ISEL
INSTITUTO SUPERIOR
DE ENGENHARIA DE LISBOA

MESTRADO EM ENGENHARIA INFORMÁTICA E DE COMPUTADORES

Computação Distribuída

Relatório do Trabalho Final

Semestre de Inverno 2020/2021

Docente:

- Luís Assunção

Realizado pelo Grupo 6:

- Edgar Alves nº 33017
- Paulo Pimenta nº 47972
- João Silva nº 42086

13/01/2021

Índice

Índice de Figuras

Introdução

Com este trabalho visamos aplicar conceitos de *gRPC* e *Spread* aprendidos nas aulas. Bem como consolidar os conhecimentos de *protobuff*, Maven, Intelij, que foram necessários também para os trabalhos anteriores.

Houve também um primeiro contacto com os serviços da Google Cloud onde nos foi possível criar máquinas virtuais em cloud e gerir as mesmas, de forma a tirar o melhor proveito para os nossos projetos.

Análise do Problema

Objetivo

Desenvolver um sistema de armazenamento de dados (Chave, Valor), num cluster de servidores. Este sistema tem de conter replicação de dados, equilíbrio de carga e tolerância a falhas o qual deve envolver um modelo de consistência por consenso entre todos os servidores. A comunicação entre os servidores do mesmo grupo será realizada utilizando Spread.

Especificações

Para este projeto será realizado utilizando como *middleware* gRPC (*Google Remote Procedure Call*) e o *Spread toolkit* de comunicação por grupos.

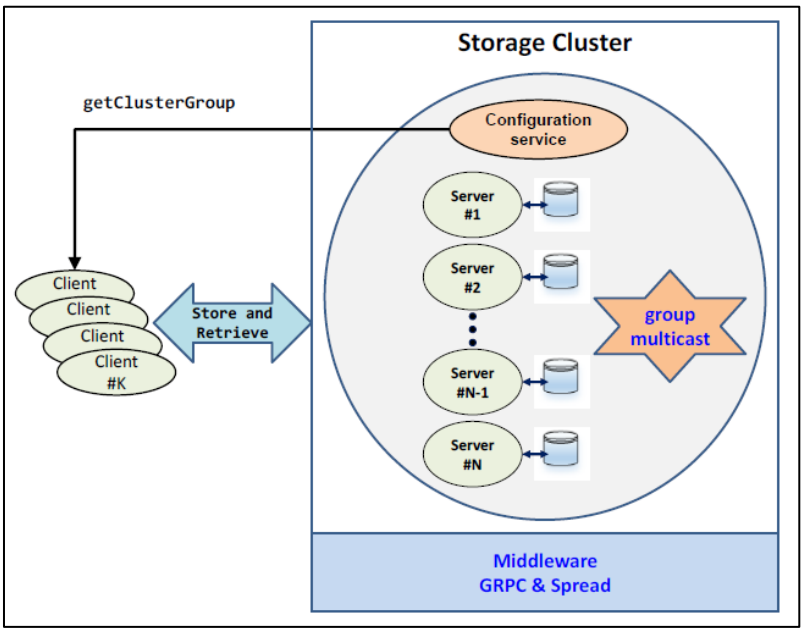

Figura 1 - Diagrama das partes envolvidas

Ao analisarmos a imagem retiramos as seguintes informações:

- Dentro de um grupo existe um serviço de configuração que detém todos os ip's dos elementos do grupo
- Todos os servidores possuem um ficheiro no qual iram guardar os dados;
- Dentro do mesmo grupo todos os servidores comunicam entre si utilizando *multicast*;
- Os clientes iram realizar um pedido (*getClusterGroup*) ao serviço de gestão de configurações para que este lhes forneça a lista de ip's do grupo de servidores disponíveis;
- Cada cliente só realiza pedidos de escrita e leitura aos servidores.

Requisitos

Funcionais

1. Criação de instâncias com IP e Porto bem definidos;
2. Dentro de cada grupo:
    a. Comunicação *multicast*;
    b. Grupo dinâmico suportado por *Spread toolkit*;
3. Serviço de gestão de configuração deverá ter um ip e porto bem conhecido e conhecer os ip's e portos dos servidores e os seus grupos.
4. Serviço de gestão de configuração deve estar atualizado e ter conhecimento de todas as entradas(*join*), saídas(*leaves*) e falhas de todos os servidores. Estes eventos devem ser sempre informados ao cliente através da operação *getClusterGroup* de *Stream* Servidor.
5. O ip's dos servidores seram retornados aos clientes através da operação *getClusterGroup*;
6. Caso haja falha na conexão a um servidor deverá ser repetida noutro;
7. Valores presentes no cluster:
    a. Chave é o valor de *hash* (*String hashCode() method*) de Valor;
    b. Valor contém a serialização em JSON de informação válida;
8. Operações a disponibilizar:
    a. *Read* de um objeto (valor ← *read*(chave)
    b. *Write* de um objeto: *write* (chave, valor)
9. Deverá existir um mecanismo de consenso baseado em comunicação por grupo, que coordene as operações em cada servidor;
10. Ações dos servidores:
    a. **Leitura de objeto:** Caso exista no servidor uma réplica local devolve imediatamente o valor do objeto. Caso não exista questiona o grupo pelo objeto e devolve o valor;
    b. **Escrita de objeto:** Caso o objeto não exista em nenhum servidor é criado nesse servidor. Caso já exista um objeto com a mesma chave devem ser invalidadas todas as réplicas e então é criado um objeto com o novo valor no servidor.
11. Cada servidor armazenas os objetos em memória e deve persistir os dados num repositório em memória secundária.
12. Um servidor que entre de novo no grupo ou que faça *startup* após falha deverá iniciar um processo de eleição que lhe garanta que unicamente um coordenador o ajuda a atualizar as eventuais réplicas de dados;

Arquitetura

Descrição

O projeto será constituído por 3 máquinas virtuais nas quais estarão presentes instâncias de servidores, podendo ser de três tipos:

- **Servidores *Followers*** – Servidores responsáveis pelas operações pedidas pelo cliente (leitura e escrita). Os dados estarão presentes numa estrutura de dados e um backup dos mesmos para um ficheiro em formato ".txt" ambos presentes no servidor. O Servidor Follower mediante necessidade e eleição poderá se tornar servidor monitor.
- **Servidores Monitores** – Servidores responsáveis pela resposta a operações a cliente (escrita) e coordenação da atualização dos servidores que se juntam ao grupo. Estes servidores têm as mesmas características que os Servidores Follower têm em conta a estrutura de dados e formatos de backup.
- **Serviço de Configuração** – Servidor responsável pelo mapeamento e atualização dos ip's e portos das instâncias presentes no cluster. Este servidor estará atento a todas as alterações das instâncias, atualizando o mapeamento sempre que exista uma alteração. Este servidor será também responsável pela resposta ao pedido inicial do cliente (*getClusterGroup()*).

Tendo em conta os requisitos optamos por instanciar os servidores Follower e Monitores em duas máquinas virtuais e dedicar uma máquina virtual só para o Serviço de Configuração.

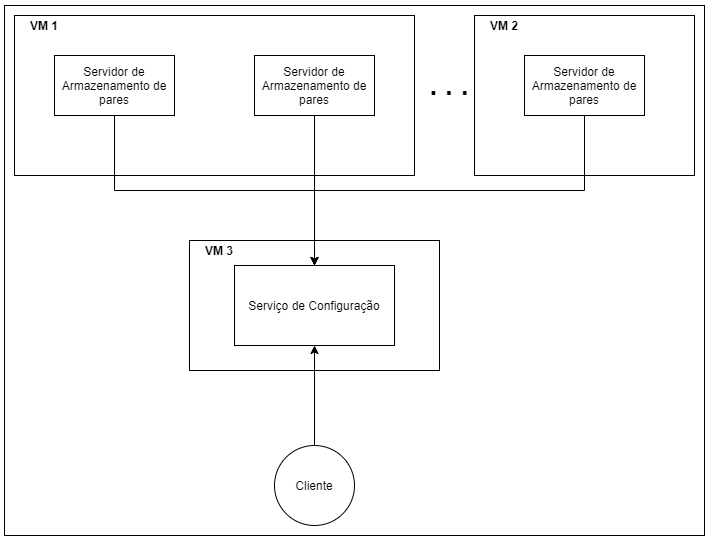

Figura 2 - Arquitetura

Funcionamento

No processo iniciação do cluster será realizado as seguintes ações:

- O serviço de configuração irá iniciar a monotorização das instâncias presentes no cluster, iniciando o mapeamento.
- Após adicionadas a um grupo a primeira inserida será eleita Servidor Monitor. Enviando essa informação para os novos membros que entrarem.

Considerando um funcionamento normal dos servidores, caso haja falha do servidor Monitor o primeiro Servidor Follower a detetar a falha irá informar todos os outros servidores do grupo que será o novo monitor através de uma mensagem *multicast*. Caso já exista algum outro servidor a fazer o mesmo pedido será desempatado analisando os *logs* (tempo de envio da mensagem). Os servidores que detetem a falha e recebam a mensagem do novo monitor passaram guardam o ip e caso recebam uma outra mensagem de eleição ignoram.

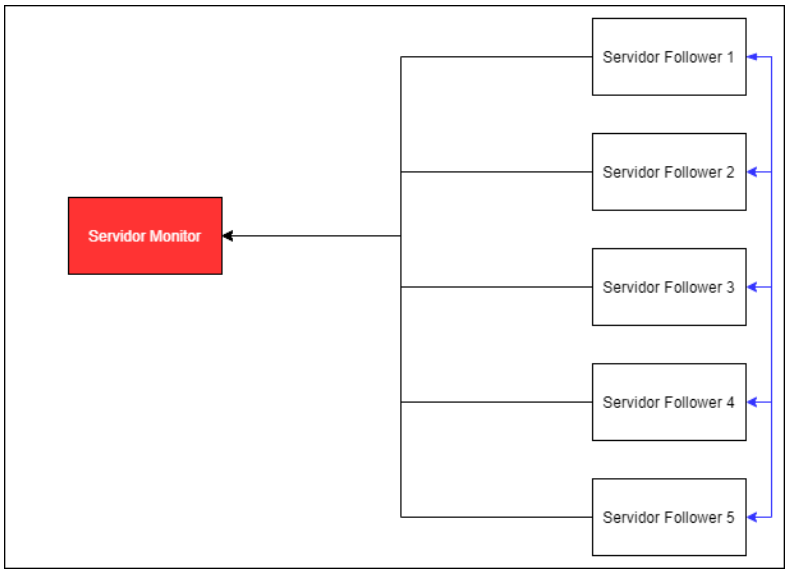

Figura 3 - Servidor Monitor Indisponível

Caso um Servidor Follower fique offline e volte a ficar online, questiona o Servidor Monitor sobre os objetos que possuí. Se esses objetos já estiverem mapeados noutros servidores, as réplicas são invalidadas.

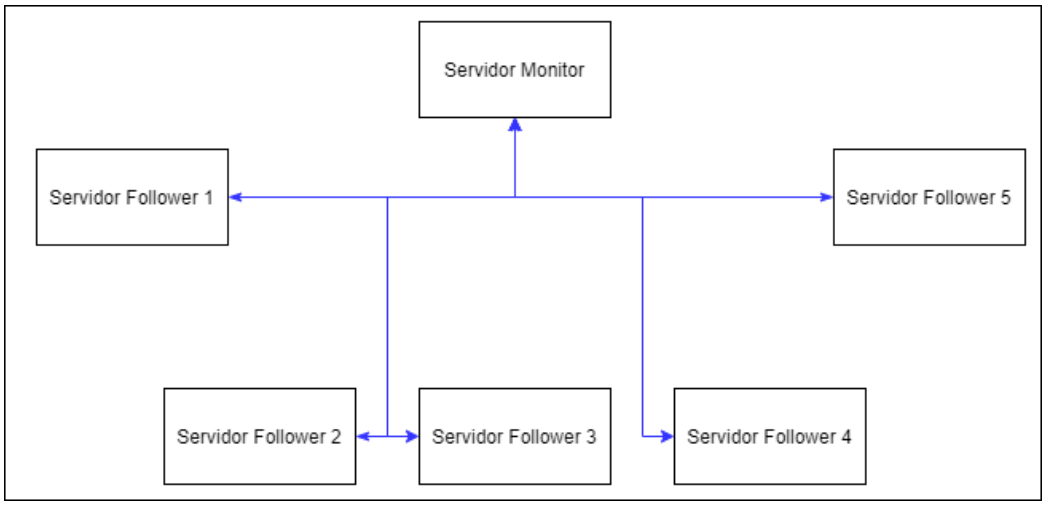

Figura 4 - Comunicação entre servidores

Implementação

Operações

Leitura de Dados

Cliente comunica com o Servidor Follower para aceder a um objeto. Se o Servidor Follower não tiver o objeto, questiona os restantes servidores se possuem esse objeto. O Servidor que possuir o objeto retorna o meu diretamente ao cliente.

```
public void read(Key request, StreamObserver<Value> responseObserver) {
    String val = db.get(request.getK());
    if (val == null) {
        //Key não existe no servidor
        try {
            SpreadMessage msg = new SpreadMessage();
            msg.setSafe();
            msg.addGroup("1");
            msg.setData("ping".getBytes());
            connection.multicast(msg);
            for (SpreadGroup member :
msg.getMembershipInfo().getMembers()) {
                String[] strSplit = member.toString().split(":");
                String follwrIP = strSplit[0];
                int followrPrt = Integer.parseInt(strSplit[1]);
                ManagedChannel configurationServiceChannel =
ManagedChannelBuilder
                        .forAddress(follwrIP, followrPrt)
                        .usePlaintext()
                        .build();

                ServerGrpc.ServerStub configurationStub =
ServerGrpc.newStub(configurationServiceChannel);
                configurationStub.readChk(request, responseObserver);
            }

        } catch (SpreadException e) {
            e.printStackTrace();
        }
    } else {
        //Key existe no servidor
        responseObserver.onNext(Value.newBuilder().setV(val).build());
        responseObserver.onCompleted();
    }
}
```

Escrita de Dados

Cliente comunica com o Servidor Follower emitindo a mensagem de escrita do valor 7 no objeto "a". É feita uma escrita no Servidor Follower do objeto "a" com o valor 7 e é pedido ao Servidor Monitor para invalidar as réplicas existentes. No final, o Servidor Monitor cria uma réplica num servidor aleatório para consistência de dados.

```
public void write(KeyValuePair request, StreamObserver<Void> responseObserver) {
    /*
      KeyValuePair: {
        k: {
           string k;
        }
        v: {
           string v;
        }
      }
     */
    StreamObserverInvalidateReplica rplcaStream = new
StreamObserverInvalidateReplica();
    if (isMonitor) {
        invalidateReplicas(request.getK(), rplcaStream);
    } else {
        try {
            SpreadMessage msg = new SpreadMessage();
            msg.setSafe();
            msg.addGroup("1");
            msg.setData("ping".getBytes());
            connection.multicast(msg);
            for (SpreadGroup member : msg.getMembershipInfo().getMembers()) {
                String[] strSplit = member.toString().split(":");
                String follwrIP = strSplit[0];
                int followrPrt = Integer.parseInt(strSplit[1]);
                //Create connection with Server Monitor
                ManagedChannel monitorServerChannel = ManagedChannelBuilder
                        .forAddress(follwrIP, followrPrt)
                        .usePlaintext()
                        .build();

                ServerGrpc.ServerStub monitorStub =
ServerGrpc.newStub(monitorServerChannel);
                //if connection fails, begin election process
                if (monitorServerChannel.getState(true) ==
ConnectivityState.TRANSIENT_FAILURE) {
                    electionProcess(Void.newBuilder().build(), new
StreamObserverGeneric() );

                } else {
                    monitorStub.invalidateReplicas(request.getK(), rplcaStream);
                    while (!rplcaStream.isCompleted) {
                        System.out.println("Waiting For Monitor to conclude
Invalidations...");
                        Thread.sleep(4000);
                    }
                }
            }

        } catch (SpreadException e) {
            e.printStackTrace();
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
    }
```

getClusterGroup

O serviço de configuração envia uma mensagem *multicast* para membros de cada grupo, e consoante os ip's obtidos é realizado o mapeamento das instâncias.

```
public void getClusterGroup(Void request, StreamObserver<ServerFollower>
responseObserver) {
        ArrayList<String> followers = new ArrayList<String>();


        if (connection == null) {
            try {
                connection = new SpreadConnection();
                connection.connect(InetAddress.getByName(daemonAddress),
daemonPort, user, false, true);
            } catch (Exception e) {
                e.printStackTrace();
            }
            //connection.disconnect();
        }

        if (spreadGroup == null) {
            spreadGroup = new SpreadGroup();
            try {
                spreadGroup.join(connection, "1"); //To join a specific group
(1) → Associa connection ao grupo
                SpreadMessage msg = new SpreadMessage();
                msg.setSafe();
                msg.addGroup("1");
                msg.setData("ping".getBytes());
                connection.multicast(msg);
                for (SpreadGroup member :
msg.getMembershipInfo().getMembers())
                    followers.add(member.toString());
                followers.remove(user);
            } catch (SpreadException e) {
                e.printStackTrace();
            }
        }
        followers.forEach( it -> {
            ServerFollower sv = ServerFollower.newBuilder().setIp(it).build();
            responseObserver.onNext(sv);
        });

        responseObserver.onCompleted();
    }
}
```

Configurações das Máquina Virtuais

```
Spread_Segment                  10.154.0.3:4803 {
        spreadNode1             10.154.0.3
}
Spread_Segment                  10.154.0.4:4803 {
        spreadNode2             10.154.0.4
}
Spread_Segment                  10.128.0.2:4803 {
        spreadNode3             10.128.0.2
}

# Descomentar Linux user e group para o spread
DaemonUser = spread
DaemonGroup = spread
#comentar DangerousMonitor = true
#DangerousMonitor = true
#descomentar
SocketPortReuse = AUTO
```

Figura 5 - Spread confg file "simple.spread.conf"

```
Conf_load_conf_file: using file: simple.spread.conf
Successfully configured Segment 0 [10.154.0.3:4803] with 1 procs:
                spreadNode1: 10.154.0.3
Successfully configured Segment 1 [10.154.0.4:4803] with 1 procs:
                spreadNode2: 10.154.0.4
Successfully configured Segment 2 [10.128.0.2:4803] with 1 procs:
                spreadNode3: 10.128.0.2
Set user name to 'spread'
Set group name to 'spread'
Setting SO_REUSEADDR to auto
Finished configuration file.
Hash value for this configuration is: 302201645
Conf_load_conf_file: My name: spreadNode1, id: 10.154.0.3, port: 4803
Membership id is ( 177864707, 1610663437)
--------------------
Configuration at spreadNode1 is:
Num Segments 3
        1       10.154.0.3      4803
                spreadNode1             10.154.0.3
        0       10.154.0.4      4803
        0       10.128.0.2      4803
====================
```

Figura 6 - Spread Return

Conclusão

Com este projeto foi possível ter *hands on* na criação de Sistemas de Computação distribuída.

Houve dificuldade, na perceção do modo de funcionamento do Sistema, bem como gerar uma arquitetura que correspondesse aos requisitos pretendidos mantendo um bom nível de consistências, velocidade de processamento e um bom nível de disponibilidade.

Através de alguns algoritmos de consenso disponibilizados durante a aula, foi possível haver uma discussão mais completa e concisa sobre o tipo de arquitetura a implementar, bem como um ideia geral dos aspetos positivos e negativos que a arquitetura poderia trazer.