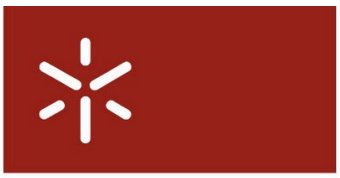


Universidade do Minho

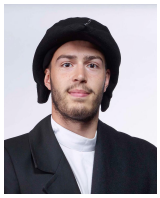
Sistemas Distribuídos

MIEI - 3º ano - 1º semestre
Universidade do Minho

Gestor de leilões


Dinis Peixoto
A75353


Ricardo Pereira
A74185


José Bastos
A74696


Marcelo Lima
A75210

16 de Novembro de 2017

Conteúdo

1 Introdução **2**

2 Implementação **2**

2.1 Utilizador 2
2.2 Comprador 2
2.3 Vendedor 2
2.4 Leilão 3
2.5 MensagemServidor 3
2.6 GestorLeilões 3
2.7 Menu 3
2.8 Cliente 3
2.9 ThreadClienteInput 3
2.10 ThreadClienteOutput 3
2.11 Servidor 3
2.12 ThreadServidorRead 4
2.13 ThreadServidorWrite 4
2.14 Exceptions 4

3 Comunicação Servidor-Cliente **4**

4 Controlo de Concorrência **5**

5 Interface **6**

6 Conclusão **7**

1. *Introdução*

No âmbito da cadeira de Sistemas Distribuídos foi-nos proposto a elaboração de um trabalho prático cujo o tema imposto seria a implementação de um "Gestor de Leilões". Desta forma, este projeto tem como objetivo a implementação de uma aplicação distribuída que faça a gestão de um serviço de leilões, podendo este ser acedido por dois tipos de utilizadores: vendedores e compradores. Os vendedores podem leiloar itens, tendo para isso a capacidade de criar e encerrar leilões. Por outro lado, os compradores podem fazer licitações sobre estes mesmos itens até o leilão ser encerrado, sendo o item vendido ao comprador com licitação mais alta até ao momento. Tudo isto será implementado usando um modelo cliente-servidor escrito em linguagem de programação JAVA, no qual os utilizadores devem poder interagir, intermediados por um servidor multi-threaded, e recorrendo a comunicação via sockets TCP.

2. *Implementação*

2.1 Utilizador

A classe **Utilizador** é uma classe abstrata que permite representar qualquer tipo de utilizador do nosso sistema de gestão de leilões. Os utilizadores da nossa aplicação podem ter dois tipos distintos: Comprador e Vendedor. Cada um destes será explicado de seguida.

2.2 Comprador

A classe **Comprador** é responsável por representar um utilizador da nossa aplicação que pretenda somente consultar e licitar os leilões disponíveis no momento.

2.3 Vendedor

A classe **Vendedor** é responsável por representar um utilizador que pretenda, para além de poder consultar os leilões já existentes, criar um leilão, permitindo assim que outros utilizadores (Compradores) licitem no mesmo.

2.4 Leilão

A classe **Leilao** representa um Leilão em curso. Nesta, estão representados todos os parâmetros que um leilão deve ter guardados, desde a sua descrição a uma lista dos licitadores que já efetuaram qualquer tipo de licitação no respectivo leilão.

2.5 MensagemServidor

A classe **MensagemServidor** representa as mensagens que o servidor envia para o utilizador, como resposta às suas ações. Veremos mais adiante o funcionamento da comunicação Servidor-Cliente onde esta classe tem um papel muito significativo.

2.6 GestorLeilões

A classe **GestorLeiloes** representa a classe responsável por armazenar toda a informação presente na aplicação num determinado momento. Nesta estão presentes todos os utilizadores registados, assim como todos os leilões em decurso e todas as mensagens servidor. Esta classe é partilhada por todos os utilizadores da aplicação, qualquer interação com a nossa aplicação implica um acesso obrigatório a esta.

2.7 Menu

A classe **Menu** é responsável por apresentar os diferentes tipos de menus, para os diferentes tipos de utilizadores e situações no nosso sistema.

2.8 Cliente

Esta classe representa o executável que qualquer utilizador da nossa aplicação irá usar para desfrutar da mesma.

2.9 ThreadClienteInput

A classe **ThreadClienteInput** representa a classe responsável por, para cada cliente, informar o servidor da ação que o cliente pretende efetuar.

2.10 ThreadClienteOutput

A classe **ThreadClienteOutput**, por sua vez, é responsável por receber do servidor informação para o cliente.

2.11 Servidor

Esta classe representa o executável que permite manter o servidor ligado enquanto os utilizadores desfrutam da aplicação.

2.12 ThreadServidorRead

A classe **ThreadServidorRead** é responsável por interpretar a informação recebida do cliente, fazendo a respectiva ação.

2.13 ThreadServidorWrite

A classe **ThreadServidorWrite**, por sua vez, é responsável por, após executada a ação do cliente, transmitir o resultador/informação para este.

2.14 Exceptions

Neste caso não se trata de uma classe mas de uma série delas. Os *Exceptions* são exceções enviadas quando acontecem situações indesejadas durante a execeução da nossa aplicação.

- LeilaoInexistenteException
- LicitacaoInvalidaException
- PasswordIncorretaException
- SemAutorizacaoException
- UsernameInexistenteException
- UsernameInvalidoException

3. *Comunicação Servidor-Cliente*

Para cumprir com os requisitos do enunciado, nós implementamos um servidor e um cliente que comunicam via sockets (TCP). Quando o Servidor aceita um Cliente, a comunicação entre estes fica estruturada em torno de quatro threads, duas threads para o lado do servidor, e as restantes duas para o lado do cliente. Uma das threads do Servidor, tem como objectivo receber mensagens por parte do Cliente, e a outra thread tem como objectivo enviar mensagens para o Cliente. Do lado do Cliente verifica-se a mesma estrutura, onde uma das threads do Cliente envia mensagens e outra recebe mensagens por parte do Servidor. As mensagens neste problema em questão, são orientadas à linha, isto é, a comunicação é feita através de linhas de texto (Strings), onde cada linha de texto equivale a uma mensagem. A imagem que se segue clarifica melhor a nossa estruturação.

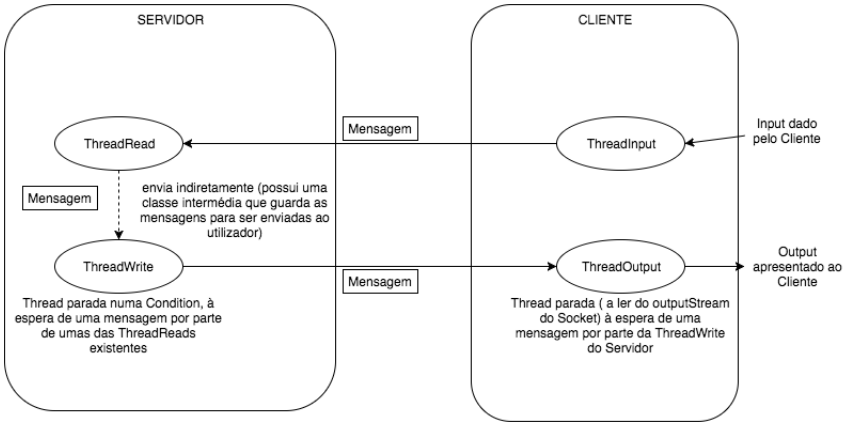


Figura 3.1: Esquema da comunicação Cliente-Servidor

É de referir que do lado do Servidor a comunicação entre a ThreadRead e ThreadWrite, é feita indiretamente, isto é, implementamos uma classe MensagemServidor, que guarda as mensagens a ser enviadas a um utilizador específico. Para além disso, para que a ThreadWrite saiba quando enviar uma mensagem, possui uma Condition, onde fica parada (*await()*), até que receba um sinal para acordar (*signal()*) por parte de uma ThreadRead, que quer enviar uma mensagem ao utilizador que este comunica. Qualquer ThreadRead que o servidor possui (uma ThreadRead é para apenas um Cliente), pode mandar mensagens para qualquer MensagemServidor que exista, permitindo assim que acções de outros clientes no Servidor possam ser notificadas a qualquer cliente.

4. *Controlo de Concorrência*

O principal objectivo deste trabalho é controlar o acesso ao conjunto de leilões e envio de mensagens para os diversos utilizadores, de maneira que o programa consiga lidar com a informação de vários clientes em simultâneo. Para isso, tivemos que utilizar mecanismos de controlo concorrência, que permitam ao servidor transmitir uma imagem de coerência em resposta a todos os clientes.

No código por nós desenvolvido, existem três classes, o GestorLeiloes, o Leilao e o MensagemServidor, onde podemos verificar que existe partilha, por todas as threads do Servidor. O GestorLeiloes, possui um conjunto de leilões, um conjunto de registos pertencentes aos utilizadores e um conjunto de MensagemServidor, para que seja possível haver comunicação entre os diferentes clientes. Para garantir que esta informação partilhada por todos, esteja sempre atualizada quando é pretendida por cada utilizador, implementamos um sistema que faz com que só um utilizador possa aceder à informação partilhada por todos os utilizadores de cada vez, uma vez que esta se trata da secção crítica do nosso código.

Sendo assim, na classe GestaoLeiloes, possuimos 3 locks, um para a lista de utilizadores, outro para a lista de mensagens, e por fim para a lista de leiloes, permitindo assim bloquear o acesso de vários utilizadores, enquanto um está a aceder aos objectos, e libertar o mesmo quando este acaba. Para além disso, sempre que um utilizador acede

a um leilão, também bloqueamos o acesso aos outros utilizadores, no entanto, o lock é referente a um leilão concreto, permitindo assim que dois clientes possam aceder a dois leilões diferentes, e não ao mesmo leilão em simultâneo.

O mesmo ocorre com a classe MensagemServidor, que bloqueia o acesso simultâneo dos utilizadores, perimitindo que utilizadores adicionem uma ou um conjunto de mensagens em diferentes MensagemServidor, mas não no mesmo, em simultâneo.

```
public void licitarLeilao(String idLeilao,Utilizador u,double lic) throws SemAutorizacaoException, LeilaoInexistenteException, LicitacaoInvalidaException{
    Leilao l;

    if(u.getClass().getName().equals("Vendedor")){
            throw new SemAutorizacaoException("Necessita de ser um Comprador para fazer uma licitação!");
    }
    this.leiloesLock.lock();
    try{
        if(!(leiloes.containsKey(idLeilao))){
            throw new LeilaoInexistenteException("Não existe nenhum leilão com tal ID!");
        }

        l = this.leiloes.get(idLeilao);
    }
    finally{
        this.leiloesLock.unlock();
    }

    String topo;
    l.getLock().lock();
    try{
        if(l.getLicAtual() > lic){
            throw new LicitacaoInvalidaException("Licitação com valor inferior a última licitação!");
        }
        else if(l.getLicAtual() == lic){
            throw new LicitacaoInvalidaException("Licitação com valor igual a última licitação!");
        }

        topo = l.getVencedor();
        l.licitar((Comprador)u, lic);
    }
    finally{
        l.getLock().unlock();
    }

    if(topo != null && topo != u.getUsername()){
        MensagemServidor m;
        this.mensagensLock.lock();
        try{
            m = this.mensagens.get(topo);}
        finally{this.mensagensLock.unlock();}

        m.setMsg("No leilao "+idLeilao+" foi ultrapassado pelo "+u.getUsername()+", com o valor de "+lic+"€",null);
    }
}
```

Figura 4.1: Exemplo de utilização dos diversos lock's.

Em suma, o programa por nós desenvolvido não deixa que dois utilizadores acedam a informação partilhada em simultâneo, permitindo assim que a informação esteja correta no momento em que é consultada.

5. *Interface*

A nossa aplicação apresentada uma interface muito simples, uma vez que o seu propósito é ser utilizado numa linha de comandos.

```
************ MENU ***************
* 1 - Iniciar Sessao             *
* 2 - Registar como Comprador    *
* 3 - Registar como Vendedor     *
* m - Mostrar Menu               *
* 0 - Sair                       *
**********************************
```

```
************ COMPRADOR ***********
* 1 - Licitar um Leilao           *
* 2 - Consultar Leiloes           *
* m - Mostrar Menu                *
* 0 - Terminar Sessao             *
**********************************
```

```
************ VENDEDOR ************
* 1 - Iniciar Leilao              *
* 2 - Consultar Leiloes           *
* 3 - Encerrar Leilao             *
* m - Mostrar Menu                *
* 0 - Terminar Sessao             *
**********************************
```

Figura 5.1: Menu inical, de um Comprador e de um Vendedor, respectivamente.

6. *Conclusão*

Com a realização deste trabalho pudemos aplicar todos os conceitos e o vasto conhecimento adquirido durante as aulas, pondo em prática grande parte daquilo que nos foi transmitido.

Tal como sabemos, os Sistemas Distribuídos têm como principal objetivo otimizar o desempenho de maneira que seja possível dividir tarefas e processá-las separadamente, promovendo assim paralelismo e reduzindo o tempo de execução.

Sendo a escalabilidade um dos mais importantes aspetos de um Sistema Distribuído, um dos nossos principais objetivos foi promover a concorrência, usufruindo para isso da implementação de Threads. No entanto, ao mesmo tempo que promovemos a concorrência é necessário assegurar a fiabilidade dos dados através da implementação de exclusão mútua em secções criticas, utilizando para isso os mecanismos adequados, tal como locks e variáveis de condição. Assim, podemos concluir que alcançámos os objetivos inicialmente formulados tanto ao nível das funcionalidades como da arquitetura da aplicação, de maneira que, com este trabalho conseguimos atingir um patamar bastante sólido no que diz respeito à implementação de um Sistema Distribuído.