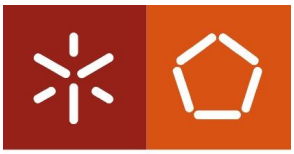

Universidade do Minho

Departamento de Informática

Sistemas Distribuídos

Filipe Monteiro (a80229)
Bruno Martins (a80410)
Márcio Sousa (a82400)

6 de Janeiro de 2019

Conteúdo

1 Introdução

No âmbito da cadeira de Sistemas Distribuídos foi nos proposta a elaboração de um trabalho prático com o objetivo de criar um serviço de alocação de servidores e de contabilização do custo incorrido pelos utilizadores. Este programa deverá então possibilitar a autenticação e registo dos utilizadores, fazer a reserva e a libertação de servidores, permitindo que a reserva seja feita tanto por pedido como por leilão, e deverá também permitir ao utilizador consultar o valor que este terá que pagar pelo uso dos recursos. Tudo isto será implementado usando um modelo cliente-servidor, recorrendo a comunicação via sockets TCP.

2 Arquitetura da Solução

A primeira fase consistiu em decidir que estruturas e mecanismos usar para evitar qualquer *Deadlock* que o bloqueasse, ou sintomas de *Starvation*. Foi decidido a favor de uma solução orientada à mensagem, em que todas as comunicações são feitas através do envio e receção de Strings que seguem a mesma estrutura base:
<ação>-<argumento1>-<argumento2>-...-<arguentoN>.
Como por exemplo, no caso do registo de um novo utilizador, a String enviada tem a estrutura **register-nome-email-password**. A arquitetura da solução divide-se em 3 *Packages*:

- **Client**
- **Server**
- **Common**

sendo as classes importantes e objetivos de cada um explicados de seguida. São apenas explicadas as classes importantes devido ao limite de páginas imposto.

2.1 Client

É aqui definido todo o processo do lado do utilizador que está a fazer a reserva do servidor, sendo definidas todas as funcionalidades a que cada utilizador tem acesso. Foi dividido em duas partes: o Client e o ClientController.

2.1.1 Client

É esta a classe responsável por inicializar a *Socket* que estabelecerá ligação com o servidor, e por apresentar o Menu Inicial ao utilizador, através do qual serão chamados os devidos métodos do ClientController, dependendo da mensagem que recebe através do *input* na *Socket*.

2.1.2 ClientController

São aqui definidos os métodos que codificam todas as possíveis ações a que um utilizador tem acesso. Quando este inicializa o programa, são lhe apresentadas as opções de **Log In** e **Register**. Apenas utilizadores que já estejam registados conseguirão reservar servidores e aceder a todas as funcionalidades inerentes. Ao utilizador que está *logged in* são apresentadas, como opções no Menu Inicial, cada uma das funcionalidades. O utilizador pode ver os detalhes da sua conta, os servidores que estão reservados por si, pode fazer nova reserva por pedido ou cancelar alguma das suas já existentes, fazer reserva de um servidor por leilão ou simplesmente escolher a opção **Quit** e sair da aplicação.

2.2 Server

Este **Package** contém todas as classes relacionadas com o comportamento do servidor face a cada um dos *requests* recebidos do lado do **Client**. É composto por classes que definem o funcionamento do servidor, mas também por "classes estruturais"que definem mais alguma entidades envolvidas no funcionamento do programa.

2.2.1 Container

A classe **Container** representa aqui um servidor, quer esteja livre ou já alocado a alguém. A cada **Container** associa-se um ID, um tipo de servidor, um preço, a data de inicio da reserva do mesmo e o utilizador que o alugou, que caso não exista, está a `null`. É aqui onde se definem os métodos que permitem a atribuição de um servidor a certo utilizador, ou a sua libertação.

2.2.2 Middleware

É esta classe que contém a "lógica de negócio". Aqui se encontram os Maps e as Lists onde são guardados os containers, os utilizadores e todas as restantes entidades necessárias. O **Middleware** é responsável também por registar novos utilizadores e por efetuar o *log in* de utilizadores existentes.

Da mesma forma, é a partir daqui que são abertas e fechadas as *auctions*, realizadas ou canceladas reservas.

2.2.3 Server

Classe "main"do servidor, onde este é criado, instanciado os tipos de servidores e aceitadas conexões de clientes. Para podermos atender vários clientes, associamos uma *Thread* à medida que cada cliente se liga:

```
ServerThreadConnection thc = new ServerThreadConnection(ss.accept(), m);
executor.submit(thc);
```

em que **m** é o *middleware* que contém a lógica/estado do programa.

Visto que criar *threads* é custoso, reutilizamo-las usando **ThreadPools** - uma *thread* é criada apenas se não existir nenhuma em espera (estas serão eliminadas após x tempo de inatividade).

2.2.4 ServerThreadConnection

Nesta classe é mapeado a ação que o utilizador introduziu com os diferentes métodos do servidor.. É aqui decidido o que deve ser ser feito do lado do servidor, dependendo do que o utilizador tiver requerido no Menu Principal.

2.2.5 ServerThreadController

Para tornar o código mais limpo, foi definida a classe **ServerThreadController** para definir os métodos que executam as ações que o utilizador pretende. Os metodos são chamados na classe **ServerThreadConnection** mediante o *input* do utilizador.

2.2.6 Timer

O Timer é o responsável pelo fecho das *auctions* quando estas estiverem prontas para tal. É executado por uma thread que se encontra inativa á espera de alguma *auction* pronta a ser fechada, fechando-a de seguida.

2.3 Common

O *Package* Common é maioritariamente composta por definições de **Exceptions**. Há, no entanto, duas classes que se destacam no meio das **Exceptions**: a classe **User** e a classe **Pair**.

2.3.1 User

Nesta classe define-se a estrutura que representará cada um dos utilizadores, juntamente com métodos de acesso aos dados do utilizador e o método de autenticação. Cada utilizador terá atribuído a si um ID, um nome, um email, uma password e uma dívida que corresponde ao montante que deverá ser pago pela reserva de cada um dos servidores que possuiu até momento.

2.3.2 Pair

Aqui define-se uma estrutura auxiliar que é composta por um utilizador e pelo preço que ele está disposto a pagar por um servidor de um certo tipo em leilão. Estes **Pairs** são inseridos num **HashMap** que funciona como *queue* do sistema, quando o utilizador quer reservar um servidor de um tipo do qual não existem, de momento, servidores livres. Este **HashMap** usa como *key* o tipo de servidor que deseja reservar. Quando é libertado algum servidor de um certo tipo, é procurado na *queue* pelo utilizador disposto a pagar o maior preço pelo servidor, e atribui-lhe o servidor.

3 Controlo de Concorrência

Para o controlo de concorrência, no middleware utilizamos *ReadWriteLocks* da seguinte forma:

- write: quando adicionamos/removemos componentes do estado partilhado;
- read: quando consultamos este mesmo estado.

Temos ainda um método *synchronized* na classe *Auction*, pois só faz sentido haver uma pessoa de cada vez a realizar propostas de compra do servidor.

4 Conclusão e Trabalho Futuro

4.1 Trabalho Futuro

No âmbito de melhorar o desempenho e acrescentar funcionalidades extras, pensamos em, antes de mais, tentar diminuir ainda mais as zonas críticas para prevenir tantos tempos de espera grandes em certas ações no código. Como funcionalidades extras, adicionar uma camada de comunicação por onde as mensagens cliente-servidor e servidor-cliente passam para facilitar o envio/recolha de mensagens. O grupo começou implementação mas cedo se chegou à conclusão que não haveria tempo para

complementar corretamente, tendo sido abondonado (continua, no entanto, no projeto para possiveis tentativas futuras).
O terminal do cliente também poderia ser melhorado, permitindo um maior número de ações que a este seriam úteis.

4.2 Conclusão

Uma aplicação desenvolvida tendo em mente sistemas distribuídos exige um planeamento muito mais cuidadoso e estudado. Tanto o paralelismo como a concorrência trazem novos problemas a uma solução que pode ser complicada por si só. *Deadlock*, dependências circulares, *starvation* são só alguns dos problemas introduzidos neste paradigma. Uma arquitetura pensada em função de paralelismo pode não ser viável para pequenas aplicações dado o *overhead* que introduz.

As vantagens dos sistemas distribuídos são no entanto bastante evidentes. Graças à repartição de tarefas o tempo de execução pode ser reduzido de forma significativa. Com a utilização de *threads* conseguimos aproveitar o CPU de forma mais eficiente. Em junção com a capacidade de comunicar entre várias máquinas, usando *sockets* TCP, conseguimos criar uma aplicação que pode ser usada por vários utilizadores e de forma assíncrona.

Desta forma este trabalho possibilitou aprofundar o conhecimento de programação *multi threaded* através de *threads* da *JVM* seguindo a arquitetura dada na disciplina, a arquitetura *cliente servidor*.