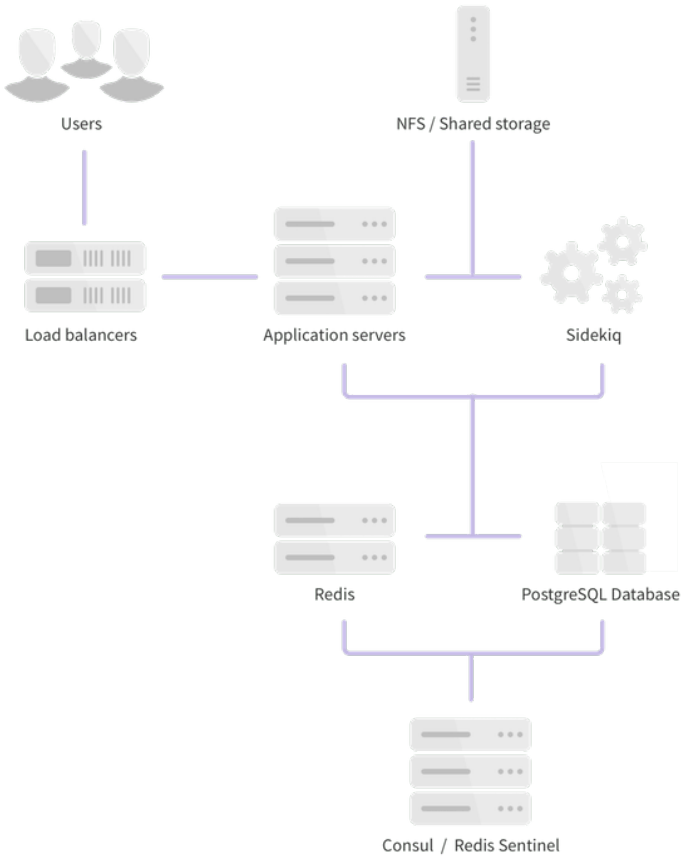
GitLab High Availability, GitLab Documentation. Acedido em 9 de Novembro de 2018, em `https://goo.gl/Dt93fF`

8 Anexos

8.1 Arquitetura horizontal



8.2 Arquitetura híbrida



8.3 Arquitetura completamente distribuída