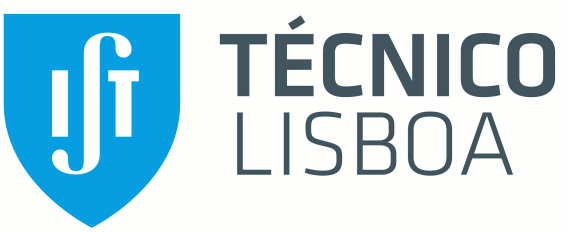

2019/2020

Computação em Nuvem e Virtualização

Sudoku Cloud

Grupo 27

Autores:

André Godinho (Nº 84006)
Sebastião Almeida (Nº 97115)

26/05/2020

1 Introdução

O objetivo deste trabalho passa por projetar um *elastic cluster of web servers* que é capaz de descobrir a solução de puzzles sudoku. Para isso, o sistema está desenhado em diferentes componentes que interagem entre si de forma a que seja possível uma distribuição e resolução eficiente dos pedidos feitos. A implementação foi feita em *Java* utlizando recursos da *Amazon Web Services*.

Para o checkpoint anterior foi feita uma análise exaustiva de diferentes métricas possíveis, tendo-se chegado a uma conclusão face a uma métrica eficiente que minimiza o overhead criado na instrumentação e carrega em si informação suficiente para uma previsão acertada do custo dum pedido. Para esta entrega final, o foco passou a ser em desenhar uma arquitetura que permite distribuir os pedidos feitos à aplicação pelos servidores *workers* de forma eficiente.

2 Arquitetura do Sistema

O sistema implementado é composto por quatro componentes essenciais: **Load Balancer**, **Auto-Scaler**, **Metrics Storage System** e **Web Servers**.

Em primeiro lugar, o **Load Balancer** é o ponto de entrada do sistema. Ou seja é a componente que faz a ligação entre o *front-end* e o *back-end* da arquitetura. Esta é a componente principal, pois é onde existe a lógica capaz de gerir todos os pedidos recebidos pela aplicação e o seu redirecionamento.

O **Metrics Storage System** tem a função de ajudar o **Load Balancer** a fazer uma boa previsão do custo de cada pedido recebido. Para tal, utiliza a base de dados *DynamoDB* da *AWS* e uma cache localizada no **Load Balancer** para otimizar a leitura e escrita na base de dados.

No que toca ao **Auto-scaler**, este é responsável por tornar o sistema elástico. Por outras palavras, com o aumento de pedidos por parte do cliente os servidores *workers* devem ser escalados. Caso aconteça o contrário, o número de servidores deve diminuir.

Por fim, os **Web Servers** têm a função de resolver os puzzles de Sudoku e enviarem a informação de quanto é que custou resolver cada pedido para o **Metrics Storage System**.

3 Load Balancer

Tendo em conta que os **Web Servers** têm características iguais, a base do nosso algoritmo de *load balancing* consiste no **least load**. Contudo, foi feita uma otimização essencial no que toca a quantos pedidos serão distribuídos em simultâneo. Por outras palavras, como os servidores funcionam em *multi-thread*, a acumulação de pedidos para um reduzido número de CPUs, como é o nosso caso, onde só temos disponível 1 CPU por instância irá comprometer a performance em termos de tempos de execução de cada um dos puzzles nesta instância.

Para resolver este problema, criou-se um mecanismo de bloqueio de instâncias. Em concreto, cada instância tem um limite máximo de carga e aceita pedidos enquanto não for ultrapassado esse limite. É evidente que este mecanismo provoca que certos pedidos possam ser encaixados numa instância enquanto que outros não, consoante o custo previsto desse pedido e independentemente do tempo de chegada do pedido. Para resolver esse problema, implementou-se uma lógica capaz de detetar se vários pedidos já ultrapassaram um pedido de maior dimensão que não cabe nos *workers* atualmente em processamento. Quando um pedido grande é ultrapassado a partir de um determinado limite, este pedido irá ser resolvido num novo servidor que irá ser ligado. Optou-se por esta alternativa em prol de se reservar uma instância ativa pois esta implementação permite uma **fault tolerance** mais simples.

De modo a que seja possível identificar cada tipo de pedido, foram definidos três tipos de pedidos:

1. Pedido pequeno - custo inferior a 15% do custo máximo
2. Pedido intermédio - custo compreendido entre 15% e 50% do custo máximo
3. Pedido grande - custo superior a 50% do custo máximo

Sendo o custo máximo igual a **38000**, pois é a carga do maior pedido do nosso dataset (obtido para o puzzle **25x25 DLX** com **625 missing elements**). É importante referir que o sistema é extensível a um pedido novo com maior custo de modo a que seja possível escalar o sistema.

3.1 Algoritmo

Com o intuito de pôr em prática este mecanismo, toda a implementação se baseia num mecanismo de ***polling***. Consoante o pedido recebido por uma thread, esta terá diferentes valores de tempos de espera antes de verificar condições. Consequentemente, certas threads acederão primeiro a certos recursos. Deu-se prioridade aos pedidos pequenos e grandes. Em concreto, pretende-se que os pedidos pequenos consigam ser encaixados com facilidade em instâncias disponíveis para os receber uma vez que são pedidos com resolução mais rápida, pois não faria sentido o cliente ter de esperar imenso tempo para os receber. Deu-se também prioridade aos pedidos grandes para evitar que entrem em *starvation*.

Assim que o **Load balancer** recebe um pedido novo, irá ser aberta uma nova thread neste servidor *master*. Após ter sido feito o *parse* da *query* obtém-se a previsão para o pedido recebido. Este processo será explicado em maior detalhe na secção **Metrics Storage System**. A partir deste ponto já se sabe qual dos três tipos de pedido é que a thread está encarregue.

De seguida, a thread irá entrar num *nested loop*. O primeiro *loop* será usado para **fault tolerance**, ou seja, para a reexecução de pedidos falhados quando o *http request* falha ao comunicar com o servidor *worker*, nomeadamente em caso do processo do **Web Server** ser desligado. Enquanto que o segundo *loop*

serve para verificar se já existe um servidor disponível para o pedido. Repare-se que ao utilizar o algoritmo de **least load** percebe-se facilmente se o sistema está ou não disponível para receber um novo pedido. De facto, caso a instância escolhida não tiver carga disponível, então nenhuma das outras tem. É importante referir que a carga é máxima caso a instância tenha uma carga superior a uma acumulação de pedidos com custo superior ao do pedido **25x25 DLX** com **625 missing elements**. Esta carga máxima é extensível a um novo valor de carga máxima de modo que seja possível escalar o sistema.

A primeira condição no *inner loop* consiste em verificar se o pedido é grande e já ultrapassou o limite de ultrapassagens. Definiu-se este limite de acordo com um **acumulador que ao ser superior a 20% do custo do pedido a ser tratado pela thread**, então não são permitidas mais ultrapassagens. Optou-se por esta implementação pois a melhor estimativa disponível do tempo total de execuções de diferentes ultrapassagens é dada pela **acumulação dos custos** desses pedidos que ultrapassaram e não, por exemplo, **por quantos pedidos** é que já ultrapassaram o pedido grande.

Com o objetivo de evitar que existam **active waits**, a segunda condição verifica qual é o tipo de pedido e antes de se saber se existe um servidor disponível (executar o **least load**) a thread é colocada em espera consoante o pedido:

1. Para pedidos pequenos, a thread é adormecida durante 1 segundo.
2. Para pedidos grandes, a thread é adormecida durante 1 segundo.
3. Para pedidos intermédios, a thread é adormecida durante 2 segundos.
4. Para pedidos pequenos quando não existem pedidos grandes à espera, a thread não é adormecida.

Optou-se por não adormecer a thread para pedidos pequenos quando não existam pedidos em espera porque esperar um segundo pode ser significativo para pedidos extremamente pequenos. Os restantes tempos foram definidos desta forma para dar prioridade aos pedidos pequenos e grandes.

Após estas condições as threads irão obter a instância que está com menor carga. Esta operação é uma **critical section** para garantir que a instância escolhida é a certa.

Após correr o **least load**, existem cinco casos possíveis:

1. A instância escolhida tem a carga no máximo e volta-se ao início do *inner loop*
2. Foi feito um pedido pesado, cujo limite de ultrapassagem já foi accionado. Irá ser reservada uma nova instância para este pedido. Contudo, esta reserva apenas garante que este pedido é o primeiro a chegar a esta instância, podendo ela logo de seguida receber mais pedidos.
3. A instância escolhida não está cheia e o pedido a ser executado é pequeno. O pedido irá ser direcionado para essa instância e a carga da instância é atualizada somando o custo deste pedido. Atualiza-se os acumuladores de todos os pedidos grandes à espera com o custo deste pedido pequeno.
4. A instância escolhida não está cheia e o pedido a ser executado é intermédio. O pedido irá ser direcionado para essa instância e a carga da instância é atualizada somando o custo deste pedido. Atualiza-se os acumuladores de todos os pedidos grandes à espera com o custo deste pedido intermédio.
5. A instância escolhida não está cheia e o pedido a ser executado é grande. O pedido irá ser direcionado para essa instância e a carga da instância é atualizada somando o custo deste pedido.

Caso aconteça a segunda situação, sai-se do *inner loop* e irá ser feito um *http request* após ter sido aberto o novo *worker*. Ao receber a resposta, desconta-se o custo do pedido enviado à carga da instância e envia-se a solução do puzzle para o cliente.

Ao acontecer o terceiro, quarto ou quinto caso, sai-se do *inner loop* e irá ser feito um *http request* para a instância disponível. Ao receber a resposta, desconta-se o custo do pedido enviado à carga da instância e envia-se a solução para o cliente.

Para qualquer um destes casos, caso o *http request* falhe, volta-se ao início do *outer loop*, recomeçando o direcionamento do pedido para outra instância caso esta instância seja declarada como **unhealthy** durante os **health checks**.

3.2 Estruturas de dados

No **Load balancer** existem três estruturas de dados essenciais para a execução.

1. Um hashmap que guarda a informação de cada um dos ***running*** *workers*. A chave é o ID da instância e o valor é um objeto da classe *MyInstance* que encapsula a informação relativa a cada instância (ID, IP, carga, número de pedidos na instância, se está *healthy*, se está reservada, etc).
2. Um concurrent hashmap que guarda a informação dos pedidos grandes em espera. A chave é o ID da thread, pois cada pedido grande está a ser executado por uma thread específica. O valor para cada chave é um objeto da classe *Request* que encapsula toda a informação para um pedido. Em concreto, os parâmetros do pedido (estratégia a usar, tamanho, nome do puzzle e missing elements), o acumulador de ultrapassagens e o custo previsto para este pedido.
3. Um concurrent hashmap que funcionará como uma cache. A chave é uma primary key dos pedidos guardados em memória e o valor é um objeto da classe *CachedSample* que encapsula toda a informação no que toca aos parâmetros dum tipo de pedido e ao custo real desse tipo de pedido.

Nesta secção ir-se-á analisar em maior detalhe as duas primeiras estrutura de dados, enquanto que na secção do **Metrics Storage System** analisar-se-á a terceira estrutura de dados.

Ao ligar-se novas instâncias, a complexidade de inserção na primeira estrutura de dados enunciada é *O(1)*. Para além disso, quando se pretende aceder a uma determinada instância, por exemplo, para a remover, o acesso tem também complexidade *O(1)*.

No que toca a memória, usar-se um hashmap tem também a vantagem de a memória ser alocada dinamicamente, o que é eficiente pois o nosso cluster é elástico.

Os métodos que lidam com operações concurrentes que pudessem levar a incoerências nesta estrutura de dados foram implementados como **critical sections**. Por exemplo, ao ser ligada uma nova instância, ao remover-se uma instância e ao aplicar o **least load** para obter a instância com menos carga.

As complexidades referidas em cima também se aplicam à segunda estrutura de dados, o que permite uma inserção e remoção eficiente dos pedidos grandes nesta estrutura.

3.3 Future work

Acrescentamos que para **future work** seria ideal implementar a primeira estrutura de dados como *concurrent hashmap* pois facilita a implementação do programador ao lidar com problemas de concurrência. Para além disso, a eficiência de acessos é drásticamente superior porque o *concurrent hashmap* permite que haja **atomic operations**, ou seja, serem bloqueadas certas posições de memória da estrutura de dados em vez de toda a estrutura de dados ser bloqueada através de **critical sections**.

Uma vez que se implementou o **least load**, faria sentido ter uma estrutura mais eficiente para obter a instância com menor carga. Em concreto, criar-se-ia uma lista ordenada para guardar as instâncias ordenadas consoante o valor da carga. Ao alterar-se o valor da carga de uma instância procurava-se nesta lista usando **binary search** que tem uma complexidade média de *O(log(N))* e reordenava-se a lista usando *quicksort* que tem uma compleidade média de *O(N log(N))*. Assim, os acessos à instância com menor carga passariam a ter uma complexidade de *O(1)*, pois seria o primeiro elemento da lista.

A última melhoria que deixamos para **future work** à nossa implementação, que não foi implementada pois tornava mais díficil a conexão *http* entre o **cliente** e o **load balancer**, consiste em criar uma fila para cada tipo de pedido recebidos. Sucintamente, os pedidos entrariam nas fila consoante a sua ordem de chegada e custo previsto. De seguida, seriam reencaminhados para os solvers à medida que houvesse espaço nos servidores. O mecanismo de ultrapassagens implementado podia ser aproveitado para reservar uma instância para um pedido grande que já tivesse ultrapassado o limite de ultrapassagens por pedidos que saíram das filas de pedidos pequenos e intermédios. Claro que a dificuldade desta implementação está em manter a conexão *http* com o cliente aberta, para que ao se obter a solução ela ainda possa ser reencaminhada para o cliente. Certamente que o objeto que trata desta conexão *http* teria de ser enviado em conjunto com o objeto que guarda a informação do pedido. Para além disso, alteraria-se a implementação que usa **polling** para a utilização de **monitors**. Desta forma, as threads não estão continuamente a verificar condições e colocadas a dormir (o que não é eficiente em termos de utilização de CPU no **load balancer**). Dito de outro modo, os pedidos só seriam enviados para os *workers* assim que uma certa condição tomasse um valor e a thread acordasse para remover o pedido da fila e enviá-lo para o servidor com menor carga.

4 Auto Scaler

O sistema tem de ser capaz de reagir consoante os pedidos feitos pelo cliente. Por outras palavras, quando se recebe uma grande quantidade de pedidos, o sistema deve ser capaz de escalar o número de *workers* para ser capaz de responder aos pedidos rapidamente. De modo oposto, quando existe uma pouca quantidade de pedidos, o sistema deve ter um número de instâncias suficientes de modo a que consiga satisfazer os pedidos e ter uma gestão eficiente dos créditos a serem utilizados.

4.1 Algoritmo

Tendo em conta os testes feitos, optou-se por manter no mínimo duas instâncias ligadas ao longo do tempo. Em concreto, o sistema ao ligar irá ligar automaticamente duas instâncias e caso seja ativada uma política de **scale down**, não se irá desligar uma instância caso haja apenas duas ativas. Caso se desligue uma instância ou haja uma falha numa das instâncias e ela seja removida por ficar **unhealthy**, o sistema irá ajustar-se a ter duas instâncias **healthy** automaticamente. Os detalhes de implementação serão abordados na secção **Health checks e verificação de instâncias**.

No que toca ao algoritmo, existe um objeto da classe *timer* no **Load balancer** que é responsável por, de 15 em 15 segundos, **percorrer todos os healthy *workers*** e calcular a carga média nestas instãncias. Mais precisamente, a carga para cada instância é dada pelo quociente entre a soma dos custos previstos nessa instância e o custo máximo definido anteriormente (38000). Tendo os valores calculadores para estas instâncias, basta então fazer a média e obtém-se uma amostra da média de cargas para este instante. É importante referir as médias devem ser feitas sobre as instâncias **healthy** e não incluir instâncias que estejam **unhealthy** ou que ainda estejam a ser ligadas.

Ao fim de existirem 8 amostras de medições de médias, ou seja, ao fim de dois minutos, é calculada a média destas médias. Consoante esse valor, uma de duas **auto scalling policies** é ativada.

1. **Scale up policy** - a média calculada sobre os dois minutos é **superior a 50%** e irá ser ligada uma nova instância.

2. **Scale down policy** - a média calculada sobre os dois minutos é **inferior a 30%** e irá ser desligada uma instância **que tenha 0 pedidos a serem resolvidos**.

De modo que não sejam perdidos pedidos feitos pelo cliente, garantiu-se que apenas se removem instâncias que não tenham pedidos a serem resolvidos.

Os valores das cargas de cada instância estão guardadas nos objetos *MyInstance* que se encontram no *hashmap* definido na secção **3.2.**.

4.2 Future work

Possíveis otimizações a esta componente da arquitetura passariam por bloquear os pedidos para a instância escolhida pelo *auto scaller* a ser removida. Para a nossa implementação pode acontecer que demore bastante tempo a ser removida uma instância pois tem de existir uma instância com 0 pedidos.

Outra otimização seria escalar o sistema tendo em conta os custos dos pedidos em espera, de modo que caso existissem bastantes pedidos em espera, poder escalar um número dinâmico de servidores em vez de se incrementar o número de instâncias ativas.

5 Metrics Storage System

Para o **Load Balancer** fazer uma previsão do custo de cada pedido, é necessário guardar o custo verdadeiro de cada pedido após ter sido resolvido e instrumentado nos *workers*. Posteriormente, o **Load balancer** acede a estes custos guardados e consoante o pedido recebido, irá procurar na base de dados por um pedido igual/semelhante.

Uma implementação simples, seria por um lado cada *worker* escrever na base de dados o custo do pedido que acabou de resolver. Pelo outro, o **Load balancer** fazer uma *query* com os parâmetros dum novo pedido recebido através duma leitura na base de dados. Esta solução tem diversos problemas **graves**.

Em primeiro lugar, iria haver um enorme número de *misses* nas leituras da base de dados por parte do **Load balancer**, dado que é pouco provável que existam pedidos iguais.

Em segundo lugar, o número de leituras e escritas à base de dados é maximo o que iria criar um *bottleneck*. Em concreto, sempre que é resolvido um pedido no *worker* e sempre que é

recebido um pedido no **Load balancer**, aceder-se-ia à **DynamoDB** para fazer uma operação de leitura no primeiro caso e de escrita no segundo. Repare-se que esta solução leva também a um maior dispêndio de créditos dado o enorme volume de leituras e escritas na **DynamoDB**.

Em terceiro lugar, fazer operações de leitura sobre todos os parâmetros dos pedidos tendo um número elevado de entradas na **DynamoDB** iria provocar leituras extremamente demoradas, o que iria prejudicar a latência de todos os pedidos.

5.1 Otimização com pedidos semelhantes

Para resolver o **primeiro problema enunciado**, decidiu-se não guardar o custo por cada pedido mas sim criar médias que agrupam pedidos semelhantes tendo em conta os parâmetros do pedido.

A nossa análise para o *checkpoint* demonstrou que para o algoritmo **DLX** o número de *missing elements* é totalmente irrelevante para o custo do pedido. Para além disso, o puzzle inserido também não tem grande influência no custo. Para este solver, o custo do puzzle é dado de forma praticamente deterministica através do tamanho do mesmo. Desta forma, para este puzzle ir-se-á guardar os custos dos pedidos tendo em conta o seu tamanho e calcula-se a média sobre esses valores. Ou seja, para o nosso *dataset* existirão apenas três valores guardados em memória para o solver **DLX**:

1. Custo médio para pedidos com tamanho 9x9
2. Custo médio para pedidos com tamanho 16x16
3. Custo médio para pedidos com tamanho 25x25

A implementação está extensível para receber pedidos com diferentes tamanhos.

Para os restantos pedidos, a situação é um pouco mais complexa, pois já dependem bastante do número de *missing elements*. Decidiu-se agrupar os pedidos por tamanho, estratégia, nome do puzzle e **intervalos de *missing elements***. Em concreto, para o âmbito deste projeto foram definidos três intervalos para cada tamanho. No entanto, para **future work** deixa-se a sugestão de se aumentar este número de intervalos consoante o aumento do tamanho do puzzle de modo que se consigam médias mais corretas pois já existirão discrepâncias consideráveis entre os valores nos intervalos.

Por exemplo, para o nosso *dataset* existem cinco puzzles diferentes com tamanho 9x9. Então iriam haver $5 \times 3 \times 2 = 30$ médias diferentes. De facto, existem três intervalos definidos de missing values (0-27; 28-54; 55-81, para puzzles 9x9) e duas estratégias diferentes (BFS e CP).

Para um pedido que use a estratégia **DLX**, o **Load balancer** irá aceder ao **Metrics Storage System** e verificar se existe uma amostra que tenha como atributo uma área igual à do novo pedido. Caso exista, usa esse custo como previsão, caso contrário atribui-se a esse pedido um custo igual a metade do pior custo já encontrado para o seu tamanho.

Para um pedido que use a estratégia de **CP** ou **BFS**, o **Load balanecer** irá aceder ao **Metrics Storage System** e verificar se existe uma amostra que tenha a mesma área, estratégia, nome do puzzle e intervalo de *missing elements*. Caso isto não aconteça, irá escolher uma amostra que tenha apenas a mesma área, estratégia e intervalo de *missing elements*. Em último caso, se nenhuma destas situações acontecer, é atribuído um custo igual a metade do pior custo já encontrado para o seu tamanho.

Por conseguinte, aplicando esta implementação o número de *misses* reduz substancialmente e o primeiro problema está solucionado.

5.2 Otimização com cache

Uma vez definida a forma como se vai guardar as amostras do **Metrics Storage System**, falta então definir em concreto onde é que se está a guardar os custos para estas amostras e como. Note-se que, com o objetivo em mente de resolver o **segundo e o terceiro problema enunciados anteriormente**.

A nossa implementação baseou-se em criar uma cache no **Load balancer** onde se irá guardar **todas as amostras das médias agrupadas até ao momento**. Para isto ser possível, deixar-se-á de escrever na **DynamoDB** no final de resolver e instrumentalizar cada pedido.

Em concreto, aproveitou-se a comunicação do *worker* com o **Load Balancer** ao enviar a solução do puzzle, e adicionou-se também o custo do pedido a esta resposta. Uma vez que cada thread está à espera da resposta ao *http request* e tem todos os parâmetros dos pedidos guardados em variáveis privadas, é possível escrever diretamente na cache após se ter recebido a resposta do *worker*. Claro que esta escrita é feita de duas formas, ou o pedido é o primeiro do seu tipo e é registada uma nova amostra na cache ou tem de se atualizar a média do seu pedido semelhante respetivo. Para fazer isto mesmo, implementou-se uma média incremental que se baseia na média currente desse pedido e no número de termos já encontrados desse tipo de pedido.

Tendo o mecanismo de escrita na nossa cache definido, a leitura é trivial. Perante um novo pedido, o **Load balancer** apenas tem de averiguar se existe ou não um pedido semelhante na cache.

Falta então perceber qual é a utilidade da **DynamoDB** na implementação feita. De facto, esta cache é suficiente para a previsão dos pedidos caso o **Load balancer nunca falhe**. Assim, aproveiou-se a **DynamoDB** para serem guardadas as médias calculadas numa base de dados **segura**. Por outras palavras, o sitema quando se liga faz um *scan* completo à **DynamoDB** e guarda as amostras em cache. Após esta leitura, não volta a ler da **DynamoDB**. De seguida, por segurança, irá escrever todas as médias guardadas em cache periodicamente (definiu-se um período de dois minutos).

Em suma, a nossa implementação consiste numa única leitura à **DynamoDB** ao ligar-se o **Load balancer** e em escritas periódicas à mesma, de modo a que seja possível salvaguardar as médias já registadas. Desta forma, não existe qualquer *bottleneck* estando o segundo problema resolvido. Para além de que desta forma a implementação feita sai substancialmente mais barata dado que existe apenas uma leitura À **DynamoDB** e escritas periódicas (cujo período pode até ser ajustado). O terceiro problema está também resolvido pois a nossa implementação baseia-se em estruturas de dados com acessos *O(1)* e *O(N)* (dependendo do pedido e do que exista na cache) e estas procuras são feitas localmente no **Load balancer**, não havendo qualquer leitura feita à **DynamoDB** o que iria sempre provocar um aumento de latência para todos os pedidos. Para além disso, o facto de se terem agrupamentos com médias de pedidos semelhantes, faz com que exista um número muito inferior de amostras em memória.

Uma objeção possível ao nosso sistema seria que perante imensos puzzles diferentes, estas amostras não caberão na cache. Gostaríamos de referir que isto é um problema resolvido ao escalar-se também a quantidade de servidores usados para o **Load balancer**. Para além disso, cada amostra na base de

dados ocupa pouca memória dado que se guarda médias representativas de uma enorme quantidade de pedidos.

5.3 Estruturas de dados

Para otimizar as leituras e escritas na cache, optou-se por utilizar um *concurrent hashmap* que tem também a vantagem das operações na cache serem *thread-safe*. Dado que irão haver imensas escritas e leituras ao longo do tempo, esta estrutura de dados é essencial pois permite um que os acessos sejam feitos através de **atomic operations**, bloqueando apenas a posição de memória onde se está a aceder.

No que toca às keys criadas, optou-se por criar uma espécie de *natural key* para cada entrada que regista as médias para um tipo de pedidos. Por exemplo, os pedidos de tamanho 25x25 que usem a estratégia **DLX** estão identificados de acordo com a chave: **"25x25DLX"**. Para as outras estratégias esta chave toma a seguinte forma:
"25x25BFSR3SUDOKU_PUZZLE_25x25_01", onde o 3 corresponde a pertencer ao terceiro intervalo de *missing values* e os caracteres a seguir identificam o nome do puzzle.

Ao receber um pedido, como o **Load balancer** contém todos os parâmetros necessários, consegue gerar a *natural key*. Desta forma, o acesso é feito através de *O(1)*. Para os casos em que os pedidos são do tipo BFS e CP e não há pedidos do mesmo puzzle, ir-se-á percorrer todos os elementos da cache. Seria possível de implementar um mecanismo de *parse* para achar outro puzzle com a mesma área, estratégia e intervalo de *missing elements*, o que permitiria para estes casos um acesso de *O(1)*, no entanto não houve tempo. Assim, para esses casos a complexidade é de *O(N)*.

6 Health checks e verificação de instâncias

É importante que o sistema consiga aperceber-se que existem instâncias para onde não devem ser enviados pedidos para evitar falhas. Estas instâncias podem não estar em bom estado ou o processo que está a correr o programa nos *workers* estar desligado.

Com este objetivo em mente, implementou-se um *timer* que envia *health checks* de 20 em 20 segundos para todas as instâncias através dum novo *endpoint*. Caso a instância esteja operacional irá ser retornado um *status code* = 200. Ao fim de três pedidos devidamente respondidos a instância é declarada como **healthy**. Caso algum destes falhe, a instância é imediatamente declarada como **unhealthy**, faz-se reset ao número de *healths checks* e incrementa-se um acumulador de **unhealthy checks**.

Existe também outro *timer* que desempenha três funções. A primeira consiste em desligar instâncias que têm mais do que 1 **unhealthy checks**. A segunda é averiguar se as *running instances* são coerentes com as que existem no *hashmap* referido na secção **3.2.**. Caso exista uma instância com um ID que não esteja nas *running instances* é retirado do *hashmap*. Isto faz parte do nosso mecanismo de **fault tolerance**, ao ser terminada uma instância, é este *timer* que remove a instância do nosso *hashmap*, pois ela já não faz parte das *running instances*. Por fim, a terceira função é averiguar se existe menos do que duas instâncias a correr e se assim for o caso, ligam-se instâncias até se terem duas disponiveis.

7 Fault tolerance

O sistema deve ser capaz de redirecionar os pedidos duma instância para outra ao ser terminado o processo no *worker* que resolve o puzzle ou em caso da instância ir abaixo.

Com este objetivo em mente, criou-se o *nested loop* explicado em detalhe na secção **3.1.**. Caso um *http request* **falhe** é levantada uma exceção e não se sai desse *nested loop*. Caso contrário, ao receber-se a resposta, a lógica implementada força que se saia desse *nested loop*.

Existe um pequeno detalhe na implementação que é importante referir. De facto, verificou-se que ao terminar uma instância, a *framework* usada (*Apache HttpClient*) não dá *throw* de uma exceção. Desta forma, acontece um ***deadlock***. Para solucionar este problema, definiu-se um *timeout* máximo de 10 minutos (pode ser ajustado), que quando é accionado obriga que a execução regresse ao início do *nested loop*. Tendo em conta os testes feitos, não houve problema com a implementação e isto só acontece para o caso em que se termina a instância. Caso se aceda ao *worker* (por *ssh*) e se termine o processo então é levantada uma exceção e acontece o que foi inicialmente explicado.

8 Conclusão

Neste trabalho procurou-se desenhar uma arquitetura de *elastic cluster of web servers*. Numa primeira fase, foi feita uma análise profunda de como varia o tempo de execução consoante os parâmetros dos pedidos, tendo-se chegado a uma função de custo que visa prever o tempo de execução desse pedido usando uma *BIT tool*. Para além disso, implementou-se a instrumentação nos *web servers* que permitem calcular parâmetros para esta função de custo.

Na fase seguinte, procurou-se projetar toda uma arquitetura complexa através duma implementação de blocos simples que mais tarde se foram unificando de tal forma a construir a arquitetura final. O **Load balancer, Auto scaler, Metrics Storage System e Web Server** foram desenhados com o intuito de otimizar a tarefa de cada um deles e no final foram unificados para trabalharem em constante cooperação.

Ao longo do relatório foram deixadas diversas sugestões para *future work* que tornariam o sistema mais eficiente mas não houve tempo para as pôr em prática. Contudo, foram pensadas.

Foi utlizada a *framework Apache HttpClient* para realizar as comunicações entre o **Load balancer** e os **Web servers**. Para além disso, usou-se a *AWS SDK* para realizar toda a implementação de gestão das intâncias da *AWS*.

9 Notas

É importante referir que por falta de tempo não foi possível implementar uma classe de pedidos caso se adicione um novo *solver*. Eles serão tratados como se fossem um pedido **DLX** no que toca tanto ao cálculo da função de custo como à forma de como se guardaria os pedidos na cache. No entanto, note-se que isto é uma implementação trivial.

Durante a implementação do **Web Server** fez-se uma alteração na classe *AbstractSudokuSolver* devido a um problema de concurrência entre threads ao receber pedidos com tamanhos diferentes. A alteração consistiu em alterar os atributos ***static SIZE*** e ***static BOX_SIZE*** para ***protected SIZE*** e ***protected BOX_SIZE*** . De facto, estas variáveis devem ser privadas a cada objeto da subclasse que herda da classe abstrata. Senão, ao ser resolvido em simultâneo um puzzle dum

determinado tamanho e um puzzle de maior dimensão, estas variáveis vão ser alteradas para os valores do puzzle de maior dimensão (quando ele é o segundo a chegar ao *worker*). Assim, o primeiro solver irá aceder a indíces não existentes para os *arrays* alocados levando a um erro.