

Computação em Nuvem

Grupo 9

65964 – Diogo Antunes

70614 – João Azevedo

## Introdução

No âmbito da disciplina de Computação na Nuvem e na tentativa de melhor aplicar os conceitos apreendidos durante o semestre lectivo, foi proposto que fizéssemos um sistema de processamento de *logfiles* capaz de responder a três questões básicas previamente estipuladas.

Conforme constava no enunciado do projecto, o sistema deveria estar organizado em três componentes:

- Uma aplicação de MapReduce cujo algoritmo recebe como *input logs* provenientes de células de uma rede e o manipula de modo a inserir numa base de dados de uma forma simplificada;
- Uma aplicação *web* que disponibiliza *queries* sobre o *output* da aplicação de MapReduce guardado na base de dados;
- Um mecanismo de *load balancing* que permite distribuir por múltiplas instâncias a carga da aplicação *web*.

De modo a chegar às melhores condições finais possíveis, foi necessário tomar um conjunto de decisões a nível da arquitectura do sistema.

Todas as decisões tentarão ser justificadas da melhor forma possível, sendo importante referir que a possibilidade criada de ter um crédito para usufruir do universo de produtos da *Amazon Web Services* foi naturalmente relevante no processo de escolha deste *service provider* como a base para todo o PhoneLogSystem.

## Aplicação MapReduce

O processo de desenvolvimento da aplicação de MapReduce iniciou-se localmente com a criação da *class* de Java *PhoneLogMapReduce* esta utilizava as bibliotecas fornecidas pela *Apache* e com a instalação do Hadoop.

Como seria suposto esta classe contém quer a classe de *mapper*, quer a classe de *reducer* tornando-se integralmente responsável pelo processamento e análise dos ficheiros de *logs*.

Observando as *queries* a que a aplicação *web* teria de responder:

- Dado o *phone-id* e a data, devolver a sequência de células visitadas pelo telemóvel nesse dado dia.
- Dado o *cell-id*, a data e a hora, listar os telemóveis presentes numa célula nesse momento.

- Dado o *phone-id* e a data, devolver os minutos que um dado telemóvel esteve ligado à rede.

Podemos constatar que apenas são relevantes cinco tipos de eventos (entradas e saídas da célula, conexão e desconexão da rede e *pings*) para o *mapper* sendo que aqueles relacionados com a rede apenas são importantes para os telemóveis.

## **FileInputFormat**

Como é natural, dado que o *input* da aplicação de MapReduce é um ficheiro, é utilizada a classe que trata deste tipo de objectos.

## **ValuesWritable**

Esta classe é utilizada para armazenar os *values* entre os *mappers* e os *reducers*, sendo as *keys* meras *strings* (Text).

Nos dados a armazenar temos três tipos:

- eventID;
- data/ hora;
- dados - que no caso de a *key* ser um *phone-id* se trata de um *cell-id* e vice versa.

A existência desta classe permite criar um maior nível em encapsulamento e assim melhorar a qualidade do código da aplicação, usufruindo das potencialidades do Hadoop.

## **MyDynamoDBOutputFormat**

Salvaguardando o devido mérito para quem o fez, existia um repositório no *github* [1] que continha implementações de *DynamoDBOutputFormat*, uma classe que estende *OutputFormat* e implementa a forma de escrever para uma base de dados de *DynamoDB* (futuramente referida). A necessidade de perceber bem esta classe, *DynamoDBConfiguration* e outras que derivam de *AttributeValue* conduziram a que não fosse possível utilizar este método de escrita na base de dados na entrega intermédia.

## **Elastic MapReduce**

Recorrendo à ajuda dos benefícios que a Amazon apresenta para o uso desta ferramenta são de salientar quatro:

- Não ser necessário ter em consideração factores como número de nós, *setup* do *cluster*, configuração do Hadoop, etc. Podendo apenas haver um foco na análise dos dados - Abstracção;
- Escalabilidade no número de instâncias que processam os dados;
- Monitorização fácil do processo de MapReduce com acesso a estatísticas como o número médio de tarefas de *map* e de *reduce*.
- Documentação acessível e detalhada.

Outros valores como o custo podem ser relevantes numa situação normal mas não podemos dizer que foram importantes no projecto.

## Base de dados - DynamoDB

Uma vez que estávamos a utilizar o EMR, consideramos três soluções para o armazenamento dos dados resultantes do MapReduce:

- Amazon Simple Storage Service (Amazon S3)
- Amazon Relational Database Service (Amazon RDS)
- Amazon DynamoDB

A não escolha do Amazon S3 prende-se com o facto de se tratar de um sistema de ficheiros clássico em que os dados se apresentam em blocos de dimensões demasiado variáveis e de forma pouco estruturada. Por outro lado, o facto de o Amazon RDS se tratar de uma base de dados relacional fazia com que grandes quantidades de dados como o caso do *output* do MapReduce fossem dificilmente armazenadas. Sendo assim, e porque a simplicidade da estruturação da informação e o facto de as *queries* não serem realmente elaboradas, conduziu à escolha da Amazon DynamoDB.

## Estrutura da Informação

Foram criadas duas tabelas para facilitar as leituras da aplicação *web* uma vez que cada uma das *queries* apenas afecta uma das duas.

Sendo assim, temos:

- cellrecords < cellid (string), datetime (string), phones[ ] (strings) >
- phonerecords < phonenummer (string), date (string), minutesonnet (number), trace [ ] (strings) >

## Aplicação Web e Mecanismo de *Load Balancing*

Um dos objectivos de uma aplicação *web* consiste em estar disponível o máximo de tempo possível, e que consiga responder aos pedidos por parte dos clientes rapidamente e da melhor forma. Para que tal seja possível, foi necessário implementar um mecanismo de Load-Balancing.

Para implementar a nossa aplicação *web* utilizamos uma linguagem baseada em Javascript e no Google Engine V8, denominada por Node.JS. Optamos por Node.JS por ser uma linguagem bastante recente, com muita aderência, e haver imensa documentação.

Posto isto passamos a explicação da implementação da nossa aplicação *web*. Numa primeira tentativa optamos por criar uma instância AWS EC2 a correr Amazon Linux AMI de tamanho t1.micro, por ser elegível pelo *free-tier* da Amazon, e por fornecer hardware mais do que necessário para a nossa aplicação. De seguida tivemos de instalar um interpretador de Node.JS para poder correr a aplicação, e por fim criar um *script* que inicie a aplicação web cada vez que a máquina se ligue, este *script* serve para o caso de iniciarmos novas instâncias, copias desta máquina, estas possam estar logo preparadas a correr a aplicação *web*. Por fim gravamos uma imagem desta máquina para ser replicada no mecanismo de *Auto Scalling* da Amazon (futuramente referido).

No entanto para aplicar o sistema de *Load Balancing* ainda seria necessário criar um load-balancer, e definir uma politica de *Auto Scalling*, baseada em métricas de utilização do serviço, monitorizadas pelo serviço da Amazon Cloudwatch.

Após alguma pesquisa sobre os serviços disponíveis e as suas vantagens, optamos por usar o serviço da Amazon, AWS Elastic Beanstalk.

Este serviço proporciona alguma abstracção na criação de instâncias EC2, uma maior rapidez na gestão da aplicação *web*, melhor monitorização e fácil escalonamento.

Para adicionar uma aplicação ao AWS Elastic Beanstalk, basta desenvolver a aplicação, no nosso caso em Node.JS, localmente e de seguida fazer *upload* para um repositório *git*, correr um *script* do AWS Elastic Beanstalk (Figura 1.) onde escolhemos as preferências iniciais da nossa máquina, e por fim fazer *upload* do *git* para a máquina criada na Amazon.

```

To get your AWS Access Key ID and Secret Access Key,
visit "https://aws-portal.amazon.com/gp/aws/securityCredentials".
Enter your AWS Access Key ID (current value is "AKIAJ*****2SQCA")>:
Enter your AWS Secret Access Key (current value is "Le5C2*****sXuZU")>:
Select an AWS Elastic Beanstalk service region.
Available service regions are:
1) US East (Virginia)
2) US West (Oregon)
3) US West (North California)
4) EU West (Ireland)
5) Asia Pacific (Singapore)
6) Asia Pacific (Tokyo)
7) Asia Pacific (Sydney)
8) South America (Sao Paulo)
Select (1 to 8): 2
Enter an AWS Elastic Beanstalk application name (auto-generated value is "asd")>:
Enter an AWS Elastic Beanstalk environment name (auto-generated value is "asd-en
v")>:
Select a solution stack.
Available solution stacks are:
1) 32bit Amazon Linux 2013.09 running PHP 5.4
2) 64bit Amazon Linux 2013.09 running PHP 5.4
3) 64bit Amazon Linux 2013.09 running PHP 5.5
4) 32bit Amazon Linux 2013.09 running PHP 5.5
5) 32bit Amazon Linux running PHP 5.3
6) 64bit Amazon Linux running PHP 5.3
7) 32bit Amazon Linux 2013.09 running Node.js
8) 64bit Amazon Linux 2013.09 running Node.js
9) 64bit Windows Server 2008 R2 running IIS 7.5
10) 64bit Windows Server 2012 running IIS 8
11) 32bit Amazon Linux 2013.09 running Tomcat 7 Java 7
12) 64bit Amazon Linux 2013.09 running Tomcat 7 Java 7
13) 32bit Amazon Linux 2013.09 running Tomcat 7 Java 6
14) 64bit Amazon Linux 2013.09 running Tomcat 7 Java 6
15) 32bit Amazon Linux running Tomcat 7
16) 64bit Amazon Linux running Tomcat 7
17) 32bit Amazon Linux running Tomcat 6
18) 64bit Amazon Linux running Tomcat 6
19) 32bit Amazon Linux 2013.09 running Python 2.7
20) 64bit Amazon Linux 2013.09 running Python 2.7
21) 32bit Amazon Linux 2013.09 running Python
22) 64bit Amazon Linux 2013.09 running Python
23) 32bit Amazon Linux running Python
24) 64bit Amazon Linux running Python
25) 32bit Amazon Linux 2013.09 running Ruby 1.8.7
26) 64bit Amazon Linux 2013.09 running Ruby 1.8.7
27) 32bit Amazon Linux 2013.09 running Ruby 1.9.3
28) 64bit Amazon Linux 2013.09 running Ruby 1.9.3
Select (1 to 28): 7
Select an environment type.
Available environment types are:
1) LoadBalanced
2) SingleInstance
Select (1 to 2): 1
Create an RDS DB Instance? [y/n]: n
Attach an instance profile (current value is "[Create a default instance profile
]")>:
1) [Create a default instance profile]
2) aws-elasticbeanstalk-ec2-role
3) [Other instance profile]

```

Figura 1

Após a criação da máquina passamos a criação de um load-balancer de forma a poder desviar pedidos a nossa aplicação *web*, e distribui-los por várias máquinas, impedindo assim a sobrecarga do sistema, e aumentando a disponibilidade da aplicação.

Apesar de, através do AWS Elastic Beanstalk, criar um load-balancer seja apenas definir métricas para disparar o serviço de *Auto Scalling* (Figura 2.), por trás as coisas funcionam de forma diferente. A forma como o Amazon gere um load-balancer é a seguinte:

1. Criação de um elastic load balancing, que permite distribuir o tráfego de entrada das aplicações em várias instâncias EC2;
2. Adicionar instâncias EC2, já criadas ao load-balancer;
3. E por fim é atribuído um DNS A Record, aonde os utilizadores se podem ligar.

Auto Scaling

Use the following settings to control auto scaling behavior. [Learn more](#).

Minimum instance count:  Minimum number of instances to run.

Maximum instance count:  Maximum number of instances to run. Must be less than 10000.

Availability Zones:  Number of Availability Zones to run in.

Custom Availability Zones:  Specific Availability Zones to launch instances in.

Scaling cooldown (seconds):  The amount of time after a scaling activity before any further trigger-related scaling activities can occur.

Figura 2

Esta solução tem algumas desvantagens, em primeiro lugar exige que as instâncias associadas ao load-balancer estejam operacionais, e sejam estáticas. Ou seja, não existe uma política de criação de novas máquinas, ou de remoção de máquinas em desuso. Isto faz com que existam recursos em desperdício, e por outro lado pode levar a um *bottleneck* do serviço, devido a ser um número estático de máquinas, que podem estar todas em sobrecarga.

Para que o load-balancer possa criar novas máquinas ou eliminar máquinas quando apropriado, é necessário utilizar o serviço da *Amazon Auto Scalling*, que passa pela definição de políticas e métricas baseadas em *CPU usage*, *memory usage*, *network usage*, etc.

Desta forma podemos adicionar novas instâncias EC2 quando o serviço se encontra em sobrecarga, ou eliminar instâncias em funcionamento, quando o sistema se encontra com pouca carga.

## Métricas

Um dos *triggers* mais utilizados para incrementar ou diminuir o número de máquinas de forma a manter o sistema disponível é o *CPU usage*. Esta métrica é importante devido ao facto de quando uma máquina se encontra com um *CPU load* elevado, 80%, 90% faz com que as aplicações se tornem lentas, ou até deixarem de responder por completo.

No entanto como a nossa aplicação apenas efetua *queries* à Amazon DynamoDB, não gera *CPU usage* suficiente para criar *bottleneck* na rede. Como tal criamos um *script* que gera stress ao *CPU* de forma a pudermos verificar o funcionamento do load-balancer com esta métrica. Este *script* calcula séries de *Fibonacci* num ciclo recursivo.

A métrica definida é a ilustrada na Figura 3. Que como podemos observar, foi definida com os parâmetros a seguir explicados:

Scaling Trigger

Trigger measurement:  The measure name associated with the metric the trigger uses.

Trigger statistic:  The statistic that the trigger uses when fetching metrics statistics to examine.

Unit of measurement:  The standard unit that the trigger uses when fetching metric statistics to examine.

Measurement period (minutes):  The period between metric evaluations.

Breach duration (minutes):  The amount of time used to determine the existence of a breach. The service looks at data between the current time and the number of minutes specified to see if a breach has occurred.

Upper threshold:  The upper limit for the metric. If the data points in the last breach duration exceed the threshold, the trigger is activated.

Upper breach scale increment:  The incremental amount to use when performing scaling activities when the upper threshold has been breached. Must be an integer, optionally followed by a % sign.

Lower threshold:  The lower limit for the metric. If all the data exceeded this threshold during a breach duration, the trigger is activated.



















Lower breach scale increment:  The incremental amount to use when performing scaling activities when the lower threshold has been breached. Must be an integer, optionally followed by a % sign.

Figura 3

A cada minuto o serviço de Cloudwatch da Amazon calcula a percentagem de *CPU* utilizada pelas máquinas associadas ao load-balancer, se durante um período de 2 minutos a percentagem de utilização do *CPU* exceder 80%, o serviço de *Auto Scalling* adiciona uma nova instância EC2 a correr a aplicação *web*, se baixar dos 20% remove uma instância EC2. Estes incrementos ou decrementos após ocorrerem existe um “*grace period*” onde o serviço de *Auto Scalling* ignora picos de utilização do *CPU*. Este período denomina-se de *Scaling Cooldown*, e foi definido para um valor de 300 segundos.

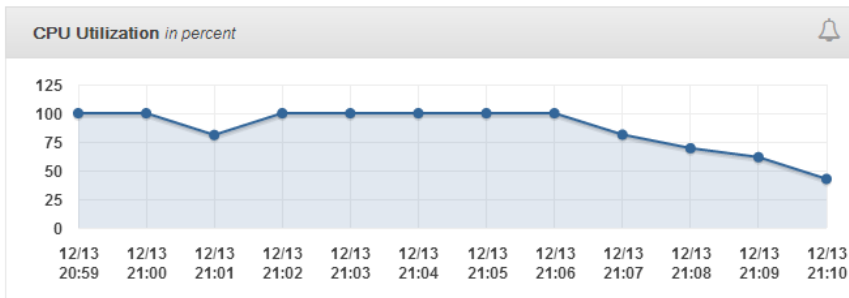
Novas instâncias EC2 apenas são iniciadas se o *CPU* exceder os 80% durante um período de 2 minutos, esta decisão tenta resolver um problema comum. O tempo que leva a instanciar uma máquina EC2, no nosso caso a *t1.micro*, e estar disponível, demora cerca de 2 minutos. Se houver um pico de utilização do serviço que leva-se a um uso excessivo do *CPU*, e fossem lançadas máquinas para responder a este problema, poderia acontecer que na verdade o uso do serviço foi apenas um pico de utilização e até a máquina ser instanciada e estar disponível já não havia pedidos para responder. Por outro lado pode acontecer, se a métrica esperar demasiado tempo para lançar o serviço de *Auto Scalling*, que novos clientes não obtenham resposta por parte da aplicação *web*.

Instanciamos uma máquina EC2, e utilizamos a ferramenta *siege*, que simula *http requests*, de forma gerar *CPU load* na nossa aplicação *web*. Testamos com 200 utilizadores em simultâneo a fazer 1000 *requests*, rapidamente atingimos o *CPU load*. Corremos este teste durante 60 minutos aonde observamos as 10 máquinas EC2 a serem instâncias e a carga a ser distribuída por todas, como se demonstra na Figura 4 e Figura 5.

<input type="checkbox"/>	cnApp-env	i-503fc966	t1.micro	us-west-2a	 running	 2/2 checks passed
<input type="checkbox"/>	cnApp-env	i-7ca26c4a	t1.micro	us-west-2b	 running	 2/2 checks passed
<input type="checkbox"/>	cnApp-env	i-6c7eb95a	t1.micro	us-west-2b	 running	 2/2 checks passed
<input type="checkbox"/>	cnApp-env	i-74b49940	t1.micro	us-west-2c	 running	 2/2 checks passed
<input type="checkbox"/>	cnApp-env	i-f5db88c1	t1.micro	us-west-2c	 running	 2/2 checks passed
<input type="checkbox"/>	cnApp-env	i-20595814	t1.micro	us-west-2c	 running	 2/2 checks passed
<input type="checkbox"/>	cnApp-env	i-ce1e84fa	t1.micro	us-west-2c	 running	 2/2 checks passed
<input type="checkbox"/>	cnApp-env	i-9607d2a0	t1.micro	us-west-2a	 running	 2/2 checks passed
<input type="checkbox"/>	cnApp-env	i-c60fe7f0	t1.micro	us-west-2a	 running	 2/2 checks passed
<input type="checkbox"/>	cnApp-env	i-ed9820db	t1.micro	us-west-2b	 running	 2/2 checks passed

**Figura 4**





**Figura 5**

Quando o teste terminou, uma a uma as máquinas EC2 foram sendo terminadas, e o CPU *load* estabilizou.

## Conclusões

O serviço de *Auto Scalling* da Amazon tem algumas falhas bastante visíveis com o exemplo anterior de *stress test*. Se um utilizador ao tentar aceder a aplicação *web* depara-se com o serviço indisponível devido a estarem demasiados *requests* a acontecer, o tempo entre o CloudWatch verificar a métrica definida no Auto Scalling, dar *trigger* do load-balancer, instanciar uma máquina EC2 e por fim a máquina estar disponível e o pedido do cliente ser redireccionado para essa maquina, demora cerca de 6 minutos ou mais. O que pode levar a uma perda de negocio.