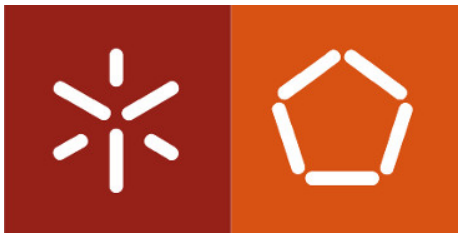
Fundamentos dos Sistemas Distribuídos

Grupo nr.	4
a84464	Válter Carvalho
a84577	José Pedro Silva
a84783	Pedro Rodrigues
a85006	Hugo Cardoso



Mestrado Integrado em Engenharia Informática
Universidade do Minho

Conteúdo

1 Introdução

No âmbito da Unidade Curricular Fundamentos de Sistemas Distribuídos(FSD), foi-nos proposto o desenvolvimento de um sistema distribuído de armazenamento em memória de pares de chave-valor. Este sistema deve admitir que o conjunto de servidores é fixo, previamente conhecido e não há falhas. O trabalho tem como objetivo apresentar uma interface para o programador de aplicações(API) com as seguintes operações:

- put
- get

Este sistema deve permitir concorrência garantindo ao mesmo tempo a consistência de dados.

Desde o início o nosso objetivo foi apresentar uma proposta modular e adequada à utilização em grande escala sem limitações artificiais.

2 Especificação do sistema

Primeiramente optamos por uma solução mais simples e à medida que íamos alcançando certos objetivos, aumentávamos a complexidade da nossa solução do problema. Inicialmente, o primeiro modelo consistia em suportar apenas um cliente e um servidor centralizado. Isto permitiu-nos agilizar os tipos de comunicação necessários para haver interação entre cliente e servidor e vice-versa.

Numa fase seguinte, adicionamos múltiplos servidores, tornando assim o sistema distribuído. Neste caso, tínhamos agora a responsabilidade de fazer a correta distribuição da carga pelos servidores. Posto isto, decidimos então que um cliente envia pedidos aos servidores de acordo com a sua função de *hashing*. Esta função consiste em fazer o resto da divisão da chave do pedido pelo número de servidores do sistema.

Após termos um cliente a comunicar com vários servidores, reparamos que era necessário etiquetar as mensagens para que estas fossem enviadas e recebidas pela ordem correta e não apenas pela ordem de chegada.

Como o sistema deve garantir a consistência dos dados foi necessário implementar um *sistema mutex distribuído* atribuindo aos clientes acesso à zona crítica apenas quando era possível respeitando sempre a ordem dos pedidos. Para isso utilizamos o algoritmo do coordenador centralizado lecionado nas aulas.

3 Arquitetura do sistema

A arquitetura do nosso sistema é composta por:

- Múltiplos clientes
- Múltiplos servidores
- Um coordenador centralizado
- Sockets assíncronos

Na figura 1 podemos observar uma representação hipotética do nosso sistema com 3 servidores participantes no sistema distribuído, 2 clientes a comunicarem com o coordenador e posteriormente com os servidores, e por fim, um coordenador responsável por garantir ordem nos acessos às zonas críticas do sistema.

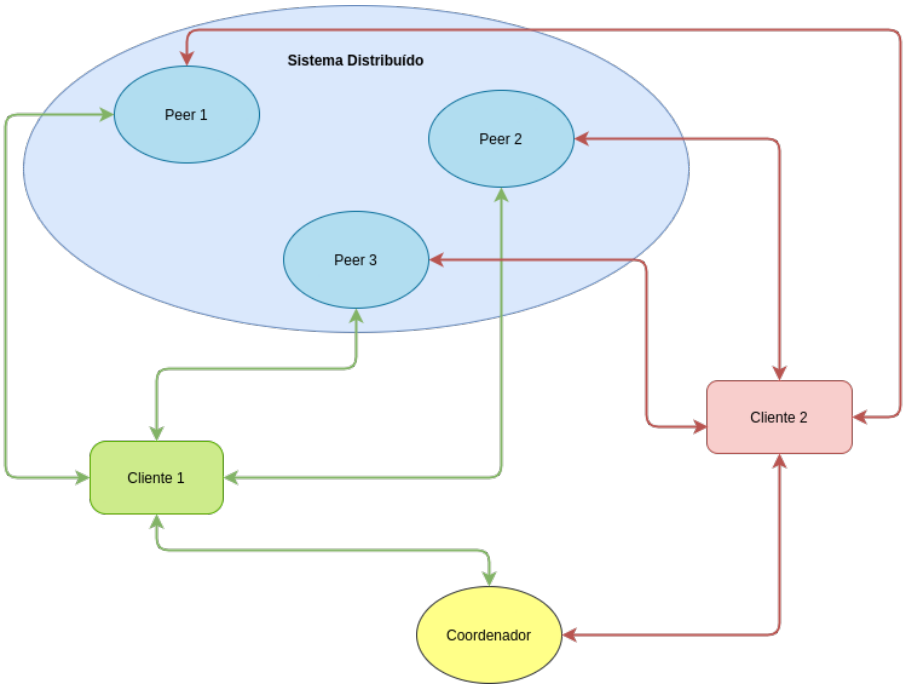


Figura 1: Arquitetura do sistema

4 Fluxo de um pedido

O fluxo de um pedido do cliente ao sistema deve basear-se no pressuposto de, antes de realizar um pedido ao sistema, o cliente deve enviar um pedido de acesso à zona crítica ao coordenador.

Figura 2: Fluxo de um pedido de lock ao coordenador.

Logo que seja possível atribuir o acesso à zona crítica ao cliente, o coordenador envia uma mensagem de confirmação para tal acesso.

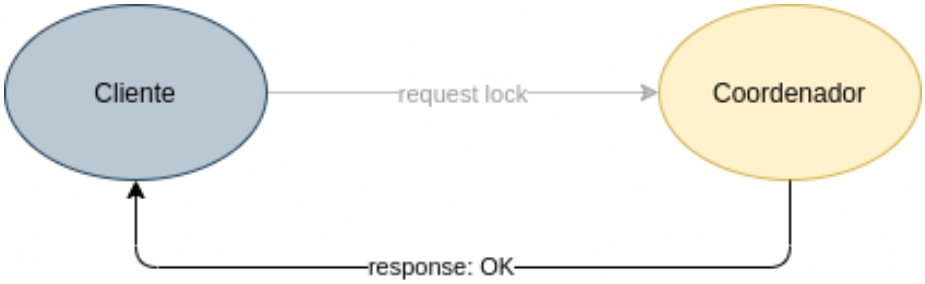


Figura 3: Fluxo de resposta do coordenador ao cliente.

Após a receção da mensagem de confirmação por parte do coordenador o cliente pode enviar pedidos ao sistema, acabando eventualmente, por receber as respostas.

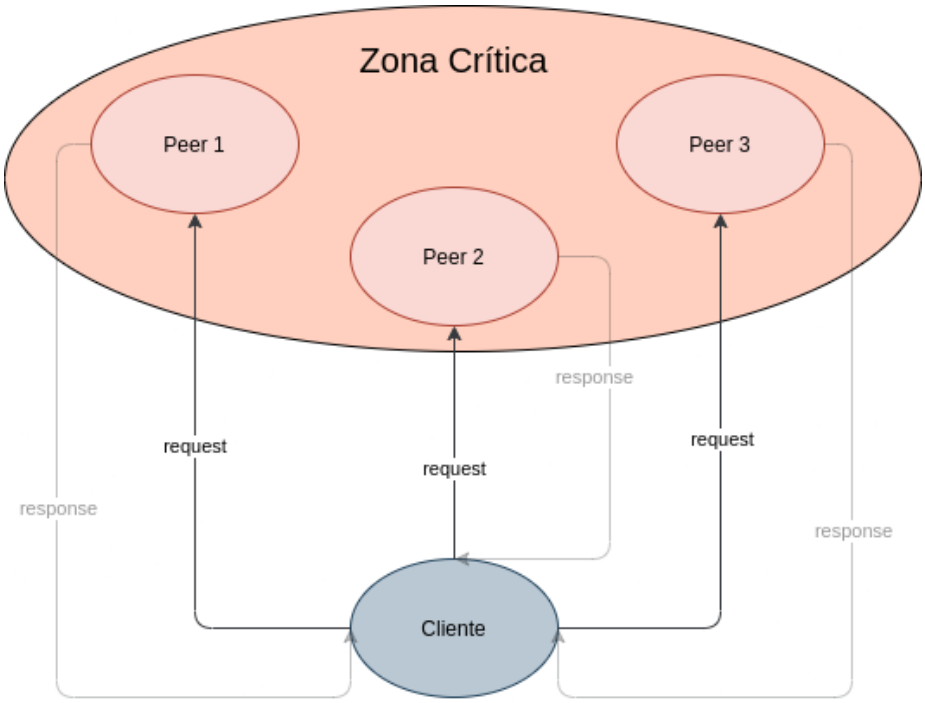


Figura 4: Fluxo de um pedido do cliente para o sistema.

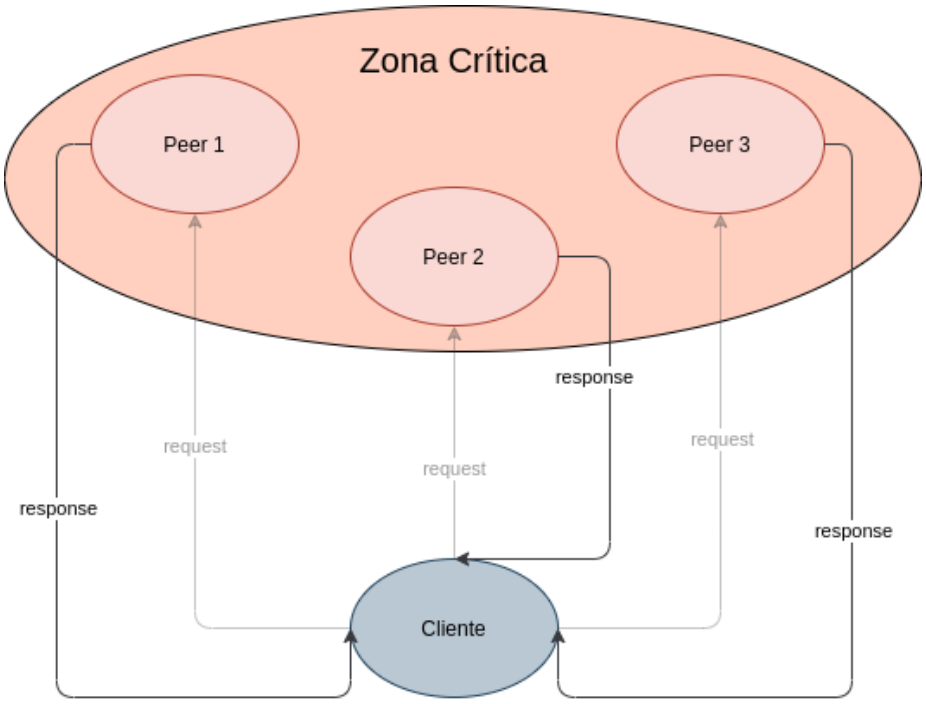


Figura 5: Fluxo de resposta do sistema para o cliente.

Aquando a receção de todas as mensagens do pedido, o cliente envia uma mensagem ao coordenador a avisar que já não necessita de continuar com acesso à zona crítica.

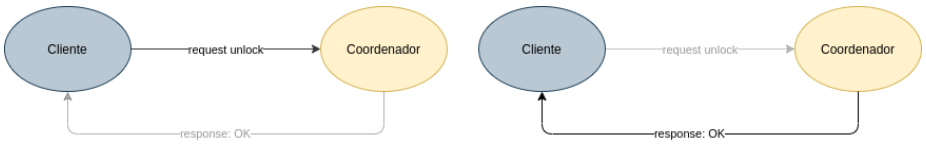


Figura 6: Fluxo do processo de unlock.

5 Funcionamento geral

5.1 Coordenador

O coordenador é responsável pela gestão do acesso à zona crítica do sistema. Após a receção de uma mensagem de lock, o coordenador deve verificar se pode fornecer o acesso a este cliente ou não. Em caso afirmativo, responde ao cliente com uma mensagem de confirmação, registando o seu tipo de acesso ao sistema. Em caso contrário, coloca o pedido numa fila de espera.

Após a receção de uma mensagem de unlock, é confirmado ao cliente o unlock e é enviada uma mensagem de confirmação de acesso ao próximo cliente da fila de espera. No caso de este pedido ser do tipo *get*, uma vez que o *get* não irá alterar o conteúdo dos dados no sistema, é possível dar acesso a outros pedidos *get* que estejam na fila de espera até que se encontre um *put*. Por outro lado, quando o pedido é do tipo *put*, apenas podemos deixar este cliente aceder à região crítica do sistema, uma vez que os dados vão ser alterados.

Em suma, o coordenador é responsável por manter no máximo um pedido *put* a trabalhar no sistema ou vários pedidos *get*, mas nunca *put*s e *get*s em simultâneo.

5.2 Servidor

O servidor é parte integrante do sistema distribuído contendo em memória todos os pares chave-valor inseridos pelos clientes. Este serviço está constantemente à escuta de pedidos e responde sempre com o resultado de aplicar a operação requisitada.

À primeira vista é intuitiva a necessidade da utilização de locks para garantir consistência dos dados devido à concorrência do sistema, no entanto, com a utilização de um mutex distribuído, é sempre garantido que todos os pedidos recebidos podem ser completados/tratados sem esse tipo de verificação.

Por outras palavras, o servidor apenas lê pedidos de clientes, computa esses pedidos e, por fim, responde ao mesmo.

5.3 Cliente

O cliente antes de realizar qualquer tipo de operação deve conectar-se ao sistema, conhecendo de antemão o coordenador e os peers do sistema. Ao conectar-se, é atribuído um identificador único a cada um dos peers fundamental para a função de *hashing*, responsável por distribuir a carga de maneira uniforme.

Sempre que um cliente realiza uma operação, este necessita de adquirir acesso à zona crítica através do processo explicado anteriormente. Quando o acesso for cedido, o cliente deve dividir o seu pedido pelos diversos *peer*s, sendo a divisão feita com recurso à função de *hashing*. Esta função consiste em, para cada chave, o pedido é feito ao *peer* que contém o identificador igual ao resultado do resto da divisão da chave pelo número de *peer*s.

Todas as operações que o cliente pode realizar utilizam métodos assíncronos devolvendo assim um **CompletableFuture**. Isto leva a que o sistema não seja bloqueante e possam ser realizadas operações em simultâneo.

6 Exemplo de utilização

O nosso sistema oferece três tipos de entidade diferentes, nomeadamente, **peer**, **Coodenador** e **Cliente**. O coordenador e os peers devem ser inicializa-

dos com a sua porta como argumento e os clientes com a porta do coordenador como primeiro argumento e as portas dos peers do sistema.

O cliente possui uma API para programadores que disponibiliza as seguintes operações:

```
CompletableFuture<Void> connect(String[] ports);
CompletableFuture<Void> put(Map<Long, byte[]> keys);
CompletableFuture<Map<Long, byte[]>> get(Collection<Long> keys);
```

Exemplo de utilização de API

```
public static void example(String[] args) throws InterruptedException {
        Client c = new Client();
        c.connect(args).thenRunAsync(() -> {
            Map<Long, byte[]> map = new HashMap<>();
            Collection<Long> list = new ArrayList<>();
            for(int i=0;i<10;i++){
                map.put((long)i, Tools.toByteArray("value"+i));
            }
            for(int i=0;i<5;i++){
                list.add((long)i);
            }

            CompletableFuture<Void> p = c.put(map);
            CompletableFuture<Map<Long, byte[]>> g = c.get(list);
        });
        TimeUnit.DAYS.sleep(1);
    }
```

7 Maiores desafios

Ao longo do desenvolvimento do projeto foram surgindo vários desafios e barreiras a ultrapassar. O obstáculo mais relevante a ser destacado, na nossa opinião é, a forma de lidar com a concorrência na receção e no envio de pedidos. Numa primeira implementação, quando sobrecarregávamos o sistema obtivemos algumas exceções do tipo *ReadPendingException* e *WritePendingException*. Após uma pesquisa à cerca destas exceções, reparamos que havia mais do que um processo a tentar ler do/escrever no socket. De modo a evitar tal coisa, decidimos alterar a nossa implementação. Para ler, um processo em vez de ler diretamente do socket, passa agora a ler de um Map onde se encontram todas as mensagens relativas aos pedidos recebidos. Caso a mensagem ainda não esteja no Map, é retornado um *CompletableFuture* que irá ser completado quando a mensagem em questão for lida. Para isto ser possível é necessário haver um e um só processo que lida com as leituras do socket e posteriormente coloca-as no Map.

Para o envio de mensagens o processo é análogo, ou seja, sempre que um processo queira enviar um mensagem, escreve a mesma num Map e esta será posteriormente enviada. Tal como acontece nas leituras, existirá apenas um processo responsável por escrever no socket as mensagens colocadas no Map.

De modo a que as operações dos Maps não fossem bloqueantes, foi criada a classe **AsyncMap**, que escreve e lê de um Map de forma assíncrona.

8 Melhorias futuras

Apesar de considerarmos que o nosso trabalho cumpre de forma distintiva os requisitos propostos, julgamos que, podem sempre haver melhorias. Após uma análise detalhada ao nosso projeto, concluímos que as melhorias que mais fazem sentido de serem aplicadas no futuro seriam, o mutex distribuído ser aplicado a uma determinada zona do sistema e não ao sistema inteiro o que poderia ser atingível caso, na mensagem ao coordenado, adicionássemos a zona crítica a ser utilizada; descentralizar o coordenador seria também uma melhoria e poderíamos alcançar este objetivo através caso organizássemos os peers em forma de anel de maneira a que apenas o peer que tivesse o *testemunho* pudesse aceitar pedidos do cliente. Por fim, achamos que podemos também melhorar a função de *hashing*, pois da maneira que se encontra implementada, não assegura load-balancing.