Trabalho Prático de Sistemas Distribuídos

Gestor de Leilões

Mestrado Integrado em Engenharia Informática

Universidade do Minho

Dezembro de 2016

Bruno Renato Carvalho 67847

Francisco Duarte Ventura 58529

João Ismael Reis 75372

Jorge Filipe Barreira 58511

Índice

1. Introdução 3
2. Implementação 3
   2.1. Servidor 4
      2.1.1. Lógica de negócio 4
      2.1.2. Multi-threading 4
   2.2. Cliente 5
3. Discussão de Resultados 5
4. Conclusão 6

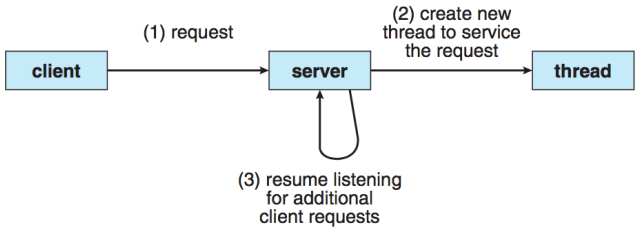
1. Introdução

O presente trabalho prático foi proposto com o objectivo de aplicar os conhecimentos adquiridos ao longo do semestre em Sistemas Distribuídos, na implementação (em JAVA) de uma aplicação capaz de gerir um serviço de leilões. A aplicação teria de ser capaz de servir dois tipos de actores: os compradores e os vendedores. Tratando-se de uma aplicação distribuída, esta seria formada por um servidor *multi-threaded* capaz de gerir múltiplos clientes que a ele se ligam através de *sockets* TCP.

Os requisitos propostos no enunciado do trabalho obrigam a que a implementação tenha especiais cuidados na gestão de acessos concorrentes às mesmas estruturas de dados. Por exemplo, será necessário gerir a forma pela qual múltiplos clientes licitam concorrentemente um dado leilão, impondo-se sempre a correção e a eficiência dos algoritmos utilizados. Era exigível que a aplicação suportasse o registo de utilizadores (compradores e vendedores), possibilitasse a um vendedor a funcionalidade de iniciar e terminar um leilão, ao comprador a funcionalidade de licitar nos leilões disponíveis e a ambos utilizadores listar a gama de leilões abertos.

2. Implementação

Tal como adiantado na secção anterior, a aplicação é implementada à custa de dois programas distintos mas interdependentes. Por um lado o servidor tem a função de aceitar conexões com os clientes que a ele se tentam acoplar, atribuindo a cada um destes uma nova *Thread*, possibilitando desta forma servir múltiplos clientes ao mesmo tempo, naquilo que é um algoritmo clássico utilizado em servidores web, por exemplo.



Arquitectura de servidor *multi-threaded.*

O cliente é munido de uma lógica mais simplista. Este apenas tenta conectar-se ao servidor e em caso de sucesso comunica através do envio e recepção de mensagens (*i.e. Strings*) cuja semântica possibilita o funcionamento de uma casa de leilões com as características já enunciadas.

Findo o resumo da arquitectura da aplicação, passemos à descrição mais pormenorizada da implementação do servidor e cliente.

2.1. Servidor

O servidor é responsável por manter em memória os dados necessários à lógica de negócio da aplicação:

2.1.1. Lógica de negócio

A lógica de negócio é composta por uma classe principal, de seu nome "Leiloeira". Esta classe é composta por estruturas de objectos (como leilões e utilizadores):

```
public class Leiloeira {

      private Integer incrementador;
      private TreeMap<Integer,Leilao> ativos;
      private TreeMap<Integer,Leilao> historico;
      private TreeMap<String,Utilizador> utilizadores;
      private Locker locker;

      ...
}
```

De referir, que por uma questão de necessidade de um maior controlo e flexibilidade da concorrência a estas estruturas, foi utilizada a interface "Lock" do JAVA, permitindo uma sincronização explícita sobre cada uma das estruturas referidas acima. Para que o código se tornasse o mais explícito quanto possível, as directivas de aquisição de *lock*, espera por *Conditions*, etc. foram, na sua maior parte encapsuladas num objecto da classe "Locker", que possui todos os métodos de gestão de acesso às estruturas da "Leiloeira". Estes métodos de acesso às estruturas foram baseados no problema (clássico) de programação concorrente "leitores-escritores", em que se pretende que múltiplas *threads* possam ler uma estrutura de dados simultaneamente, mas deverá ser impedido tal acesso sempre que uma *thread* modifica essa mesma estrutura.

2.1.2. Multi-threading

Para gerir todos os clientes que se vão associando, é arrancada uma *thread* nova para cada cliente, à qual se passa um "Handler", (*i.e.* um objecto da classe que implementa o método "run()". A esse "Handler" é passado como referência a "Leiloeira", assim como o *socket* para que haja comunicação com o cliente. De referir, que depois de um cliente se autenticar, no interior do método "run()" do "Handler", é criado um novo *thread* auxiliar para lidar com o caso em que se necessita de enviar para o cliente mensagens fora-de-tempo. Estas mensagens são originadas quando um vendedor termina um leilão e é necessário

avisar todos os clientes envolvidos nesse leilão desse mesmo facto. Esse novo *Thread* é arrancado via um objecto da classe “HandlerAsynchronous”, cujo método “run()” consiste apenas na tarefa de avisar o cliente ao qual está associado, que um leilão em que participou acabou de ser terminado.

2.2. Cliente

O cliente (*i.e.* o programa Cliente), disponibiliza ao utilizador da aplicação os métodos IO que lhe permite interagir com o sistema. O código IO é feito de forma *ad hoc*, orientado à linha e sem qualquer existência de uma interface gráfica. Os caracteres inseridos pelo utilizador no *standard input* são validados, e se estiverem de acordo com o formato exigido pelo protocolo de comunicação servidor-cliente são enviados pelo *socket* para o servidor e este envia a resposta aos pedidos. Também no cliente é criada uma *thread* auxiliar, via objecto da classe “HandlerListener”, cuja função é a de receber as mensagens que o servidor transmite e imprimir no *standard output* as mensagens assíncronas que possivelmente são enviadas pelo “HandlerAsynchronous”. De referir ainda a existência também de uma classe “Locker” responsável por conter uma cache da última mensagem enviada pelo servidor, mensagem esta que é também mediada por uma política de sincronização explícita (para impedir competição entre a *main thread* e a que é arrancada via “HandlerListener”).

3. Discussão de Resultados

No fim da implementação da aplicação, este grupo de trabalho considera que o trabalho realizado cumpre os requisitos que foram impostos pelo enunciado. Mais, nunca no decorrer da implementação da aplicação foi seguida a abordagem mais fácil por ser mais cómoda. Antes, foi seguida uma abordagem o tanto mais eficiente quanto possível. Por exemplo, em todos os métodos da classe “Leiloeira” foi evitado, a todo o custo, o bloqueio total do objecto, apesar de essa ser a alternativa que garantiria a correcção dos algoritmos, sem que tal exigisse um escrutínio de grão muito fino. Essa alternativa, no entanto, limitaria a concorrência e o paralelismo de *threads*, e isso, como sabemos, não é o objectivo desta unidade curricular.

Apesar do que foi mencionado, é possível verificar que existem diversos aspectos que poderiam ser de sobremaneira melhorados. Embora seja impossível enumerar neste documento todo o trabalho que deveria ser feito para tornar esta aplicação, uma aplicação comercial que fosse estável em todas as situações, podemos referir as que nos parecem mais relevantes. Em primeiro lugar, seria importante implementar uma classe de testes

automáticos que colocasse um grande número de clientes a fazer em simultâneo operações sobre a “Leiloeira", dessa forma, poder-se-iam detectar erros de execução que poderão estar escondidos pelo facto de nunca ter sido feito um teste com uma concorrência de relevo. Seria também necessário, no caso do servidor, em vez de estar ininterruptamente a criar *threads* para cada cliente que se conecta, criar uma *pool* de *threads*, evitando dessa forma potenciais riscos de esgotamento de memória e também promoveria a reutilização de *threads*, cuja criação é, como sabemos, um processo muito custoso. Seria também muito útil e obviamente essencial, criar a persistência dos dados (leilões, utilizadores) numa base de dados e também criar uma interface gráfica para melhorar a experiência de utilização da aplicação.

4. Conclusão

Como conclusão deste trabalho podemos referir que a linguagem de programação JAVA fornece uma API completa para a gestão de concorrência entre *threads*. O uso da directiva *synchronized* sobre métodos e objectos, apesar de raramente usada na nossa implementação, gere, de forma opaca ao programador acessos concorrentes a zonas críticas. Escolhemos no entanto, na maioria dos casos, usar sincronização explícita, com *locks* e *conditions*, pois consideramos que necessitávamos da flexibilidade que essas ferramentas proporcionam. Esta estratégia traduz-se, porém, num maior risco pois atribui-nos a nós, programadores, a responsabilidade de gerir manualmente acessos às ditas zonas críticas.

Por último, é de referir que o grupo de trabalho se encontra satisfeito com o trabalho produzido, ficando no entanto a sensação de que muito mais poderia ser feito, houvera mais tempo para implementar a aplicação.