17-10-2022	Computação Distribuída (Inverno 22/23)

Versão 2
Revisões a amarelo

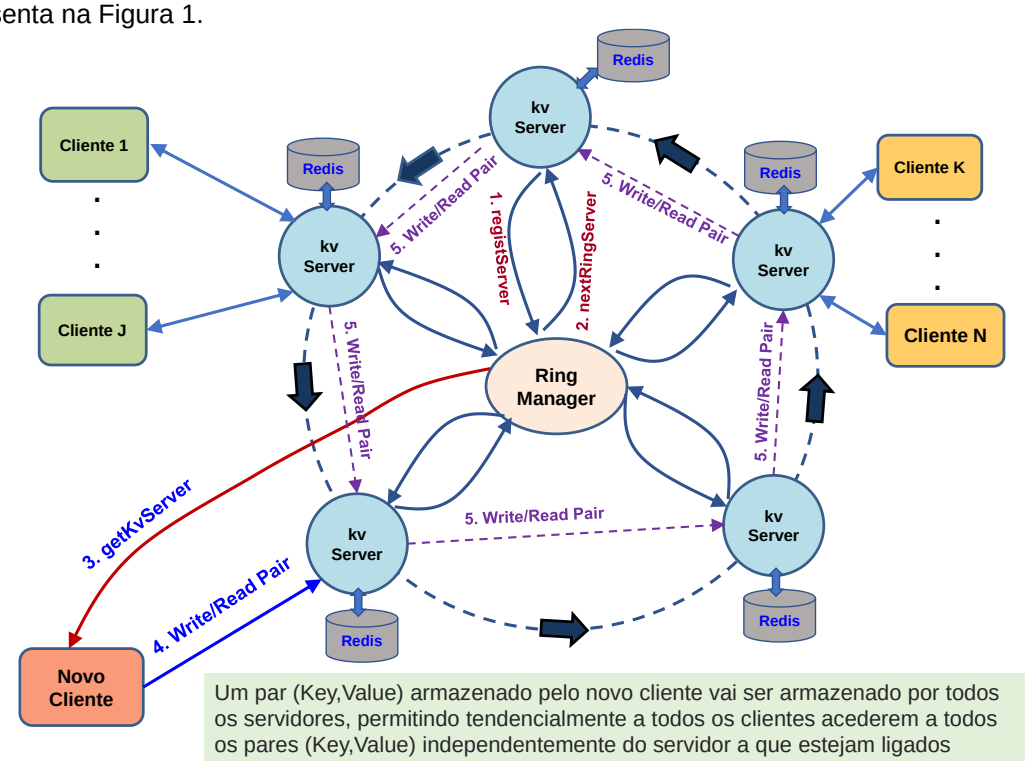
Trabalho Prático de Avaliação (TPA1)

Objetivos: Utilização do *middleware* gRPC na implementação de um sistema com múltiplos servidores ligados em anel para suportar múltiplas réplicas de dados e distribuição de carga nos pedidos dos múltiplos clientes. Utilização de containers para disponibilizar o armazenamento de dados (Key, Value)

Notas prévias:

- A aula de 20 e 27 de Outubro de 2022, na turma diurna e a aula de 21, 28 de Outubro de 2022, na turma noturna serão totalmente alocadas para realizarem o trabalho. No entanto, é pressuposto e faz parte dos ECTS da Unidade Curricular que têm de dedicar horas de trabalho fora das aulas para a realização do trabalho prático. Para eventual apoio e esclarecimento de dúvidas fora das aulas devem agendar com os professores o pedido de ajuda que poderá ser feito presencial ou remoto via Zoom. (nos *links* disponibilizados pelos respetivos professores);
- De acordo com as regras de avaliação definidas no slide 5 do conjunto *CD-01 Apresentação.pdf*, este trabalho tem um peso de 20% na avaliação final;
- A entrega será realizada em Moodle com um ficheiro Zip, incluindo os projetos desenvolvidos (src e pom.xml sem incluir os artefactos), bem como outros ficheiros que considerem pertinentes para a avaliação do trabalho. Por exemplo, podem juntar um ficheiro PDF ou txt, tipo readme.txt que explica os pressupostos que usaram bem como a forma de configurar e executar o sistema;
- Nas aulas de 10 de novembro de 2022 (turma diurna) e 11 de novembro de 2022 (turma noturna), cada grupo terá de apresentar e demonstrar, durante 10 a 15 minutos, para toda a turma, a funcionalidade do trabalho realizado;
- **A entrega no Moodle é até 8 de novembro de 2022 (23:59)**

Pretende-se o desenvolvimento de um sistema de distribuído com múltiplos servidores ligados em anel (***kvRing***), com réplicas de armazenamento de dados organizados em pares (*Key*, *Value*), para suportar equilíbrio de carga perante um elevado número de clientes conectados aos vários servidores do anel, como se apresenta na Figura 1.



Um par (Key,Value) armazenado pelo novo cliente vai ser armazenado por todos os servidores, permitindo tendencialmente a todos os clientes acederem a todos os pares (Key,Value) independentemente do servidor a que estejam ligados

Figura 1 - Diagrama geral do sistema ***kvRing***

Requisitos funcionais

- Existe um servidor central (*RingManager*) que faz a gestão de um anel de N servidores *KvServer*;
- Assume-se que o valor de N é conhecido inicialmente com valores 0 < N < 5;
- O servidor *RingManager* aceita o registo dos N servidores *kvServer*;
- Após o registo dos N *KvServer* o *RingManager* devolve a cada servidor *kvServer* que se registou qual é o próximo servidor *kvServer* no anel;
- O servidor *RingManager* permite aos clientes obterem a localização (IP, port) de um *kvServer* a que os clientes se irão ligar para armazenarem pares (*Key*,*Value*) ou dada uma chave obterem o valor do par associado;
- Para simplificar considere que tanto a chave como o valor são do tipo *String*, em que a chave pode eventualmente ser um *hash* a 256 bit e o valor uma string JSON que representa dados estruturados e complexos;
- Quando um servidor *kvServer* deteta que o seu sucessor no anel falhou (não responde) contata o *RingManager* para que este lhe indique qual é o novo sucessor, reconstituindo-se assim o anel;
- O servidor *RingManager* só disponibiliza acessos aos servidores *kvServer*, após a conclusão da configuração do anel que se processa da seguinte forma:
  - Cada um dos N servidores (*kvServer*) inicia-se com o registo no servidor central *RingManager*, indicando o seu *EndPoint* (IP, port) (ações 1 e 2);
  - Após todos os servidores *kvServer* se registarem, o *RingManager* devolve a cada servidor o seu sucessor no anel, garantindo que o último que se registou tem como sucessor o primeiro que se registou;
- Após a configuração do anel o *RingManager* aceita pedido dos múltiplos clientes (ação 3) para obterem o *EndPoint* de um servidor *kvServer*. A resposta do *RingManager* deve usar uma distribuição *Round Robin* pelos vários servidores *kvServer*;
- Após obterem do *RingManager* um servidor *kvServer*, os clientes iniciam as suas ações de armazenamento e consulta de pares (*Key*,*Value*) (ação 4);
- Para armazenar a sua réplica dos pares (*Key*, *Value*), cada *KvServer* tira partido de um serviço REDIS (ver anexo) alojado num *container* Docker na mesma máquina onde se executa o servidor *kvServer*;
- Quando um servidor *kvServer* recebe pedidos deve processá-los da seguinte forma:
  - **`WriteUpdate(key, value)`** – O par é inserido ou atualizado na réplica local e de seguida o pedido é encaminhado para o servidor sucessor no anel (ação 5), que procederá da mesma forma. Assim, independentemente do servidor a que um cliente está ligado, as várias réplicas dos vários servidores *kvServer* tenderão a estar idênticas. No entanto, note que estaremos perante um sistema eventualmente consistente na medida em que dois clientes ligados a servidores diferentes podem ver valores diferentes para a mesma chave;
  - **`Read(key) -> Result`** – Se o servidor *kvServer* tiver localmente o par retorna de imediato um *Result* com o valor. Se não existir a chave, o servidor faz circular um pedido no anel (ação 5) a requerer cópia do par. Se receber resposta a esse pedido com o par armazena-o localmente e devolve o valor em *Result*. Se o par não existir devolve essa informação ao cliente;
  - Para que possa ser detetado a fim do encaminhamento, cada pedido que circula no anel, deve ter a marca do servidor onde o pedido foi originalmente recebido a partir de um cliente.

Requisitos não funcionais

- Por questões de isolamento entre as partes (*Loose Coupling*) devem existir 4 contratos:
  1. Do servidor *kvServer* para o servidor *RingManager* para registo e obtenção do próximo no anel;
  2. Do Cliente para o servidor *RingManager* para obtenção do *EndPoint* de um servidor *kvServer*;
  3. Do servidor *kvServer* para outro servidor *kvServer* para comunicação no anel entre servidores;

4. Do cliente para o servidor *kvServer*

- Assim, cada servidor *kvServer* terá de disponibilizar dois serviços, um para acesso dos clientes e outro para acesso pelo servidor predecessor no anel.
- Da mesma forma, o servidor *RingManager* disponibiliza dois serviços, um para registo de servidores *kvServer* e outro para os clientes obterem os EndPoints dos servidores *kvServer*.
- Assume-se que o servidor *RingManager* nunca falha e que está localizado num *EndPoint (IP, port)* sempre igual e bem conhecido;
- Cada servidor *kvServer* pode estar em *Endpoints* da mesma VM ou de VM diferentes;
- A construção do protótipo de demonstração deve ter pelo menos 3 servidores *kvServer* em execução em pelo menos duas VM, com pelo menos um cliente em cada servidor;
- Assuma que o serviço REDIS de cada *KvServer* está alojado na mesma VM onde se executa o servidor;
- As várias instâncias da aplicação Cliente podem executar-se tanto nas máquinas locais dos elementos do grupo (computadores pessoais) como nas instâncias de VM onde se executam os vários servidores;
- Como melhoramento da solução inicial com número fixo de N *KvServer*, deve ser considerada a implementação ou pelo menos a discutida num ficheiro *read-me.txt* a possibilidade de introduzir novos servidores no anel, bem como as consequências e ações a realizar após a falha de um servidor *kvServer*:

Sugestões Gerais

- Qualquer questão ou dúvida sobre os requisitos deve ser discutida com o professor;
- Antes de começar a escrever código desenhe a arquitetura do sistema, os contratos dos serviços bem como os diagramas de interação mais importantes;
- Tenha em atenção o tratamento e propagação de exceções para assim o sistema ser mais fiável e permitir tratar algumas falhas;
- Quando tiver dúvidas sobre os requisitos, verifique no site *Moodle* se existem "*Frequently Asked Questions*" com esclarecimentos sobre o trabalho.

ANEXO

REDIS (https://redis.io)

REDIS (*REmote DIctionary Server*) é um sistema *open source*, para armazenamento de dados em memória, mas com possibilidade de persistência, que pode ser usado para bases de dados *NoSQL Key Value store*, *Cache, streaming engine*, *message broker, etc.*.

A forma mais simples e rápida de ter disponível um servidor REDIS é instanciar um container Docker:

a) Obter do *dockerhub* imagem *redis*
```
$docker pull redis
```
b) *Container* com armazenamento *In-memory* (sem persistência)
```
$docker run --name credis1 -p 6000:6379 -d redis
```
c) *Container* com armazenamento persistente
```
$ docker run --name redis2 -d redis redis-server --save 60 1 --loglevel warning
```
(https://hub.docker.com/_/redis) This one will save a snapshot of the DB every 60 seconds if at least 1 write operation was performed (it will also lead to more logs, so the `loglevel` option may be desirable). If persistence is enabled, data is stored in the `VOLUME /data`, which can be used with `--volumes-from some-volume-container` or `-v /docker/host/dir:/data`

Para desenvolver aplicações que acedem ao servidor REDIS podemos usar várias bibliotecas, mas no âmbito do trabalho recomenda-se o uso da seguinte dependência Maven:

```
<!-- https://mvnrepository.com/artifact/redis.clients/jedis -->
<dependency>
    <groupId>redis.clients</groupId>
    <artifactId>jedis</artifactId>
    <version>4.3.0</version>
</dependency>
```

A seguir apresenta-se um troço de programa Java que armazena e acede a pares (Key, Value)

```
static int REDIS_PORT=6379; // redis default port
public static void main(String[] args) {
  try {
     String IPSERVER="localhost";
     if (args.length > 0) IPSERVER=args[0];
     Jedis jedis = new Jedis(IPSERVER, REDIS_PORT);
     System.out.println("Redis is running on "+IPSERVER+" port="+REDIS_PORT+jedis.ping());
     Scanner input = new Scanner(System.in);
     String key;  String value;
     String key=read("What Key?",input);  String value=read("What Value?",input);
     jedis.set(key, value));  // store pair (K,V)

      key=read("Enter key to get value?", input);
      value = jedis.get(key);
      if (value == null)   System.out.println("Key " + key + " inexistent");
      else System.out.println("Value of " + key + ": " + value);

      // Hash key generation
      value = read("What value?", input);
      MessageDigest funcDigest = MessageDigest.getInstance("SHA-256");
      byte[] arr=funcDigest.digest(value.getBytes());
      key=Base64.getEncoder().withoutPadding().encodeToString(arr);
      System.out.println(jedis.set(key, value)+" key="+key);
  } catch (Exception ex) {  ex.printStackTrace();  }
}
```