10-10-2024	Computação Distribuída (Inverno 24/25)

Trabalho Prático de Avaliação (TPA1)

Objetivos: Utilização do *middleware* gRPC na implementação de um sistema com múltiplos servidores de suporte a balanceamento de carga na execução de pedidos por parte de múltiplos clientes. Cada pedido de um cliente pode envolver interação com outros servidores ou mesmo exigir processamento realizado em *containers* Docker lançado pelos servidores gRPC.

Notas prévias:

- Nas aulas de 17 e 31 de Outubro de 2024, na turma diurna e nas aulas de 15 e 29 de Outubro de 2024, na turma noturna as aulas serão dedicadas para realizarem e esclarecerem dúvidas sobre o trabalho. No entanto, é pressuposto e faz parte dos ECTS da Unidade Curricular que têm de dedicar tempo fora das aulas para a realização do trabalho. Para eventual apoio e esclarecimento de dúvidas fora do tempo de 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 no Moodle com um ficheiro Zip, incluindo os projetos desenvolvidos (*src, pom.xml, assembly.xml* sem incluir os artefactos, nomeadamente a diretoria *target*), bem como outros ficheiros que considerem pertinentes para a avaliação do trabalho. Por exemplo, podem juntar um ficheiro PDF ou tipo *readme.tx*t que explica os pressupostos que assumiram bem como a forma de configurar e executar o sistema;
- Nas aulas de 07 de novembro de 2024 (turma diurna) e 05 de novembro de 2024 (turma noturna), cada grupo terá de apresentar e demonstrar, durante 10 a 15 minutos, para toda a turma, a funcionalidade e operacionalidade do trabalho realizado;
- **A entrega no Moodle por cada grupo é até 02 de novembro de 2024 (23:59h)**

Pretende-se o desenvolvimento de um sistema distribuído (***PrimesRing***) onde existem múltiplos servidores para suportar distribuição de carga perante os múltiplos pedidos de múltiplos clientes, como se apresenta na Figura 1. Um pedido consiste em obter informação se um número do tipo *long* é ou não um número primo.

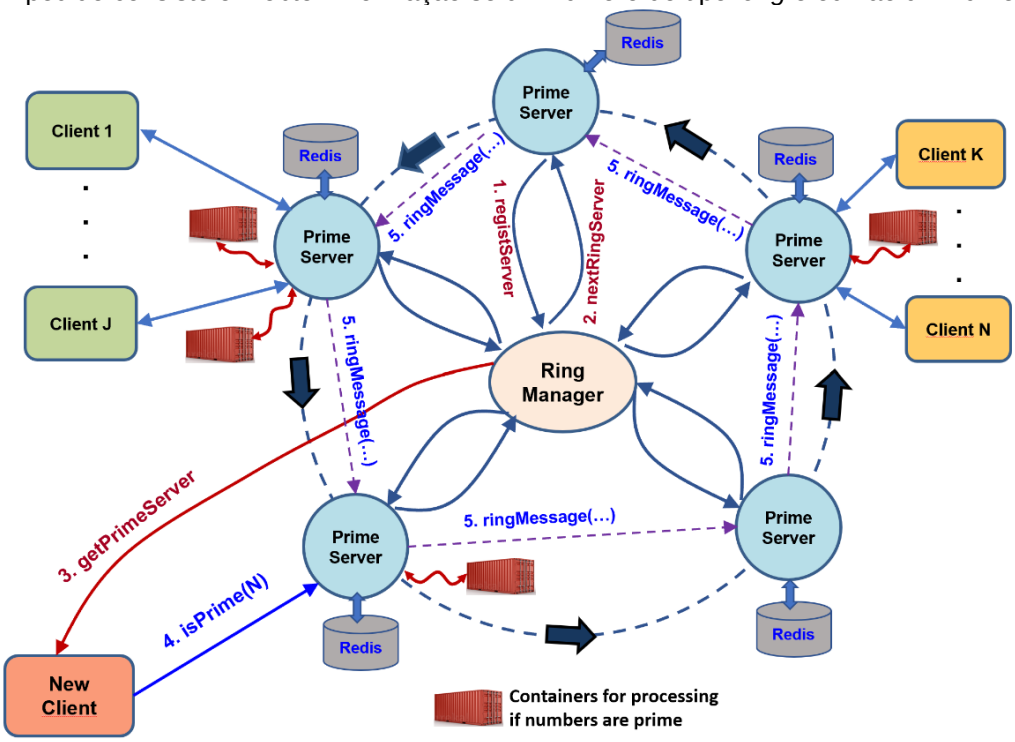

Figura 1 - Diagrama geral do sistema ***PrimesRing***

Requisitos funcionais

- Existe um servidor *RingManager* com localização (IP, porto) bem conhecida que disponibiliza dois contratos: i) Um contrato para os servidores (*PrimeServer*)) se registarem (***1.registServer***), indicando o seu IP e porto e para obterem o IP e porto do próximo servidor no anel (***2.nextRingserver***); ii) Um contrato para que os clientes possam obter o IP e porto de um servidor ao qual podem submeter pedidos (***3.getPrimeServer***);
- O servidor *RingManager* deve distribuir os clientes com equilíbrio pelos vários servidores do anel;
- Cada Servidor *PrimeServer* disponibiliza dois contratos: i) Um contrato para receber pedidos dos clientes para verificar se um número é primo (***4.isPrime(N)***); ii) Um contrato para receberem pedidos (**5.ringMessage()**) vindos do servidor antecessor no anel;
- A aplicação *Client* usa dois contratos: i) operação de acesso ao servidor *RingManager* para obter a localização de um dos servidores registados; ii) operações que permitam questionar um servidor se determinados números são ou não primos;
- Um servidor *PrimeServer* quando recebe o pedido de um cliente com um número, consulta um dicionário (*key*, *value*) existente num servidor REDIS alojado num docker *container* (no Anexo 1 ilustra-se como aceder ao dicionário para guardar/obter pares (*key*, *value*) indicando se um número é ou não primo) com o seguinte comportamento:
  - Se o dicionário já tiver informação sobre o par o servidor responde de imediato ao cliente;
  - Se o servidor não tiver essa informação envia uma mensagem no anel para pedir ao servidor seguinte;
  - Cada servidor no anel pode alterar a mensagem indicando que já conhece o resultado, Caso a mensagem já transporte a informação que um número é ou não primo o servidor deve atualizar o seu dicionário REDIS local;
  - Ao percorrer o anel a mensagem regressa ao servidor inicial com ou sem informação sobre o número ser primo ou não: i) com informação, resulta numa resposta imediata do servidor ao cliente; ii) sem informação provoca que o servidor terá de lançar um *container* para processar se o número é ou não primo.
- No Anexo 2 ilustra-se o uso de uma API que permite lançar *containers* Docker dinamicamente, a partir de qualquer aplicação Java;
- O cliente recebe sempre uma resposta *true* ou *false* de forma transparente e independente das interações que o servidor *PrimeServer* teve para obter a informação se o número é primo;

Requisitos não funcionais

- Por questões de isolamento entre as partes (*Loose Coupling*) devem existir 4 contratos:
  1. Dos servidores *PrimeServer* para o *RingManager* para suportar o registo dos múltiplos servidores;
  2. Da aplicação *Client* para o servidor *RingManager* para obtenção do *EndPoint(IP, porto)* de um servidor *PrimeServer* para que possam ser submetidos pedidos;
  3. Da aplicação *Client* para o servidor *PrimeServer* para submeter pedidos;
  4. De encaminhamento de mensagens no anel entre servidores *PrimeServer*. Note que um servidor *PrimeServer* quando recebe uma mensagem limita-se a atualizar a mensagem se a informação do número já for conhecida, reenviando-a para o servidor *PrimeServer* seguinte. Assim recomenda-se que entre dois servidores *PrimeServer* contíguos no anel exista um *stream* de cliente para facilitar o processo de encaminhamento de mensagens.
- Assume-se que o servidor *RingManager* nunca falha e que está localizado num *EndPoint (IP, port)* sempre igual e bem conhecido;
- Para alojamento do sistema, cada VM pode disponibilizar um ou mais servidores *PrimeServer*;
- A construção do protótipo de demonstração deve ter pelo menos 3 servidores *PrimeServer* em execução em pelo menos duas VM, com pelo menos uma aplicação *Client* conectada a cada servidor;

- As várias instâncias da aplicação *Client* podem executar-se tanto nas máquinas locais dos elementos do grupo (computadores pessoais) como nas instâncias de VM onde se executam os vários servidores;
- Deve ser considerada na implementação e na demonstração final que a qualquer momento podem ser adicionados dinamicamente novos servidores melhorando assim o balanceamento de carga.

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 *protobuf* 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.

10-10-2024	Computação Distribuída (Inverno 24/25)

ANEXOS

Anexo 1: Comando Docker para lançar um servidor REDIS e troços de código Java para validar se um número é ou não primo e aceder ao servidor REDIS para guardar e obter pares (*key*, *value*), em que *key* é uma *string* com o valor do número e *value* é uma *string* com o resultado *true* ou *false*;

```
docker run -d --name PrimeServerRedis -p 6000:6379 redis
// o porto 6000 será o porto do Host para aceder ao dicionário REDIS
```

```
long numero = Long.parseLong(args[0]);
String redisAddress = args[1];
int redisport=Integer.parseInt(args[2]);
Jedis jedis = new Jedis(redisAddress, redisport);
if (isPrime(numero))
   jedis.set(numero+"","true");
else
   jedis.set(numero+"","false");
System.out.println(jedis.get(numero+""));

// ex: the number 999998727899999 is prime
boolean isPrime(long number) {
   if (number <= 1L) return false;
   if (number == 2L || number == 3L) return true;
   if (number % 2L == 0) return false;
   simulateExecutionTime();
   for (long i=3; i <= Math.sqrt(number); i+=2) {
      if (number % i == 0) return false;
   }
   return true;
}

void simulateExecutionTime() {
   try {
      // simulate processing time between 200ms and 3s
      Thread.sleep(new Random().nextInt(9800) + 200);
   } catch (InterruptedException e) {
      e.printStackTrace();
   }
}
```

```
<dependency>
   <groupId>redis.clients</groupId>
   <artifactId>jedis</artifactId>
   <version>5.2.0</version>
</dependency>


// veja também as funcões:
value = jedis.get(key);
// apaga todos os pares
jedis.flushDB();
// obter todas as keys
Set<String> allKeys=jedis.keys("*");
```

Adaptando este código numa aplicação Java e criando uma imagem Docker de nome *isPrime* é possível lançar um container que insere no dicionário REDIS o par (“127”, ”true”), indicando que o número 127 é primo, usando o seguinte comando:

```
docker run -d isPrime 127 HOST_IP 6000
```

Anexo 2: A biblioteca *docker-java* que encapsula a REST API do *runtime* do Docker permite a qualquer aplicação Java executar comandos *docker*, nomeadamente lançar e eliminar *containers,* bem como obter o estado `{running, exited, …}` de execução de um *container*. De seguida apresenta-se o código java de uma aplicação que permite dinamicamente lançar, verificar o estado e remover containers, equivalentes ao comando:

```
docker run -d --name ct127 isPrime 127 HOST_IP 6000
```

```
public static void main(String[] args) {
    // arg0 : windows: tcp://localhost:2375  linux: unix:///var/run/docker.sock
    // arg1 : container name
    // arg2 : image name
    // arg3 : number to calculate if it is prime
    // arg4 : Redis Host IP
    //arg 5: Redis port on host
    try {
      String HOST_URI = args[0];   String containerName = args[1]; String imageName=args[2];
      long number=Long.parseLong(args[3]);
      String redisHostIP=args[4];
      int redisHostport=Integer.parseInt(args[5]);
      List<String> command=new ArrayList<>();
      for (int i=3; i < args.length; i++) command.add(args[i]);

      DockerClient dockerclient = DockerClientBuilder
            .getInstance()
            .withDockerHttpClient(
                  new ApacheDockerHttpClient.Builder()
                        .dockerHost(URI.create(HOST_URI)).build()
            )
            .build();
      CreateContainerResponse containerResponse = dockerclient
            .createContainerCmd(imageName)
            .withName(containerName)
            .withCmd(command)
            .exec();
      System.out.println("ID:" + containerResponse.getId());
      dockerclient.startContainerCmd(containerResponse.getId()).exec();
      for(;;) {
        InspectContainerResponse inspResp = dockerclient
              .inspectContainerCmd(containerName).exec();
        System.out.println("Container Status: " + inspResp.getState().getStatus());
        if (inspResp.getState().getStatus().equals("exited")) break;
        Thread.sleep(1*1000);
      }
      // if container is running
      // dockerclient.killContainerCmd(containerName).exec();
      // remove container
      dockerclient.removeContainerCmd(containerName).exec();
    } catch (Exception ex) {
      ex.printStackTrace();
    }
  }
```

A aplicação anterior tem as seguintes dependências Maven:

```
<dependency>
    <groupId>com.github.docker-java</groupId>
    <artifactId>docker-java</artifactId>
    <version>3.4.0</version>
</dependency>
<dependency>
    <groupId>com.github.docker-java</groupId>
    <artifactId>docker-java-transport-httpclient5</artifactId>
    <version>3.4.0</version>
</dependency>
```