Alta disponibilidade no OpenNebula

Gabriel Richter[1]

[1]Pós graduação em Cloud Computing – Faculdade Três de Maio (SETREM)
Caixa Postal 98.910-000 - Três de Maio - RS – Brasil

gabrielrih@gmail.com

Abstract. *This project consists in evaluating the functionalities and resources that are available in OpenNebula for the High Availability of its services and its web interface. Based on this research, an architecture must be suggested and tested, describing down the implementation details, advantages and downsides.*

Resumo. *O estudo propõe avaliar as funcionalidades e recursos que o OpenNebula disponibiliza para Alta Disponibilidade dos seus serviços assim como o da interface web. A partir deste estudo, uma proposta de arquitetura deve ser sugerida, implementada e apresentada. Dando ênfase para as suas características de implementação, vantagens e ao final expondo suas limitações e pontos de melhoria.*

1. Introdução

O OpenNebula é um software *open source* e de fácil uso, criado para permitir que você construa e gerencie uma Nuvem Privada. Esta solução combina virtualização, multi inquilinos, provisionamento automatizado e elasticidade. Tudo isso possibilitando a criação de uma Nuvem Privada, híbrida ou mesmo um ambiente de *Edge Computing*.

O objetivo central deste estudo foi o de avaliar e implementar uma solução que possibilite a Alta Disponibilidade dos serviços do OpenNebula. O foco principal aqui é propor o uso de recursos do OpenNebula de modo a possibilitar a redução de *downtimes* em um ambiente real. Desde *downtimes* programados, como são os casos de manutenção para aplicação de *patch* de segurança por exemplo, como também *downtimes* não programados. Também buscamos uma alternativa que possibilite tolerância a falhas de forma automatizada, assim, mesmo em situação de problemas no servidor, o sistema deveria continuar funcionando.

2. Alta disponibilidade

Apesar de um *deploy* simplificado do OpenNebula poder ser suficiente para o seu ambiente, é comum empresas de médio ou mesmo grande porte necessitar de uma maior criticidade na Nuvem provisionada, de forma que esta tenha que ter um *downtime* mínimo. É aqui que entram os recursos de Alta Disponibilidade do OpenNebula.

De forma geral, a ferramenta oferece opções de tolerância a falha e recuperação automática para *Virtual Machines* e *hosts físicos* (que são utilizados para hospedar as VMs). Além de oferecer recursos para que o próprio *frontend* e serviços do OpenNebula executem de forma redundante com poucos segundos de *downtime* em caso de falha.

Para recuperação de VMs e *hosts,* a ferramenta é bastante limitada. De forma simplificada, o que ela oferece são *hooks* que podem executar *scripts* a partir de uma determinada *trigger.* Exemplo, quando um *host* físico mudar o seu estado para erro*,* um *script* será disparado. Este *script* pode executar qualquer coisa, desde enviar uma notificação ao gerenciador da Nuvem até algo mais automatizado como realizar a migração de todas as VMs deste *host* para outro.

Entretanto, a parte que nos interessa neste estudo são os recursos de Alta Disponibilidade oferecidos ao *frontend.* Importante deixarmos claro que o Sunstone é a interface *web* que vamos utilizar com o OpenNebula, contudo, quando a documentação do OpenNebula faz referência ao *frontend* ela está se referindo aos serviços principais do OpenNebula e não à interface *web*. Para evitar confusão, utilizaremos a mesma nomenclatura da documentação oficial.

2.1. Alta disponibilidade no frontend

A alta disponibilidade do *frontend* do OpenNebula é atingida através do *deploy* da solução em vários servidores e por meio de um protocolo distribuído de consenso para fornecer mecanismo de tolerância à falha e um estado consistente dos dados. Na prática, ao invés de ter um único servidor com os serviços do OpenNebula executando, teremos 3, 5, 7 ou mais (conforme necessidade).

O consenso é um problema bastante comum em um sistema distribuído tolerante à falha. Ele envolve basicamente vários servidores concordarem com uma decisão. Tipicamente, o algoritmo de consenso faz progresso quando a maioria dos servidores envolvidos está disponível. Se mais do que a maioria dos servidores falhar, o algoritmo para de progredir, porém mantém um retorno consistente dos dados.

No caso do OpenNebula, o algoritmo de consenso utilizado é o *Raft*. Em um cluster do OpenNebula você terá um único servidor chamado de *leader* e os demais serão *followers*. Na prática, os *followers* podem receber operações de leitura, porém somente o *leader* pode realizar escrita. O *leader* se mantém como servidor "principal" através do envio de *heartbeats* periódicos aos *followers.* A partir do momento que os *followers* não receberem mais *heartsbeats* uma nova eleição é disparada e um *follower* é eleito o novo *leader.*

Sempre que uma modificação for realizada no ambiente, o *leader* atualiza o *log* e replica esta entrada para a maioria dos *followers* antes mesmo de efetivamente escrever a alteração na base de dados. Obviamente a latência da escrita aumenta, entretanto o estado consistente do sistema é mantido e o *cluster* pode seguir operando mesmo quando um ou mais servidores falharem (dependendo da quantidade total de nós).

Outro aspecto extremamente importante para esta arquitetura é a necessidade dos *datastores* do OpenNebula serem compartilhados entre todos os nós, ou seja, todos os servidores do OpenNebula devem ter acesso aos mesmos *datastores.* Isso é fundamental pois vários arquivos importantes são salvos nos *datastores,* e toda vez que o *leader* do *cluster* mudar, este novo *leader* deve ter acesso aos arquivos que foram gerados anteriormente por outro servidor. Um exemplo disso é o *datastore* do tipo

image que conterá as imagens de VMs baixadas. Outro exemplo, é o *datastore* do tipo *system,* o qual grava, entre outras coisas, arquivos com metadados das VMs que estão em estado *stopped* ou *undeployed.*

Aqui temos um resumo dos requisitos necessários para realizar o *deploy* do OpenNebula utilizando o recurso de Alta Disponibilidade:

- Temos que ter um número ímpar de servidores: devido a própria arquitetura e a forma que o algoritmo de consenso *Raft* funciona.
- Recomendamos que as configurações, bibliotecas, versão de sistema operacional e todo o resto seja exatamente igual entre cada um dos servidores.
- Todos os servidores devem possuir as mesmas credenciais (chaves do OpenNebula).
- Cada servidor deve ter uma instância de database instalada.
- Devemos implementar um *datastore* compartilhado entre todos os servidores.

2.2. Alta disponibilidade no Sunstone

A Alta Disponibilidade com a interface *web* Sunstone pode ser facilmente atingida adicionando um Balanceador de Carga na frente do ambiente. Desta forma, eu posso ter vários servidores com o Sunstone instalado e o Balanceador de Carga fará o controle de qual servidor deve receber cada requisição.

Neste estudo estamos utilizando a versão *open source* do *NGinx.* O *NGinx* possibilita configurar regras de *health check* para os servidores. Assim, caso o serviço do Sunstone não esteja executando corretamente em algum servidor, automaticamente o *NGinx* remove temporariamente este servidor e para de enviar requisições para ele (mantendo o sistema *web* funcionando e garantindo *downtime* mínimo ao usuário).

Ainda, além de garantir Alta Disponibilidade na interface *web*, já que a interface *web* seguirá funcionando enquanto um único servidor estiver com o serviço do Sunstone operando, ainda temos o benefício da distribuição de carga. Estamos utilizando o método *least connections,* o que significa que o *NGinx* sempre enviará a requisição para o servidor que tiver menos conexões. Isso garante uma paridade de carga entre todos os servidores.

Também é necessário o uso do *Memcached* para o controle das sessões de usuários. O *Memcached* nada mais é que um sistema *open source* de *storage* para dados do tipo chave/valor e que trabalha em memória (semelhante ao *Redis*). Ao configurar o Sunstone para utilizar o *Memcached,* todos os dados de sessão de usuários serão salvos no *Memcached* e não na memória do servidor. Isso evita problemas de *login* e queda inesperada de sessão.

3. Implantação

Baseado nos recursos de alta disponibilidade citados anteriormente, a arquitetura proposta e implementada para este estudo consiste em:

- Três servidores que chamaremos de *frontends:* possuem o mesmo sistema operacional e as mesmas configurações de softwares. Cada um dos servidores tem o OpenNebula, o Sunstone e o MySql instalados.
- Um servidor que chamaremos de *balancer:* que possui o *NGinx Open Source* instalado.
- Outro servidor que chamaremos de *cache:* que possui o *Memcached* e o *NFS server* instalados.

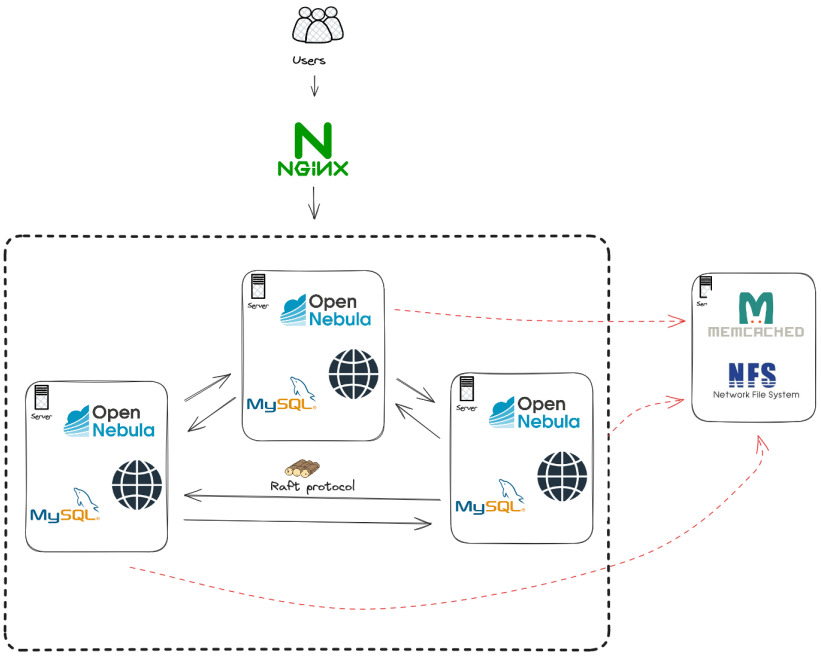

Figura 1. Arquitetura de alta disponibilidade

Esta arquitetura fornece os seguintes benefícios que serão detalhados em seguida:

- Tolerância a falha para os processos do OpenNebula;
- Tolerância a falha para a interface *web*;
- Distribuição de carga entre as requisições HTTP;
- Manutenção de servidores sem *downtime*;
- Escalabilidade horizontal para o Sunstone;

3.1. Tolerância a falhas para os processos do OpenNebula

Estamos utilizando três servidores para o *frontend.* Somente um deles será o *leader,* os demais serão *followers.* No OpenNebula, as operações de leitura podem ser realizadas em qualquer *oned*, mesmo nos *followers*. Já as operações de escrita, mesmo apontadas para os *followers*, são redirecionadas para o *leader*. Na prática, isso possibilita que qualquer operação possa ser executada em qualquer nó.

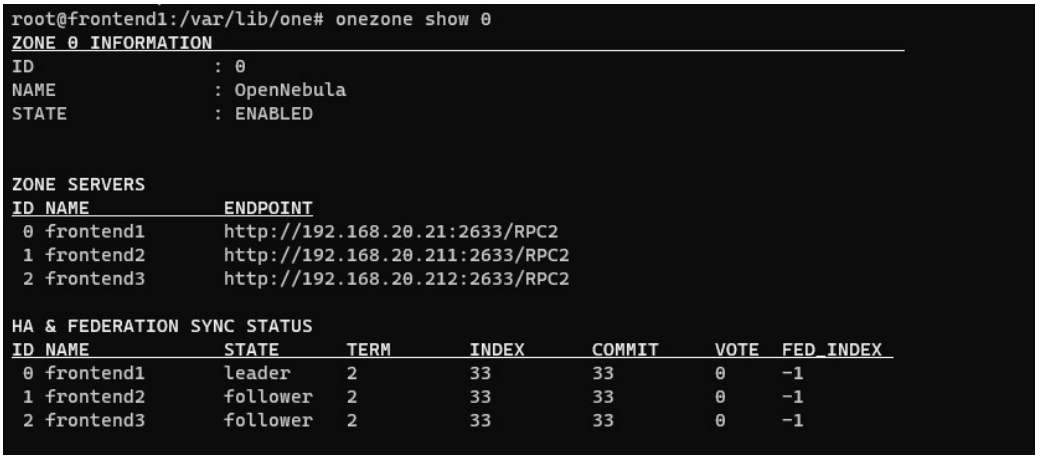

Figura 2. Nós do cluster OpenNebula

Quanto à disponibilidade, teremos sempre que ter a maioria dos nós funcionando. Como estamos trabalhando com três nós, a maioria de três é dois. Portanto, somente um nó pode ficar indisponível por vez. Caso dois nós fiquem indisponíveis, o sistema para de responder imediatamente. Importante comentar também que três nós é a arquitetura mínima para a configuração de Alta Disponibilidade do OpenNebula, porém, nada impede de adicionarmos novos nós se for necessário (lembrando sempre de trabalhar com números ímpares).

O processo de tolerância à falha é operado automaticamente pelo próprio OpenNebula via algoritmo *Raft.* Caso ocorra algum problema em um servidor do tipo *follower, o* serviço pare de funcionar ou mesmo o servidor desligue, o sistema segue em funcionamento sem nenhum problema. Agora, caso ocorra o mesmo problema em um servidor que seja o *leader,* o OpenNebula irá transformar os dois nós *followers* em *candidates,* ou seja, candidatos a se tornarem um novo *leader,* e uma eleição é disparada. Assim que tivermos um novo *leader* o sistema volta a operar. Note que este processo não deve levar mais do que poucos segundos, assim, mesmo com falha, o *downtime* do ambiente provavelmente será imperceptível ao usuário final.

Para que o novo *leader* eleito possa performar operação de escrita no sistema, precisamos ter todos os *datastores* do OpenNebula em um servidor centralizado. É aqui que entra o servidor de NFS mostrado na imagem da arquitetura proposta. Este servidor, chamado de *cache,* é utilizado para poder mapear estes *datastores* em um único local, utilizando o protocolo NFS e mapeando esses diretórios via rede em todos os *frontends*. Assim, todos os *frontends* passam a ter acesso aos mesmos arquivos, possibilitando que independentemente de quem é o *leader,* este terá acesso aos arquivos e configurações geradas anteriormente por outro servidor.

3.2. Tolerância a falhas para a interface web

O que nos garante tolerância à falha automatizada da interface *web* é o sistema de *health check* do próprio *NGinx.* No *NGinx* nós temos os três servidores de *frontends* configurados e temos também uma configuração de *health check* passiva. Esta configuração é atingida através de dois parâmetros que indicam a quantidade máxima de

tentativas de erros e o tempo que o *NGinx* espera por uma resposta do servidor. Caso a quantidade de tentativas seja atingida, o servidor é marcado como indisponível por um determinado tempo antes que o *NGinx* tente enviar uma nova requisição ao servidor.

```
...

upstream backend {
    least_conn;
    server frontend1:9869 max_fails=3 fail_timeout=30s;
    server frontend2:9869 max_fails=3 fail_timeout=30s;
    server frontend3:9869 max_fails=3 fail_timeout=30s;
}

server {
    ...
    location / {
        proxy_pass http://backend;
        proxy_set_header X-Real-IP $remote_addr;
    }
}
```

Figura 3. Configuração de load balancing no NGinx

É importante termos claro aqui que esta não é a solução ideal, porém é a solução que a versão *open source* do *NGinx* nos oferece. Note que caso o serviço do *Sunstone* pare de responder em um dos servidores, o *NGinx* igual vai enviar uma requisição para este servidor e somente entenderá que o serviço não está funcionando após algumas tentativas de erro. Isso na prática representa uma mensagem de *timeout* ou erro ao usuário. A vantagem é que apesar da pequena indisponibilidade, o sistema se recupera automaticamente após alguns segundos. Assim, não é necessária nenhuma intervenção manual.

Ainda, mesmo utilizando a versão *open souce,* o tempo de *downtime* pode ser reduzido personalizando as configurações do *health check* passivo conforme a sua necessidade. Você pode diminuir o tempo de *timeout* ou mesmo reduzir a quantidade de tentativas caso entenda que esta configuração se encaixa melhor no seu cenário.

3.3. Distribuição de carga entre as requisições HTTP

Outra vantagem que o NGinx nos oferece é a distribuição de carga. Diferentemente dos serviços do OpenNebula em que há um único *leader* e somente este *leader* pode operar escritas, quando falamos da interface *web* com um balanceador de carga, todos os servidores podem receber requisições HTTP.

Neste estudo, configuramos o *NGinx* para utilizar o método *least connection* de distribuição de carga. Isso significa que sempre que uma requisição HTTP chega ao *NGinx,* este verifica qual é o servidor com menos conexões e redireciona a requisição para ele. Na prática temos uma distribuição uniforme de carga e quanto mais servidores tivermos configurados, maior carga o sistema suportará.

O *Memcached* é um elemento obrigatório para o correto funcionamento desta estrutura. Isso porque por padrão os dados de sessão de usuário são salvos na memória RAM do servidor. Nesse caso, como estamos trabalhando com vários servidores, cada requisição de um mesmo usuário pode cair para um servidor diferente, assim, os dados de sessão de usuário devem estar salvos em um local centralizado para que todos os *frontends* tenham acesso.

3.4. Manutenção de servidores sem downtime

Conforme já comentado, para o serviços do *frontend,* podemos ter somente um nó indisponível por vez. Já para o serviço do *Sunstone,* podemos ter até dois nós indisponíveis. Dito isso, fica evidente a facilidade que isso nos traz para a manutenção dos servidores.

Sempre que for necessário aplicar algum *patch* de segurança, atualizar alguma versão de *software* ou até mesmo realizar algum ajuste de *hardware,* nós podemos fazer isso sem *downtime* na aplicação. Basta realizar o processo um servidor por vez.

Um plano de execução para esta situação poderia ser o seguinte:

- Remover o servidor que será aplicada a manutenção do balanceador de carga: Assim nenhuma requisição HTTP será direcionada para ele.
- Parar todos os serviços do OpenNebula: Assim, se o servidor é um *follower* nada acontecerá, porém se ele for o *leader,* um dos *followers* assumirá o seu lugar.
- Realizar as manutenções previstas no servidor.
- Iniciar novamente os serviços.
- Validar os serviços: servidor voltou a ser *follower* no cluster? O *Sunstone* está respondendo?
- Habilitar o servidor novamente no balanceador de carga.

3.5. Escalabilidade horizontal para o Sunstone

No cenário proposto para o Sunstone, quanto mais servidores tivermos, mais carga o sistema suportará. Isso nos dá uma escalabilidade horizontal que reduz muito o custo de *hardware* necessário e permite também uma elasticidade simplificada e sem *downtime*.

Isso é possível pois toda vez que for necessário mais recursos para o Sunstone, basta adicionar um novo servidor e incluí-lo no balanceador de carga. O mais vale quando o fluxo diminuir, podemos remover o servidor do balanceador de carga e depois eliminar o servidor. Claro que nesta arquitetura isso teria que ser feito manualmente, porém temos espaço para evolução da proposta.

4. Limitações e melhorias

Uma limitação do ambiente é a escalabilidade dos serviços do OpenNebula. Note que embora tenhamos três nós na arquitetura, os nós auxiliarem servem somente como redundância. Caso eu precise de mais recursos computacionais, a escalabilidade

da estrutura segue sendo vertical e se necessário teríamos que aumentar igualmente os recursos em todos os nós (gerando também mais custo).

Quanto à interface *web*, embora o serviço do Sunstone tenha sido configurado nos mesmos servidores do OpenNebula, ele é um *software* independente e pode ser instalado em servidores dedicados. Além disso, o fato do Sunstone estar no mesmo servidor do OpenNebula cria uma dependência na quantidade de servidores que ambos serviços precisam.

O que quero dizer com isso é que em uma arquitetura mais completa, poderíamos instalar o Sunstone em servidores dedicados. Isso possibilitaria, por exemplo, que a quantidade de servidores para o Sunstone fosse uma, enquanto que a quantidade de servidores para o OpenNebula fosse outra. Exemplo, posso ter somente dois servidores para o Sunstone porém três servidores para o OpenNebula. Assim, a escalabilidade do Sunstone não estaria atrelada à quantidade de servidores que eu tenho para o OpenNebula e vice-versa. Além disso, a configuração de recursos computacionais pode ser personalizada conforme a necessidade de cada serviço.

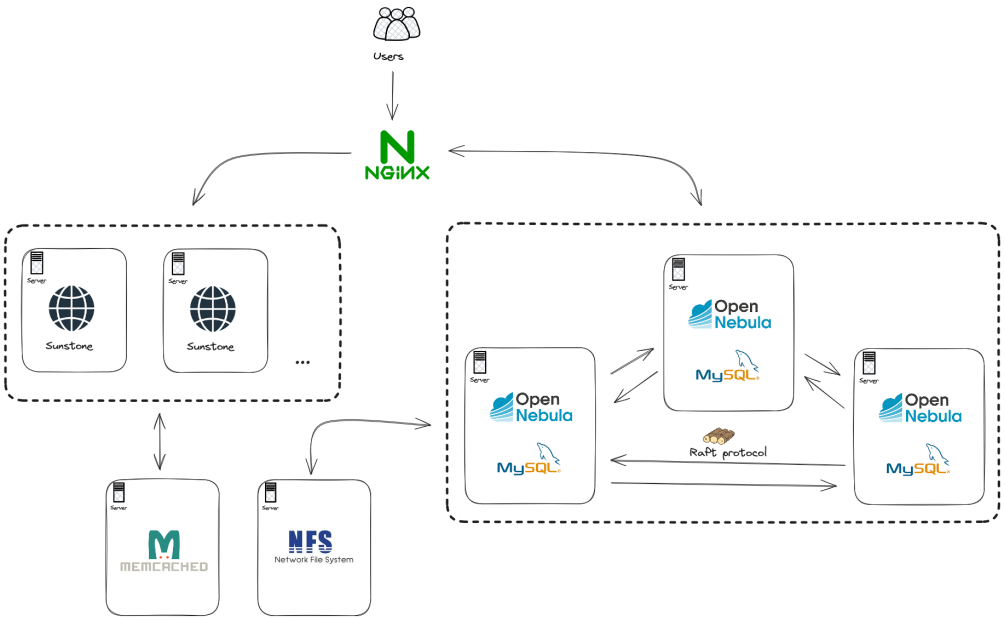

Figura 4. Arquitetura de alta disponibilidade isolada

Outra possibilidade para esta mesma sugestão de arquitetura, é a de separar também o *Memcached* do *NFS server.* As razões são as mesmas, posso ter configurações de *hardware* e *software* diferentes em cada serviço e caso um dos servidores fique indisponível, eu estaria afetando somente um serviço e não ambos, assim, melhorando ainda mais a disponibilidade do sistema como um todo.

Quanto à redução de erros e *timeout* ao usuário, poderíamos optar por utilizar o *NGinx Plus,* que é pago, ao invés do *NGinx Open Source.* A razão para tal mudança dependeria exclusivamente do nível de SLA que a empresa quer atingir para a disponibilidade da interface *web.* Falamos anteriormente sobre o *health check* passivo do

NGinx, que embora ajude bastante, pode mesmo assim apresentar algum erro ou *timeout* ao usuário.

Contudo, a versão *Plus* nos oferece a função de *health check* ativo. Esta opção não requer requisição de usuário para validar se um nó está ou não disponível. Ela basicamente envia uma requisição a todos os nós configurados a cada poucos segundos e marca como indisponíveis os nós que não responderam. O *NGinx* não enviará requisições aos nós indisponíveis até que estes voltem a passar no *health check.* Note que esta opção pode reduzir bastante a ocorrência de problemas de erro ou *timeout* ao usuário.

E por fim, mas não menos importante, mesmo na arquitetura proposta na figura acima, temos claramente vários pontos únicos de falha. O *NGinx* é extremamente importante nesta arquitetura e não possui redundância, portanto se tiver algum problema com ele, todas as outras questões de Alta Disponibilidade implementadas serão irrelevantes, pois o sistema igual ficará indisponível. Nesse caso, a versão *Plus* do *NGinx* oferece a opção de alta disponibilidade através do modelo ativo/passivo, tendo um *NGinx* respondendo às requisições enquanto um outro, de *backup,* assume caso o principal pare de responder.

O mesmo ocorre com o servidor do *Memcached* e com o servidor *NFS.* Ambos geram problemas ou mesmo indisponibilidade no ambiente caso parem de funcionar, portanto, para termos uma arquitetura totalmente redundante, precisaríamos também de recursos e ferramentas para duplicar as suas estruturas. Como o objetivo do estudo era a redundância dos servidores do OpenNebula em si, a análise de Alta Disponibilidade para estes dois recursos não foi analisada.

5. Conclusão

Após análise e testes de operação de toda a estrutura, ficou evidente que ela cumpriu com o objetivo. Conseguimos propor uma arquitetura para o OpenNebula que possibilita ter até um nó indisponível sem *downtime*. Também é interessante comentar que esta arquitetura pode ser facilmente expandida adicionando mais nós à ela, assim, poderíamos expandir para cinco nós, por exemplo, o que nos permitiria ter até dois nós indisponíveis por vez (o que certamente reduziria ainda mais a possibilidade de *downtime* em um ambiente real).

Embora a redução da possibilidade de *downtime* fosse o objetivo principal do estudo, com a arquitetura proposta conseguimos também aumentar a capacidade de carga da interface *web,* possibilitando escalabilidade horizontal, e ainda conseguimos homologar o sistema de tolerância a falhas que o próprio algoritmo *Raft* nos possibilita. Comprovando que a solução do OpenNebula para Alta Disponibilidade é totalmente eficaz para a recuperação rápida de falhas inesperadas.

De forma geral, o que vai definir qual arquitetura e recursos utilizar dependerá sempre da sua realidade. Quanto de SLA preciso garantir na minha Nuvem Privada? Quanto de dinheiro pretende-se gastar com este projeto? Qual é o custo de manutenção e complexidade do ambiente? Este custo faz sentido para o que quero entregar? No fim,

estas perguntas devem ser respondidas para definir se faz ou não sentido para o seu negócio utilizar ou não uma arquitetura de Alta Disponibilidade.

References

Dormando (2018) “What is Memcached?”, https://memcached.org/.

GitHub.io (2023) “The Raft Consensus Algorithm”, https://raft.github.io/.

IBM Documentation Help (2020) “Network File System”, https://www.ibm.com/docs/en/aix/7.1?topic=management-network-file-system.

NGINX (2023) “High Availability”, https://www.nginx.com/products/nginx/high-availability/.

NGINX Docs (2023) “HTTP Health Check”, https://docs.nginx.com/nginx/admin-guide/load-balancer/http-health-check.

NGINX Docs (2023) “HTTP Load Balancing”, https://docs.nginx.com/nginx/admin-guide/load-balancer/http-load-balancer/.

NGINX Docs (2023) “NGINX SSL Termination”, https://docs.nginx.com/nginx/admin-guide/security-controls/terminating-ssl-http/.

OpenNebula.io (2023) “Local Storage Datastore”, https://docs.opennebula.io/6.8/open_cluster_deployment/storage_setup/local_ds.html .

OpenNebula.io (2023) “OpenNebula Front-end HA”, https://docs.opennebula.io/6.6/installation_and_configuration/ha/frontend_ha.html.

OpenNebula.io (2023) “OpenNebula overview”,

https://docs.opennebula.io/6.8/overview/opennebula_concepts/opennebula_overview.html.

OpenNebula.io (2023) “Sunstone for Large Deployments”, https://docs.opennebula.io/6.6/installation_and_configuration/large-scale_deployment/sunstone_for_large_deployments.html.

OpenNebula.io (2023) “Virtual Machine High Availability”, https://docs.opennebula.io/6.6/installation_and_configuration/ha/vm_ha.html.