Universidade do Minho
Escola de Engenharia
Mestrado Integrado em Engenharia Informática

SISTEMAS DISTIBUÍDOS

Alocação de servidores na nuvem

Filipa Correia Parente(a82145)

José André Martins Pereira (a82880)

Rafaela Maria Soares da Silva(a79034)

Ricardo André Gomes Petronilho(a81744)

6 de Janeiro, 2019

Índice

Introdução

De forma a abordar a concorrência em Sistemas Distribuídos, no âmbito da Unidade Curricular de Sistemas Distribuídos em Mestrado Integrado em Engenharia Informática, foi proposta a elaboração de um sistema capaz de gerir reservas de servidores, em linguagem JAVA.

Assim, no desenvolvimento do trabalho, em praticamente todas as classes, usamos o two-phase locking como estratégia de exclusão mútua, com o objetivo de redução do tempo crítico e aumento da eficiência da aplicação.

Desta forma, garante-se que antes de bloquear um objeto em questão para o manipular, a estrutura onde o mesmo se encontra é bloqueada e, portanto, imutável. Após o bloqueio do objeto que se vai manipular, a estrutura de dados, onde o mesmo se encontra é desbloqueada, sendo que no final das alterações do objeto este também o é.

Sendo assim, o funcionamento normal da aplicação deste projeto é a existência de um servidor que aceita conexões de clientes, criando assim uma Thread para cada um, permitindo deste modo que vários estejam a usar o servidor simultaneamente.

O uso de *two fase locking*, como já foi referido acima, permite que os clientes possam aceder aos dados de forma sincronizada. Isto é, dois clientes diferentes acederem à mesma estrutura, mas de forma ordenada, ou seja, no momento em que um deles vai à estrutura, verifica se a mesma está bloqueada para outro. Caso não esteja, o mesmo bloqueia toda a estrutura para ele, e, após isto estar garantido, bloqueia apenas os objetos que vai manipular. Após esta certeza, liberta toda a estrutura, para que os outros clientes possam aceder à mesma e , no final das alterações dos objetos que bloqueou, também os vai libertar.

Classes

Cliente

A classe cliente é responsável por armazenar toda a informação acerca do cliente (email, password, saldo e identificadores das suas reservas).

Após alguns testes feitos durante a realização do projeto, decidiu-se utilizar um Set<String> para obter todos os identificadores das reservas associadas ao cliente em questão. Desta forma, quando necessitamos de saber as reservas que lhe estão associadas, apenas bloqueamos o Cliente em questão, reduzindo drasticamente a secção critica no acesso às reservas de cada cliente, como poderemos observar mais adiante.

Uma vez que vai ser utilizada exclusão mútua, esta classe contém um *ReentrantLock*. No entanto, a exclusão mútua, por ser *two-phase locking*, é assegurada apenas na classe que contém o conjunto de clientes, neste caso na GS.

O cliente é identificado pelo seu email, que por este motivo tem de ser único.

Produto

A classe Produto é responsável por representar um tipo de servidor para aluguer, contendo a informação do seu tipo, nome, e a sua ocupação, isto é, como existe um teto máximo de servidores por tipo, é necessário saber quantos estão a ser utilizados de momento.

Com a finalidade de implementar "two fase locking", para redução do tempo crítico, definiu-se um *Lock* para cada Produto.

Por fim, e não menos importante, para aumentar a eficiência da nossa aplicação, a classe Produto contém uma *Condition*, que permite adormecer *Thread do cliente q*uando todos os servidores estão ocupados, até que seja liberte algum.

No entanto, quando um Cliente não consegue alugar um servidor de um determinado tipo, tem a prioridade de ficar com algum desse tipo, que tenha sido atribuído em leilão.

Reserva

A classe Reserva tem a incumbência de conter toda a informação relativa a uma solicitação de um produto. Esta informação consiste no identificador da reserva, no tipo de reserva (a pedido ou em leilão), identificador do cliente, identificador do servidor, preço do produto à hora e da data em que foi feita a reserva, assim como da hora em que foi cessada a mesma.

Para além disso, tem a tarefa de auxiliar a assegurar que a ordem da atualização do saldo do cliente é sempre a mesma, através da implementação da interface Comparable<>. Desta forma, prevenimos *deadlock.*

Esta classe está encarregue de contabilizar o custo incorrido numa determinada reserva, através do método total.

Leilao

Esta classe contém todas as informações necessárias relativas ao leilão, tais como o seu início e fim, o nome do servidor em questão, o valor inicial estipulado, a melhor proposta e o id do cliente.

Esta tem o método responsável por atualizar a melhor proposta do respetivo leilão.

GS

A classe GS gere todos os clientes, produtos, reservas e leilões do sistema.

Para além disso, gere também os <<ultimosLeiloes>>, no intuito de não ser necessário percorrer a estrutura das reservas quando é feito uma reserva a pedido (visto que esta tem prioridade sobre o leilão, é uma alternativa que beneficia imenso o desempenho do sistema, visto que não será necessário bloquear a estrutura das reservas durante muito tempo).

Denominou-se como ultimosLeiloes a estrutura Map<String, List<String>>, uma vez que o List será FIFO, de forma a acedermos sempre à reserva a pedido o produto associado ao leilão que foi criado há mais tempo.

Esta classe é responsável pela autenticação dos clientes através do método, pela consulta do valor em dívida, pela reserva a pedido de um produto, pela proposta em leilão de um produto e pela libertação da reserva. Estas funcionalidades são possíveis através dos seguintes métodos:

- autenticar(String email, String password) throws ClienteNaoExisteException
- consulta(String email) throws ClienteNaoExisteException
- reservarPedido(String chaveProduto, String chaveCliente) throws ProdutoNaoExisteException, ReservaExisteException
- PropostaLeilao(String idCliente, String chaveProduto, float valor) throws ProdutoNaoExisteException, LeilaoTerminouException
- libertarReserva(String chaveReserva, String idArg) throws ReservaNaoExisteException, ProdutoNaoExisteException, ClienteNaoExisteException, ReservaNaoCorrespondente

É de realçar a relevância dos métodos reservarPedido, propostaLeilao e libertarReserva, uma vez que estes são os responsáveis pela gestão das reservas.

O método reservarPedido verifica primeiramente se o produto existe. Caso exista, procede à verificação da existência da reserva. Caso não exista, continua o processo normalmente. A reserva é criada de seguida, de forma a desbloquear a estrutura mais cedo. É, assim, associado ao cliente a reserva.

Se a quantidade de produtos ocupados do tipo de produto é o número máximo possível, o cliente terá prioridade caso haja um leilão corresponde ao tipo de produto que deseja (ficando com a reserva desse produto e sendo cancelada a reserva associada a esse leilão) ou então terá de esperar que um produto desse tipo esteja disponível.

O método propostaLeilao verifica primeiramente se existe um leilão associado ao tipo de produto pretendido. Caso não exista, mas o produto desejado sim, procede à criação de um leilão, através de uma *thread* (implementada na classe GestorLeilao) e à proposta do cliente. Caso o leilão já exista, procede à proposta do cliente.

O método libertarReserva, inicialmente, verifica se a reserva existe e se está associada ao cliente que tem a intenção de libertar. Se sim, verifica o tipo de reserva em questão. Caso seja uma reserva leilão, é ainda necessário remover a reserva da estrutura ultimosLeiloes, para além de, nos dois casos, ser necessário remover a reserva e libertar o produto.

Como todos os sistemas de administração, esta classe tem de conseguir adicionar ou remover clientes e produtos. No entanto, para o contexto do nosso programa apenas necessitamos de adicionar:

- criarCliente(String email, String password) throws ClienteExisteException
- criarProduto(String nome, float preco)

Para além destes métodos, no intento de facilitar a elaboração da interface, esta classe tem também:

- getTodosProdutos()
- getReservasCliente(String idCliente)

Esta classe está preparada para exceções, como se pode observar na API mostrada anteriormente.

Todos os métodos desta classe têm implementados o *two-phase locking*, como se pode observar na seguinte figura:

```
public boolean autenticar(String email, String password, PrintWriter pw) throws ClienteNaoExisteException {
    this.lclientes.lock(); // lock() da estrutura
    if (!this.clientes.containsKey(email)) {
        this.lclientes.unlock(); // unlock() da estrutura caso seja excepção
        throw new ClienteNaoExisteException(email);
    }
    Cliente cliente = this.clientes.get(email);
    cliente.l.lock(); // lock() do cliente
    this.lclientes.unlock(); // unlock() da estrutura
    boolean res = cliente.password.equals(password);
    if (res) cliente.pw = pw; // atualizamos o PrintWriter mais recente do cliente
    cliente.l.unlock(); // unlock() do cliente
    return res;
}
```

Figura 1 Two-phase Locking implementado no método autenticar() na classe GS

A vermelho verifica-se o lock() e unlock() da estrutura que contém os clientes. A verde o lock() e unlock() do cliente em questão.

Este método cumpre as duas regras de ***two-phase locking***, isto é:

- os dados de cada objeto são manipulados (leitura ou escrita) dentro dos respetivos lock()'s associados (região crítica).
- todos os lock()'s são evocados antes de todos os unlock()'s.

De referir que nesta classe, precisamente nos métodos reservarPedido e libertarReserva, a verificação de existência do cliente sucede-se após a adição ou remoção da reserva propositadamente, pois o nosso intuito é otimizar o desempenho do sistema. ClienteNaoExisteException só poderia ocorrer em caso extremo, como erro interno, visto que o cliente tem de existir para poder solicitar a reserva a pedido ou libertação de reserva.

GestorLeilao

De forma a auxiliar a gestão dos leilões na classe GS, optamos por criar a GestaoLeilao, no intuito de conferir clareza.

Esta classe tem a finalidade de facilitar a atribuição da reserva ao licitador vencedor, aquando do término do tempo de licitação.

Assim, esta é uma *thread* que inicia um leilão, "acordando" após o tempo estabelecido para a duração do leilão em questão, permitindo a criação da reserva respetiva ao vencedor.

Desta forma será inserido o id da reserva ganha no leilão no set de reservas do vencedor.

ThreadCliente

Esta classe é fulcral para a comunicação entre o cliente e o servidor, permitindo que se tenha vários clientes conectados ao servidor ao mesmo tempo.

Nesse sentido, tem como variáveis de instância um Socket (que permite o estabelecimento de comunicação), GS, PrintWriter, BufferedReader e um identificador.

É nela que é feito o *parser* daquilo que o cliente comunica. O cliente pode solicitar, assim, uma reserva a pedido, uma reserva em leilão, a consulta do custo incorrido até ao momento, a libertação de um produto que esteja reservado por si, todos os produtos disponibilizados pelo sistema e as reservas que lhe estão associadas.

Servidor

Assim como a ThreadCliente, a classe Servidor também é imprescindível para a comunicação entre o cliente e o servidor, como é óbvio.

Na perspetiva do servidor, este aguarda uma nova conexão, criando uma ThreadCliente para cada cliente que efetue uma ligação.

Comunicação entre o Servidor e o Cliente

De forma a facilitar a comunicação entre o cliente e servidor, criou-se uma linguagem única.

Com o objetivo de tornar esta linguagem universal, implementou-se a classe ClienteEscrita, que apresenta no idioma escolhido as opções que o cliente pode optar no sistema e faz o *parser* da opção escolhida para a linguagem única, que será enviada para o sistema.

Esta linguagem única é, assim, interpretada pela classe ThreadCliente referida anteriormente, sendo executado o método associado à opção.

De forma a que o sistema comunique o *output* com o cliente, implementou-se também a classe ClienteThreadLeitura. Esta faz o *parser* desse output para o idioma escolhido.

Conclusão

Relativamente à elaboração das classes, inicialmente tínhamos conceptualizado a gestão individualizada dos clientes e dos produtos. No entanto, no desenvolvimento do projeto, apercebemo-nos de que seria mais vantajoso gerir conjuntamente.

No que concerne à preocupação em minimizar o tempo em que o nosso sistema se situa em região crítica, demos primazia à procura da melhor maneira possível de o fazer. Para isso, tivemos sempre em consideração qual seria a melhor estratégia para fazer o *lock* e o *unlock* de uma determinada estrutura, e aplicando sempre *two fase locking*, recorrendo a diversos testes para verificar possíveis *deadlocks*.

Através desses testes, apercebemo-nos de que seria benéfico termos uma estrutura com os identificadores das reservas associadas ao cliente na classe Cliente e uma estrutura que associasse leilões a um produto na classe GS, de forma a evitar bloquear a estrutura de reservas durante um longo período de tempo.

No que diz respeito à interface gráfica, tivemos de adicionar alguns métodos à nossa classe principal – a GS – com o objetivo de facilitar a forma como retínhamos informação para a mesma.

Em suma, consideramos que o objetivo inicial foi cumprido, conseguindo-se ter um sistema distribuído por diferentes utilizadores em diferentes máquinas, partilhando os mesmos dados, de forma organizada e sincronizada.