FaultSee: Avaliação Reproduzível de Sistemas Distribuídos Sujeitos a Faltas

Miguel Amaral[1], Miguel L. Pardal[1], and Miguel Matos[1]

INESC-ID, Instituto Superior Técnico, Universidade de Lisboa, Portugal
{miguel.p.amaral,miguel.pardal,miguel.marques.matos}@tecnico.ulisboa.pt

Resumo Os sistema distribuídos são cada vez mais importantes na sociedade moderna operando, muitas vezes, a uma escala global e com requisitos de disponibilidade muito perto dos 100%. Estes requisitos tornam os sistemas distribuídos, que são tipicamente compostos por vários componentes, bastante difíceis de avaliar e testar de uma forma sistemática e reproduzível. Por exemplo, ao analisar um estudo ou artigo sobre um sistema real é frequente ver afirmações sobre a gestão de faltas em nós do sistema "o nó X foi terminado", que não são explicadas nem contextualizadas de forma a permitir a reprodução num ambiente de teste alternativo. Dado que o comportamento do sistema pode variar substancialmente consoante o tipo de 'falha' injetada (por exemplo é diferente matar o processo versus matar o sistema operativo ou máquina virtual) torna-se praticamente impossível a um investigador ou engenheiro reproduzir o comportamento observado nesse teste. Neste artigo propomos uma linguagem de especificação de sistemas distribuídos e injeção de faltas que capturam precisamente variáveis como o ambiente de teste, a carga de trabalho e o tipo de faltas. A linguagem é materializada na plataforma *FaultSee* que permite avaliar sistemas reais de uma forma mais sistemática e reproduzível do que o estado da arte. Estas funcionalidades são demonstradas num cenário realista usando a base de dados *Apache Cassandra* como caso de estudo.

Keywords: Sistemas distribuídos · Avaliação de sistemas· Reprodutibilidade.

1 Introdução

Na sociedade moderna a tecnologia, e em particular o *software*, tem um papel cada vez mais importante: é utilizado na bolsa e cripto-moedas, gere cadeias de fornecimento que distribuem alimentos e medicamentos, ajuda os operadores de serviços de urgência a controlar operações em curso. Os sistemas desenvolvidos são cada vez mais críticos.

Nos últimos anos, devido aos baixos custos de entrada e facilidade de utilização do serviço, tem havido uma migração dos serviços para a *cloud*. Como neste novo paradigma acrescentar um novo servidor é extremamente simples, a escala dos serviços tem vindo a aumentar. Com um número muito maior de servidores, a probabilidade de um deles estar em falta é muito grande, levando a que a falha

de componentes seja agora considerada a norma, em vez de excepção. Assim sendo é fundamental poder avaliar o comportamento dos sistemas em cenários em que são sujeitos a faltas. Esta avaliação tem que ser possível de fazer de forma sistemática e reproduzível, não só para poder garantir que erros detectados em versões anteriores já foram corrigidos, como também permitir que as descobertas feitas por investigadores sejam facilmente reproduzíveis por outros investigadores, uma vez que a incapacidade de reproduzir os resultados obtidos gera dúvida no seio da comunidade académica, atrasando a aceitação de novos contributos para o estado da arte.

Com a tecnologia actual, através de contentores *Docker* e orquestradores, como *Docker-Swarm* e *Kubernetes*, já é fácil lançar sistemas complexos em larga escala, mas avaliar cenários de falhas continua a ser uma tarefa difícil para os programadores. Existe assim uma boa oportunidade de combinar a flexibilidade da tecnologia actual para lançar sistemas complexos com a necessidade de ter uma plataforma que permita a avaliação perante cenários de falha.

A nossa proposta para resolver o problema é a especificação de uma linguagem, a *FSDL*, para descrever o sistema, o conjunto de faltas a aplicar e o momento no qual devem ocorrer. O *FaultSee* é uma plataforma que processa esta linguagem e permite executar, e monitorizar o sistema, num cenário especificado e recolher os resultados no final. O investigador pode analisar o resultado e partilhar a experiência com a comunidade, para que outros possam reproduzir a experiência nos seus ambientes. Levamos assim o conceito de resultados reproduzíveis [7] mais adiante.

O *FaultSee* permite a automatização de testes, o que leva a que o *software* possa ter um ciclo de desenvolvimento menor uma vez que os investigadores recebem *feedback* mais rapidamente. Isto leva a que existam menos erros no código, visto que que os investigadores conseguem identificar mais erros, mais cedo. A natureza da ferramenta leva ainda a que possam ser desenvolvidos *benchmarks* para diferentes aspectos de sistemas distribuídos, a nível de desempenho, não só em condições normais, como também em condições de diferentes falhas.

Este artigo apresenta a nossa proposta em detalhe, os resultados que foram obtidos numa experiência em concreto e compara com os resultados anteriores.

2 Trabalho Relacionado

Nesta secção discutimos sucintamente o trabalho relacionado com a gestão de contentores e a injeção de faltas. Uma das tecnologias desenvolvidas pela comunidade, e aproveitada pelo *FaultSee* é os contentores *Docker* [4]. O contentor *Docker* é uma tecnologia que cria ambientes de execução com custo inferior ao das máquinas virtuais. Os contentores *Docker* partilham o *kernel* de Linux com o servidor onde são lançados, não sendo assim necessário recriar todo o sistema operativo, o que reduz o seu consumo de recursos e tem menos impacto no desempenho. Estes contentores permitem que os programadores possam recriar o ambiente de produção em desenvolvimento, independente do sistema operativo que usem. O *Docker-Swarm* [1] e o *Kubernetes* [2] são duas ferramentas

que permitem orquestrar automaticamente como são lançados contentores num conjunto de servidores.

O *Cords* [5] é uma ferramenta desenvolvida para injetar faltas em sistemas de ficheiros distribuídos. Os autores demonstraram que redundância não implica tolerância a falhas, através da descoberta de vários *bugs* em oito sistemas de ficheiros distribuídos muito usados. Esta descoberta mostra a necessidade da criação de melhores mecanismos de testes no ciclo de desenvolvimento de aplicações, uma vez que mesmo aplicações com grandes comunidades têm erros não detectados. A desvantagem desta ferramenta é ter sido desenvolvida apenas com sistemas de ficheiros distribuídos em mente, não podendo ser usada para qualquer sistema, nem mesmo através de extensões.

A *Netflix* criou o conjunto *Simian-Army* [9] com o objetivo de motivar os seus engenheiros a construir ferramentas mais resilientes a faltas e seguindo o mote de que todos os servidores vão eventualmente falhar. Estas ferramentas injetam faltas, segundo uma dada probabilidade, nos sistemas de produção em horário de expediente, garantindo que os engenheiros têm que lidar com os problemas quotidianamente. Adicionalmente, motiva também os programadores a garantir que as aplicações construídas estão, de facto, preparadas para lidar com cenários de faltas. Estas ferramentas permitem que alguma falta que tenha sido introduzida na nova versão seja rapidamente detectada, e na melhor altura para os seus engenheiros. O conjunto de ferramentas inclui:

- – *Chaos Monkey* - A ferramenta original, responsável por terminar instâncias virtuais de forma aleatória;
- – *Chaos Gorilla* - Ferramenta que termina, ou cria uma partição de rede, que isola totalmente as instâncias de um centro de dados de todos os outros;
- - *Chaos Kong* - Ferramenta semelhante ao *Chaos Gorilla*, mas referente a todos os centro de dados numa determinada região, podendo simular um desastre natural;
- - *Latency Monkey* - Ferramenta que simula instâncias que estejam com problemas, mas que aparentem estar saudáveis perante os *health checks*.

O *SimianArmy* representa um avanço na área de testes em cenários de falha, no entanto é uma ferramenta construída para garantir que os sistemas de produção funcionam correctamente, e não uma ferramenta que ajude os programadores na fase de testes. No entanto, esta ferramenta é limitada na faltas que permite injetar.

O *FEX* [8] é uma ferramenta construída com o objetivo de correr medidas padrão (*benchmarks*), automatizando todo este processo, desde o momento de lançar a aplicação, correr os testes e por fim construir os gráficos com os resultados. Esta ferramenta utiliza contentores *Docker* para lançar a aplicação. Outros investigadores podem utilizar as mesmas imagens *Docker*, podendo desta forma replicar resultados, uma vez que repetem a experiência em condições semelhantes. Em contraste com o *FaultSee*, esta ferramenta, foca-se no processo de compilar e lançar uma aplicação, e não na execução de um sistema distribuído num dado cenário de falhas.

O *Dfuntest* [6] é uma ferramenta cuja função é automatizar experiências em sistemas distribuídos. Permite aos seus utilizadores correr experiências numa única máquina ou em *clusters* criados para testes. Os testes executam-se num nó centralizado, o que limita a escala dos testes possíveis.

Usa um nó centralizado para correr os testes, o que causa problemas ao escalar.

O *FaultSee*, que propomos é uma ferramenta construída com o objetivo de ajudar os investigadores a testar melhor as suas aplicações, permitindo criar faltas específicas para cada aplicação, sendo facilmente integradas no sistema. Adicionalmente, através do guião e dos contentores *Docker* é possível reproduzir os resultados, uma vez que são utilizados os mesmos ambientes. Finalmente, como o *FaultSee* injeta as faltas nos nós correspondentes, e não num nó central, terá menos problemas com a escalabilidade e tem menos impacto na rede durante a experiência.

3 Arquitetura

Nesta secção descrevemos em detalhe a arquitectura do *FaultSee*. Os dois objetivos principais são melhorar o processo de desenvolvimento de aplicações distribuídas, através da possibilidade de testar falhas segundo um guião e aumentar a reprodutibilidade na avaliação de sistemas distribuídos, ao permitir que experiências sejam repetidas com um esforço reduzido.

Um dos objetivos no desenho da arquitectura do sistema é descentralizar o sistema ao máximo, de modo que as faltas sejam injetadas independentemente do nó que controla a experiência. Assim, a arquitectura do *FaultSee* tem dois componentes principais: o Controlador Principal e o Controlador Secundário. O Controlador Principal configura os servidores para poder lançar experiências, que são especificadas em guiões. Os guiões contêm faltas que devem ser injetadas em momentos precisos, que ficam à responsabilidade dos Controladores Secundários, que estão a correr em cada um dos nós do *cluster*, permitindo assim executar experiências de maior escala. Na Figura 1 podemos observar como estes componentes interagem entre si.

3.1 Guião de Faltas

As experiências no *FaultSee* são especificadas em dois ficheiros em formato *yaml*. Um dos ficheiros, com sintaxe compatível com *docker-compose*, descreve a os componentes do sistema a testar, enquanto o outro ficheiro, o *guião*, descreve os eventos que vão ocorrer durante a experiência. De modo a explicar e motivar a linguagem do guião, usamos como caso de estudo um cenário com a base de dados *Apache Cassandra* perante o teste de desempenho *YCSB* [3], que é uma ferramenta construída para facilmente testar o desempenho de várias bases de dados, com o objetivo de poder comparar os resultados de diferentes bases de dados.

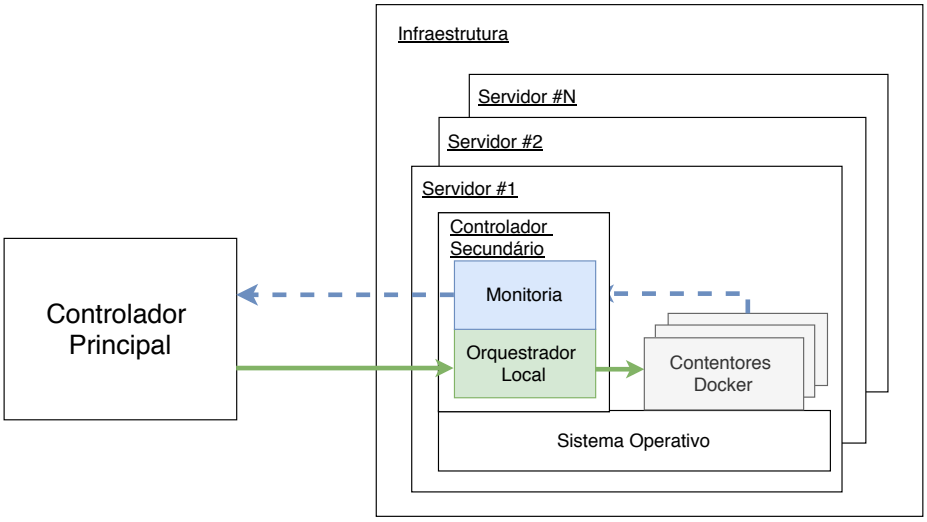

Figura 1. Arquitectura do *FaultSee*.

De seguida, explicamos em detalhe o ficheiro de exemplo da FSDL apresentado na listagem 1.1. Nas linhas 4, 5 e 6 é descrito o estado inicial da experiência, onde se especifica que o serviço *Cassandra* terá dois contentores a correr, enquanto o serviço *YCSB* não terá nenhum. Na última linha está especificado o momento em que a experiência termina, no segundo 1 300. Entre estas duas zonas do ficheiro é especificada a experiência, sendo que existem três momentos. No primeiro, entre a linha 7 e linha 14, está especificado que no segundo 300 da experiência são lançados 2 contentores do serviço *YCSB*. O segundo momento, especificado entre a linha 14 e 23, descreve que no segundo 600 um dos contentores do serviço *Cassandra*, escolhido ao acaso, sofre uma falta em que o CPU é esgotado durante 200 segundos. De notar que os investigadores podem criar novas faltas, criadas especificamente para o contexto da aplicação em teste. Estas faltas são concretizadas através de um executável, como por exemplo *Python* ou *Bash*, que o investigador tem de assegurar que está presente na imagem do contentor. Finalmente, entre as linhas 24 e 31, é especificado que no segundo 900 o sinal *KILL* é enviado ao contentor número um do serviço *Cassandra*, para que este termine de executar.

Listagem 1.1. Exemplo de ficheiro FSDL

```
1 ---
2 churn:
3   synthetic:
4     - start:
5         cassandra: 2
6         ycsb: 0
7     - moment:
```

```
 8           time: 300
 9           services:
10             ycsb:
11               - add:
12                   replicas:
13                     amount: 2
14       - moment:
15           time: 600
16           services:
17             cassandra:
18               - fault:
19                   replicas:
20                     amount: 1
21                   fault_file_name: waste_cpu
22                   fault_script_arguments:
23                     - 200
24       - moment:
25           time: 900
26           services:
27             cassandra:
28               - kill:
29                   replicas:
30                     slots: [1]
31                   signal: KILL
32       - end: 1300
```

3.2 Controlador Primário

O Controlador Primário é o componente principal do *FaultSee* e é o ponto de entrada no sistema para o utilizador. Antes de ser iniciada qualquer experiência este componente faz a inicialização de todos os nós do *cluster*, ou seja, o conjunto de máquinas onde a experiência irá correr. Isto implica os seguintes passos: criação do *cluster* de *Docker-Swarm*, sendo que o nó onde o componente principal corre o nó líder, seguidamente para cada nó do *cluster* este módulo copia as faltas que podem ser injetadas e arranca um contentor com o Controlador Secundário. Desta forma o sistema passa a estar inicializado, e passa a ser possível correr experiências.

Cada experiência pode ser corrida uma ou mais vezes, onde cada vez é denominada de uma *ronda*. Cada experiência é descrita por um ficheiro que especifica os serviços existentes, com sintaxe compatível com *docker-compose*, com o objetivo de diminuir o esforço necessário para usar o *FaultSee*. Adicionalmente, um segundo ficheiro, o guião, descrito em maior detalhe na Secção 3.1, especifica os eventos da experiência. No limite, este ficheiro especifica apenas o estado inicial da experiência e o instante no qual a experiência deverá acabar. Alternativamente, o guião pode especificar um conjunto de eventos. Os eventos podem

aumentar, ou diminuir, o número de contentores dos serviços que estão a correr, lançar *scripts* no interior de contentores e enviar sinais, desta forma é possível injetar faltas.

No início de cada experiência, este componente envia a todos os controladores secundários o guião. Desta forma o controlo da injeção das faltas é descentralizado. No entanto os eventos que pretendam aumentar, ou diminuir, o número de contentores a correr têm de ser processados pelo controlador principal, uma vez que só o nó líder do *cluster* de *Docker-Swarm* tem permissão para executar este tipo de acções.

No início de cada ronda da experiência este módulo é responsável por sincronizar o relógio de todos os nós do *cluster*. Adicionalmente, em conjunto com os controladores secundários, este módulo define o momento em a ronda da experiência começa, isto é feito para garantir a melhor sincronização dos relógios possível, de forma a que os eventos sejam processados pelos vários nós simultaneamente, da maneira como o utilizador espera que aconteça.

No final de cada ronda este módulo recolhe todos os *logs* produzidos pelos contentores e controlador secundário, para posterior análise.

3.3 Controlador Secundário

O Controlador Secundário é um módulo que se executa em todos os servidores onde a experiência corre. Este controlador tem duas funções, injetar faltas nos contentores a correr no seu nó e recolher métricas durante a experiência.

Este módulo regista métricas de utilização dos recursos do nó num ficheiro, em cada nó do *cluster*, durante cada ronda da experiência. Estas métricas incluem utilização da rede, operações de disco, CPU e memória. Esta informação é guardada num ficheiro e enviada no fim da experiência, de forma a minimizar o impacto no desempenho por parte do *FaultSee*. A seria enviar durante o decorrer da experiência, mas iria congestionar a rede, o que introduzia ruído nos resultados obtidos. Adicionalmente, caso o utilizador especifique a intenção de monitorizar ao detalhe os contentores de um serviço, este módulo é também responsável por registar métricas semelhantes às do nó, mas individualmente para cada contentor do serviço. Além das métricas de utilização de recursos, também os *logs* dos serviços são armazenados no ficheiro, para o utilizador poder analisar em detalhe o que aconteceu ao longo da experiência.

No início de cada ronda, o controlador secundário processa o guião e sincroniza o relógio com o controlador primário. Ao processar o guião, o controlador secundário trata de preencher uma lista interna ordenada por tempo onde regista cada evento. Isto inclui, nomeadamente qual o contentor ao qual o evento é aplicado e ainda o momento (temporal) em que é para aplicar. A informação organizada desta maneira simplifica o processo de iterar todos os eventos à medida que a ronda decorre.

No início de cada ronda o controlador secundário não tem maneira de saber em que nó os contentores *Docker* são lançados, pelo que tem de guardar todos os eventos que têm de ser processados. Adicionalmente, no momento de injetar a falta, o controlador tem de perceber rapidamente se o contentor alvo está

a correr no nó que está a controlar. Para cumprir este requisito, o controlador guarda esta informação numa tabela de dispersão, que permite um rápido acesso. Para preencher esta tabela, o controlador escuta uma *API* de eventos do *Docker*, onde todos os eventos que ocorrem são publicitados, em particular os eventos que informam que contentores começaram ou pararam de correr. De notar que este processo ocorre simultaneamente e independentemente em todos os nós do *cluster*.

4 Avaliação

Nesta secção avaliamos o *FaultSee* recorrendo a um cenário realista com a base de dados *Apache Cassandra*. A avaliação tem dois objetivos principais, mostrar que a descrição compacta do guião permite descrever comportamentos complexos, e ilustrar a necessidade de especificação de sistemas reais. O objetivo é mostrar que um utilizador facilmente cria experiências que depois podem ser repetidas. Ambos os cenários apresentados nesta secção foram executados num *cluster* de 6 servidores da *Google Cloud*. O primeiro servidor é uma instância *g1-small*[1], que executou somente o Controlador Principal. Os outros 5 servidores são instâncias *n1-standard-1*[2]. Nos primeiros 4 correu o *cluster* de *Cassandra*, enquanto na última instância correu os 2 clientes de *YCSB*.

4.1 Comportamento com uma falta

O primeiro cenário estudado analisa o comportamento do sistema *Cassandra* perante o *benchmark YCSB*, com a injeção de uma falta, matando um dos nós, durante a execução do *benchmark*. Ainda antes do fim do *benchmark* um novo nó é lançado para testar qual o impacto de adicionar num sistema que esteja a correr.

O guião é o seguinte:

- Lançamento de 4 nós *Cassandra* - Versão 3.11.4;
- Carregamento dos dados necessários a correr o *benchmark*, com replicação em 3 nós;
- Início do *benchmark YCSB* - Versão 0.14 (Modo 50% de leituras e 50% de escritas);
- Terminação abrupta de um nó do *cluster* de *Cassandra*;
- Lançar um nó após 700 segundos.

Na Figura 2 podemos observar a variação do número de instâncias ao longo do tempo do serviço *Cassandra*, linha a tracejado azul, e do serviço YCSB, linha contínua amarela. No gráfico podemos observar a adição de um nó de *Cassandra* de cada vez até atingir um total de 4 nós por volta do segundo 600. Por volta do segundo 1 400, são lançados 2 clientes YCSB, cada um com

[1] Servidor partilhado, com 0.5 vCPU e 1.70 GB de memoria.
[2] Servidor com1 vCPU e 3.75 GB de memória.

10 *threads*, este serviço corre continuamente durante aproximadamente 2 300 segundos, fazendo um total de 5 750 000 operações, metade são escritas e a outra metade leituras. No segundo 2 000, um dos contentores do serviço *Cassandra* é terminado repentinamente, sendo lançado um novo contentor para o substituir passados 700 segundos.

Na Figura 3 é possível observar o número médio de operações por segundo realizado por cada um dos clientes *YCSB*. Após alguns segundos necessários para o sistema atingir um valor mais estável, vemos que o número médio de operações para cada cliente é de aproximadamente 2 750 por segundo. Ao segundo 2 050 vemos o desempenho do *Cassandra* a baixar consideravelmente durante um curto espaço de tempo. Isto deve-se ao sistema ter sido informado que o nó que deixou de responder está de facto terminado, como tal o *cluster* reage replicando os dados armazenados até atingir novamente um valor de replicação 3. Após esta operação estar concluída podemos observar que o número de operações estabiliza em 2 250, um valor substancialmente inferior ao anterior, explicado pelo facto de existir menos uma réplica para responder aos pedidos do cliente. No segundo 2 700 é lançado um novo contentor para substituir o nó que está em baixo. Isto provoca uma nova perda de desempenho, uma vez que o *cluster* de *Cassandra* decide repartir parte dos dados armazenados para o novo nó, gastando recursos que não podem desta forma ser aproveitados para responder aos pedidos do cliente. Aproximadamente 200 segundos depois a operação termina e o desempenho volta a melhorar, estabilizando nas 2 500 operações.

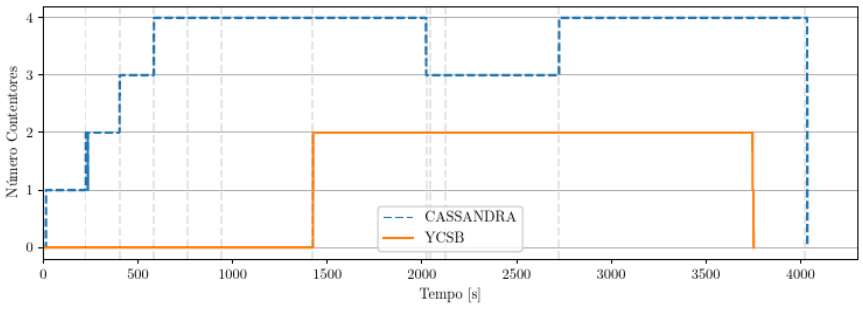

Figura 2. Número de Contentores no decorrer da Experiência no Cenário Terminação de um Contentor

4.2 Exaustão de Recursos

O segundo cenário pretendemos mostrar como a exaustão de recursos afecta o desempenho de um *cluster* de *Cassandra*, correndo na mesma o *benchmark* de *YCSB*. Durante 700 segundos ocupamos a totalidade do CPU de todos os contentores do serviço *Cassandra*.

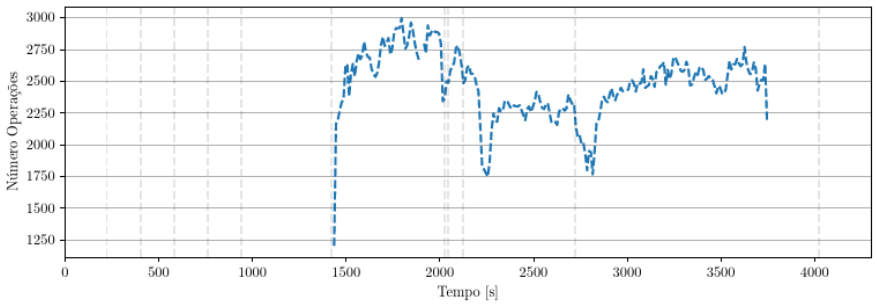

Figura 3. Número Médio de Operações dos Clientes YCSB no Cenário Terminação de um Contentor

O guião é o seguinte:

- Lançamento de 4 nós *Cassandra* - Versão 3.11.4;
- Carregamento dos dados necessários a correr o *benchmark*, com replicação em 3 nós;
- Início do *benchmark YCSB* - Versão 0.14 (Modo 50% de leituras e 50% de escritas);
- Ocupar totalmente o CPU durante 700 seguntos, aproximadamente 12 minutos.

Na Figura 4 podemos observar a variação do número de instâncias ao longo do tempo do serviço *Cassandra*, linha a tracejado azul, e do serviço *YCSB*, linha contínua amarela. O gráfico apresentado é muito semelhante ao anterior, no entanto neste caso o número de réplicas do serviço *Cassandra* mantém-se constante, sendo de resto mantidas as mesmas condições.

Na Figura 5 é possível observar o número médio de operações por segundo por cada um dos clientes *YCSB*. Após alguns segundos para chegar a um número estável, o número de operações estabiliza numa média de 2 500 por segundo. No segundo 2 000 é injetada a falta que consome CPU, que se vai manter durante 700 segundos. Podemos observar que o desempenho baixa consideravelmente, para aproximadamente 1 750 operações por segundo. Assim que o CPU é libertado por volta do segundo 2 700 o desempenho melhora rapidamente, voltando a estabilizar nas 2 500 operações por segundo. No final é possível ver um pico em que o desempenho duplica, isto é explicado pelo facto de um dos clientes já ter efetuado todas as suas operações, pelo que o *cluster* de *Cassandra* passa ter metade dos pedidos simultâneos.

Através dos resultados obtidos podemos perceber que a injeção de faltas tem um impacto real no desempenho do *Apache Cassandra*. Em comparação, o uso intensivo de CPU teve um impacto mais negativo do que a remoção de um dos nós do *cluster*, como é visível ao compararmos a Figura 3 com a Figura 5.

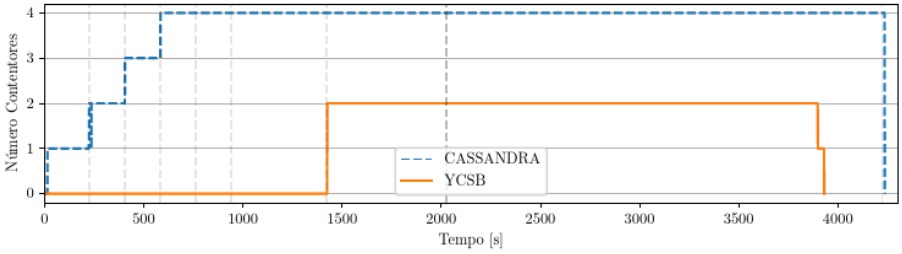

Figura 4. Número de Contentores no decorrer da Experiência no Cenário Exaustão de Recursos

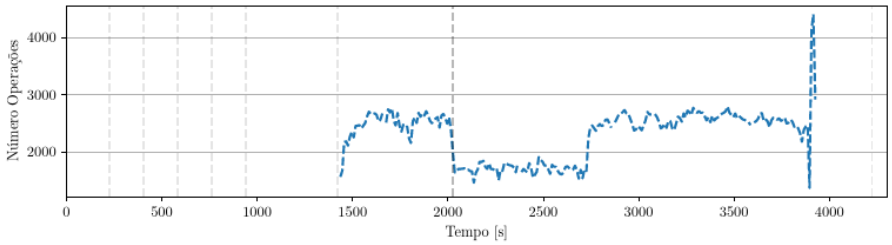

Figura 5. Número Médio de Operações dos Clientes YCSB no Cenário Exaustão de Recursos

5 Conclusão

Neste artigo apresentámos o *FaultSee*, uma nova ferramenta que permite automatizar a execução de cenários de falha em sistema distribuídos, uma necessidade cada vez maior pois os serviços operam cada vez mais numa escala maior e com maior número de componentes nos seus sistemas. Esta automatização permite melhorar o ciclo de vida de desenvolvimento de aplicações, ao permitir o rápido *feedback* aos programadores de possíveis erros introduzidos nas suas aplicações e configurações, aumentando também a confiança que os programadores têm em relação às aplicações que constroem.

Adicionalmente este sistema é uma melhoria para a investigação na ciência ao permitir que as experiências feitas por diferentes investigadores sejam reproduzíveis por outros investigadores independentes, necessitando apenas dos guiões utilizados pela equipa original e acesso a servidores idênticos.

Como trabalho futuro iremos desenvolver novas faltas que possam ser injetadas. Iremos também aumentar o número e diversidade de experiências-exemplo que serão disponibilizadas para toda a comunidade poder experimentar e modificar. Adicionalmente, iremos construir uma interface gráfica na qual será possível interagir com FaultSee durante o decorrer das experiências, permitindo não só montar a experiência, mas também manipular dinamicamente o guião. Iremos também automatizar a produção de gráficos com os resultados da experiência,

com a possibilidade de ver mais do que uma experiência em simultâneo, facilitando assim a comparação de duas versões diferentes de uma aplicação perante um mesmo cenário de teste.

Referências

1. Docker-Swarm overview | Docker Documentation, https://docs.docker.com/engine/swarm/
2. Kubernetes, https://kubernetes.io/
3. Cooper, B.F., Silberstein, A., Tam, E., Ramakrishnan, R., Sears, R.: Benchmarking cloud serving systems with YCSB. In: Proceedings of the 1st ACM symposium on Cloud computing - SoCC '10. p. 143. ACM Press, New York, New York, USA (2010). https://doi.org/10.1145/1807128.1807152, http://portal.acm.org/citation.cfm?doid=1807128.1807152
4. Docker Inc.: Docker Documentation (2018), https://docs.docker.com/engine/reference/builder/
5. Ganesan, A., Alagappan, R., Arpaci-Dusseau, A.C., Arpaci-Dusseau, R.H.: Redundancy Does Not Imply Fault Tolerance. ACM Transactions on Storage **13**(3), 1–33 (2017). https://doi.org/10.1145/3125497, http://dl.acm.org/citation.cfm?doid=3141876.3125497
6. Milka, G., Rzadca, K.: Dfuntest: A testing framework for distributed applications. In: Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics), vol. 10777 LNCS, pp. 395–405 (3 2018)
7. Munafò, M.R., Nosek, B.A., Bishop, D.V., Button, K.S., Chambers, C.D., Percie Du Sert, N., Simonsohn, U., Wagenmakers, E.J., Ware, J.J., Ioannidis, J.P.: A manifesto for reproducible science (1 2017). https://doi.org/10.1038/s41562-016-0021, http://www.nature.com/articles/s41562-016-0021
8. Oleksenko, O., Kuvaiskii, D., Bhatotia, P., Fetzer, C.: Fex: A Software Systems Evaluator. In: Proceedings - 47th Annual IEEE/IFIP International Conference on Dependable Systems and Networks, DSN 2017. pp. 543–550. IEEE (6 2017). https://doi.org/10.1109/DSN.2017.25, http://ieeexplore.ieee.org/document/8023152/
9. Tseitlin, A.: The antifragile organization. Communications of the ACM **56**(8), 40 (8 2013). https://doi.org/10.1145/2492007.2492022, http://dl.acm.org/citation.cfm?doid=2492007.2492022