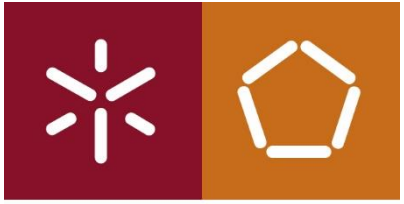

Universidade do Minho
Escola de Engenharia

Sistemas Distribuídos

Relatório do trabalho prático

Gestão de leilões

Grupo de Trabalho

Ana Esmeralda Fernandes A74321

Diogo Alexandre Machado A75399

Miguel Dias Miranda A74726

Rui Filipe Castro Leite A75551

Mestrado Integrado em Engenharia Informática

dezembro de 16

1. Introdução

Neste relatório é apresentado o trabalho desenvolvido para a unidade curricular de Sistemas Distribuídos onde se implementou em JAVA uma aplicação que permite interagir com um sistema de gestão de leilões. Considerando o contexto, desenvolveu-se uma aplicação capaz de dar resposta, em simultâneo, a vários utilizadores, fossem eles compradores de artigos em leilão, ou vendedores que disponibilizam artigos para leilão durante um tempo por eles determinado.

Para dar resposta à necessidade de concorrência e acesso a variáveis partilhadas da nossa aplicação foram implementados os conceitos de controlo de concorrência e comunicação via *Sockets* TCP lecionados na disciplina, os quais permitem a comunicação com o servidor *multi-threaded*.

2. Arquitetura da aplicação

2.1. Comunicação cliente – servidor

Quando o servidor é iniciado é aberto um *Socket* (definido, por omissão, na porta 9999) que ficará a correr enquanto espera por novos clientes. Sempre que um novo cliente tenta estabelecer ligação ao servidor são criadas e executadas duas *threads* para realizar a comunicação com esse cliente e satisfazer as funcionalidades da aplicação. Uma dessas *threads* executa o método `run()` duma instância da classe `ServerReader`, cuja função é interpretar os comandos enviados pelo cliente através do *Socket*, e executar a operação correspondente.

Para um maior controlo da comunicação entre um cliente e o servidor, decidiu-se usar uma *Log*. Cada um dos utilizadores registados na aplicação tem uma instância desta classe. A implementação da *Log* não passa de uma `BlockingQueue<String>`, uma classe de java que implementa a *interface* `Queue` e que tem a particularidade de fazer o controlo de acessos e de se bloquear quando se encontra cheia (tentando alguém lá escrever) ou vazia (tentando alguém de lá ler).

A thread responsável por interpretar os comandos vindos do cliente, escreve uma mensagem de controlo sobre o sucesso ou insucesso da execução desse comando para a `Log` atualmente em uso. A outra *thread*, que executa o método `run()` da instância da classe `ServerWritter`, ficará sempre a ler da `Log` à espera que lá surja alguma informação. Quando ler de lá alguma informação, envia a mensagem para o *Socket* do Cliente.

Sempre que um cliente realiza um pedido na porta onde se estabeleceu o servidor, além das duas *threads* criadas pelo servidor e referidas anteriormente, são lançadas outras duas, por parte do programa do cliente. Uma destas *threads* arranca o método `run()` de uma instância da classe `ClientWritter` que é responsável pela listagem da interface de menus ao utilizador e por pedir, diretamente ao seu `System.in` os comandos necessários à navegação pelos menus. Interpretados os comandos, a thread envia ao servidor uma mensagem que identifique a operação que deseja. Por exemplo, para fazer login, a mensagem enviada estará no formato:

"l|<username>|<password>"

O servidor terá a capacidade de interpretar esta mensagem (partindo a mensagem num `String[]`), pelo elemento `|` da mensagem. Serão desencadeados por parte do servidor as operações necessárias à concretização do pedido e no final, enviada a respetiva resposta. Caso o *username* indicado exista e a password coincida com a que está armazenada, então é enviada uma mensagem "`OK`" para a Log em operação no servidor, que, depois, e enviada ao cliente. Caso contrário, é enviada a mensagem de "`ERRO`".

A outra *thread* criada pelo programa cliente, executa o método `run()` da classe `ClientReader` e fica à espera de respostas por parte do servidor (enviadas pela *thread* da instancia `ServerWritter` do servidor). Esta thread interpreta as mensagens recebidas da seguinte forma: caso seja uma notificação do término de um leilão em que o utilizador com sessão iniciada participou, mensagem essa no formato:

"`f|<mensagem>`"

então é imediatamente imprimido no `System.out` do cliente o conteúdo da mensagem. Caso a mensagem contenha o caracter "`q`" então significa que é uma resposta a um pedido do cliente para terminar com o programa, e então a *thread* fica responsável por fechar a ligação com o servidor e o programa termina. Caso a mensagem recebida não esteja em nenhum destes dois formatos, então é porque é uma resposta a um qualquer pedido enviado ao servidor, e então, é escrita numa Log específica do cliente. A *thread* da instância de `ClientReader` fica então a ler as respostas por esta mesma Log e não diretamente no *Socket*.

A forma como são enviadas as mensagens de fim de um leilão a todos os seus licitadores é explicada num tópico posterior deste relatório.

2.2. Funcionalidades do Servidor e do Cliente

Dada a forma de comunicação explicitada anteriormente, consegue-se, com os métodos implementados no servidor, dar resposta às seguintes funcionalidades:

- Validação de login;
- Registar um novo utilizador;
- Inicializar um novo leilão;
- Licitar um item, dado o respetivo número de leilão, desde que não seja vendedor e que ofereça mais do que a maior licitação atual;
- Listar todos os leilões ativos, com a adoção da seguinte simbologia:
  - `+ <Leilao>` - quando é licitador do leilão e possui a maior licitação (é o atual vencedor);
  - `- <Leilao>` - quando é licitador do leilão, mas não possui a maior licitação (não é vencedor);
  - `* <Leilao>` - quando é vendedor do leilão;
  - `<Leilao>` - quando não é licitador do leilão.
- Fechar um Leilão, pelo seu vendedor e notificar todos os solicitadores do vencedor e da quantia conseguida;
- Sair do sistema de Leilões;

As referidas funcionalidades estão assim de acordo com aquelas que foram solicitas e pedidas nos requisitos do trabalho.

No contexto da aplicação de Gestão de Leilões o Cliente tem a funcionalidade de permitir navegar pela interface textual de menus, sejam os utilizadores vendedores ou compradores.

2.3. Classe `GestaoLeiloes` e classe `GestaoContas`

A classe `GestaoLeiloes` guarda toda a informação relativa aos leilões da aplicação. Foi tomada a escolha de guardar os leiloes num objeto `Map`, onde a chave é o número do leilão (valor único e auto incrementado) e o valor o objeto `Leilao` correspondente. Existe também nesta classe, um `Lock`, duas variáveis de condição e variáveis inteiras que são usadas para o controlo de concorrência do sistema de leilões, que será abordado no capitulo de controlo de concorrência do relatório.

A classe `Leilao` guarda toda a informação relativa ao leilão registado: a descrição do item, o *username* do vendedor, o *username* do vencedor até ao momento, o valor da maior licitação registada, todas as licitações feitas (a maior de cada licitador), se está ativo ou não e ainda todos os utilizadores que participaram. Esta última variável é crucial para implementar a funcionalidade de notificar os licitadores do fim de um leilão. Tem-se que a classe `Utilizador` guarda a correspondente `Log`. Com isto, consegue-se adicionar mensagens relativas a cada um dos utilizadores. Quando o vendedor decide terminar um leilão, são adicionadas as respetivas mensagens (de felicitação por ter ganho ou de informação do vencedor) às `Log`s dos licitadores. Quando cada um desses licitadores iniciar sessão, a *thread* do `ServerWriter` passará a ler dessa mesma `Log` e as mensagens que lá estiverem aparecerão no `System.out` do Cliente (depois de serem enviadas pelo Socket e de serem analisadas pela thread do Cliente `ClientReader`). Mesmo que o utilizador esteja com a sessão iniciada no momento do fecho do leilão, este será imediatamente notificado.

A classe `GestaoContas` regista toda a informação sobre os utilizadores registados no sistema. De igual modo escolhemos uma estrutura `Map` para guardar estas informações, sendo que a chave é a `String` do *username* do utilizador e a chave a instância de `Utilizador` correspondente. Também para a questão de controlo de concorrência esta classe tem um `lock` para regular o acesso ao `Map` de contas registadas.

Sendo que a informação completa do sistema de leilões é guardada nestas duas classes, o servidor tem assim uma variável do tipo `GestãoLeilao` designada `gl` e outra do tipo `GestaoContas` designada `gc`, além das variáveis necessárias para realizar a comunicação com os clientes. Desta forma, todas as funcionalidades que o servidor realiza a pedido dos clientes, são de consulta ou de escrita nos dados das instâncias relacionadas com estas duas classes.

3. Controlo de Concorrência

Pelo facto de as *threads* que são criadas acederem de forma concorrente a determinadas variáveis, foi necessário implementar um correto controlo de concorrência. Como as principais técnica de controlo surgem nas classes `GestaoContas` e `GestaoLeiloes`, estas são de seguida explicitadas.

3.1. Controlo de concorrência da classe `GestaoContas`

O controlo de concorrência implementado na classe `GestaoContas` foi feito recorrendo a um `ReentrantLock`. Como os métodos `registar()` e `validarLogin()` que a classe dispõe, necessitam de acesso ao `Map` de utilizadores que a classe possui, o controlo de concorrência é garantido pela tentativa de aquisição do `lock`, no inicio de cada método e a sua respetiva libertação quando os métodos conseguirem executar o seu conjunto de instruções. Deste modo garantimos a proteção da variável partilhada no acesso concorrente.

3.2. Controlo de concorrência da classe `GestaoLeilao`

Para a classe `GestaoLeilao` o controlo do acesso concorrente à sua estrutura de dados é também conseguido através de um `Lock`, das variáveis de condição `escrever` e `ler` e das variáveis inteiras `nExcritores`, `nQuerEscrever` e `nLeitores`.

Para um melhor controlo, foram criados quatro métodos para adquirir e libertar o `lock` ou ativar as variáveis de condição, conforme seja necessário ler ou escrever. O método `lerLock()`, para adquirir o `lock` numa tentativa de leitura, `lerUnlock()`, para libertar, `escreverLock()`, `escreverUnlock()`.

Para o método `registarLeilao()` é necessário garantir que não existe mais ninguém a escrever ou a ler nos dados dos leiloes. Para isso é usado o método `escreverLock()` onde são postos à espera quem pretender escrever enquanto existir quem esteja a ler ou a escrever. Depois de registado o leilão, pode-se marcar a saída da região crítica com `escreverUnlock()`.

Nos métodos `licitar()` e `fecharLeilao()` é necessário ler do `Map` de leilões para obter o `lock` do leilão em que se vai operar. Para isso faz-se `lerLock()`, obtém-se o `lock` do leilão (caso ele exista) e `lerUnlock()`. Depois de retido o `lock` do leilão, já se pode alterar as suas variáveis (como acrescentar licitadores ou marcá-lo como inativo).

O método `getLeiloesAtivos()`, usado para listar os leilões ativos, constrói uma `Collection<Leilao>` com uma cópia dos leilões registados. Antes de clonar cada um dos leilões ativos, é necessário fazer-se `lerLock()`, pela letura do `Map` dos leilões, e obter-se o `lock` de cada um dos leilões (libertado depois de fazer `clone()`).

Como a probabilidade de haver métodos a desejarem inserir novos leiloes é, em teoria, inferior à probabilidade de haver métodos a quererem listar leiloes, é dada prioridade a quem está a escrever ou a quem quer escrever (`nEscrever > 0` e `nQuerEscrever > 0`, respetivamente).

4. Conclusão

Com o presente trabalho, conseguimos cumprir os requisitos e objetivos das diretivas iniciais deste projeto. Com a implementação de um sistema *multi-threaded* e com a necessidade de implementar diretivas de controlo de acesso concorrente a dados partilhados, aplicamos todos os conceitos abordados na unidade curricular assim como compreendemos a importância desta estruturação e organização do código de forma a comunicar com sistemas que podem estar localmente dispersos e que acedem aos mesmo dados, sem perder a consistência e veracidade destes.