Projeto de Sistemas Distribuídos - G20

Diogo Braga A82547 João Silva A82005
Ricardo Caçador A81064 Ricardo Veloso A81919

6 de Janeiro de 2019

Resumo

Neste relatório é apresentado um projeto relacionado com Alocação de servidores na nuvem, desenvolvido no âmbito da unidade curricular de Sistemas Distribuídos.

Conteúdo

1 Introdução

Este trabalho tem como objetivo a implementação de um sistema de reserva e uso de servidores que terá um custo que será prosteriormente contabilizado. A reserva pode ser feita tanto a leilão como a pedido. No caso de a reserva ter sido feita a leilão é necessário o custo ser escolhido pelo utilizador do servidor enquanto que a reserva a pedido já tem um custo associado. Em termos de funcionalidades é imperativo que exista a possibilidade de reservar um servidor, libertar o mesmo, consultar a conta corrente e, obviamente, fazer login de forma a ter acesso a todas estas funcionalidades. Também deve ser possível registar Clientes no sistema.

Com a realização deste relatório é pretendido que o leitor consiga entender, de uma forma esclarecedora, todos os métodos e ideias implementadas para a realização do trabalho proposto.

2 Abordagem ao problema

Perante o enunciado apresentado, achamos necessário ter as seguintes classes:

- **MainServer**, que tem como principal função inicializar o **ServerSocket**, aceitar os Clientes e estabelecer uma ligação Socket com cada um deles através da thread MainThread.
- **MainThread**, que contém a execução por parte do Servidor no Socket. Esta classe possui as variáveis de instância:
  - **out** : PrintWriter do Socket;
  - **in** : BufferedReader do Socket;
  - **cs** : Socket;
  - **users** : acesso aos Utilizadores;
  - **servers** : acesso ao Map Servers;
  - **owner_user_thread** : username do cliente que está a usar o Socket.
- **Client**, que contém a execução por parte do Cliente no Socket. Esta classe possui as variáveis de instância:
  - **out** : PrintWriter do Socket;
  - **in** : BufferedReader do Socket;
  - **sin** : BufferedReader do Socket.
- **Utilizadores**, que possui:
  - **Map <String,Utilizador >** para armazenar todos os Clientes que se conectam com o Servidor. A chave utilizada no Map é o username do Utilizador.
- **Utilizador**, que caracteriza um Utilizador. Esta classe possui as variáveis de instância:

  - **username**;
  - **password**;
  - **debt** : débito;
  - **autenticado** : se está autenticado ou não.

- **Servers**, que possui:
  - **Map <String,Server >** para armazenar todos os Servidores disponíveis a serem requisitados. A chave utilizada no Map é o nome do Server.

- **Server**, que caracteriza um Servidor a ser requisitado. Esta classe possui as variáveis de instância:
  - **l** : Lock;
  - **leilao_server_ocup** : Condition;
  - **pedido_server_ocup** : Condition;
  - **id_name**;
  - **preco**;
  - **total_servers_pedido** : número de servidores para reservas a pedido;
  - **total_servers_leilao** : número de servidores para reservas a leilão;
  - **num_servers_pedido** : número de servidores em uso para reservas a pedido;
  - **num_servers_leilao** : número de servidores em uso para reservas a leilão;
  - **ticket_atual** : identificador da reserva;
  - **users** : acesso aos Utilizadores;
  - **valores_leilao** : Map que possui as ofertas dos utilizadores para os servidores a leilão;
  - **num_pedido** : número de threads à espera de obterem reservas a pedido;
  - **num_leilao** : número de threads à espera de obterem reservas a leilão;
  - **proximo_a_entrar** : user com a maior oferta do momento em espera.

- **ThreadServer**, que contém a execução do Servidor requisitado, no Socket do lado do Cliente. Esta classe possui as variáveis de instância:
  - **users** : acesso aos Utilizadores;
  - **preco**;
  - **owner_user_server** : username do cliente que está a usar o Socket com o Servidor requisitado;
  - **cs** : Socket;

  - **ticket_atual**;
  - **id_name**.

- **ValoresLeilao**, que possui:
  - **Map <String,Double >** que possui as ofertas dos utilizadores para os servidores a leilão. A chave utilizada no Map é o nome do Utilizador.

3 Cliente-Servidor com Sockets TCP

Tendo em conta o requerido no enunciado, implementamos um servidor que aceita a conexão de múltiplos clientes, isto é, um servidor multithreaded no qual existe partilha de estado.

A ligação Cliente-Servidor foi estabelecida através de Sockets TCP e é assegurada pelas classes Client e MainServer. Assegurada esta ligação o Servidor começa o programa e inicia uma Thread (MainThread) que vai tratar da interação do Servidor com o Cliente.

4 Variáveis de Condição

As variáveis de condição que usamos no nosso trabalho, ambas associadas ao Lock de cada Server, foram **pedido_server_ocup** e **leilao_server_ocup**.

A primeira é utilizada para suspender (**await()**) as threads quando estas pretendem obter um servidor a pedido, e os servidores do tipo que o Cliente pretende estão todos ocupados (**num_servers_pedido** == **total_servers_pedido**). Quando uma thread liberta o servidor ao qual estava associado, uma outra thread que estava suspensa na condição é acordada (**signal()**) para que possa adquirir o servidor do tipo que pretende. A thread escolhida para acordar é a reserva mais antiga, visto a fila de espera ser FIFO.

A segunda é utilizada para suspender (**await()**) as threads quando estas pretendem obter um servidor a leilão, e os servidores do tipo que o Cliente pretende estão todos ocupados (**num_servers_leilao** == **total_servers_leilao**). Quando uma thread liberta o servidor ao qual estava associado, todas as outras threads que estavam suspensas na condição são acordadas (**signalAll()**) para que possam concorrer pelo servidor do tipo que pretendem. É calculado o utilizador que tiver a oferta mais elevada (**valores_leilao.maiorLicitacao()**) e é lhe atribuido o servidor, sendo que todas as outras threads voltam a ficar suspensas na variável de condição. Algoritmicamente, a thread que consegue sair do ciclo é a que tem o nome igual à do utilizador com a maior oferta **(!owner_user_thread.equals(proximo_a_entrar))**.

5 Exclusão Mútua

Quanto ao controlo de concorrência no nosso projeto, achamos necessário que este exista nos seguinte pontos:

Ao nível do Map dos Utilizadores:

- **Registo e Autenticação** : Utilizando synchronized, garantimos que duas ou mais threads não conseguem registar, ou autenticar, o mesmo Utilizador. Caso tal acontecesse, iriamos estar perante um conjunto de utilizadores incoerente e um mau funcionamento do Sistema Distribuído.

Ao nível de cada Servidor:

- **Reserva a pedido** : Utilizando locks explícitos neste método garantimos que os incrementos e decrementos das variáveis de instância neste realizados não irão desenvolver problemas, pois só uma thread de cada vez acede a estas mesmas. Também com o lock, garantimos o funcionamento da variável de condição pedido_server_ocup.

- **Reserva a leilão** : Utilizando locks explícitos neste método garantimos que os incrementos e decrementos das variáveis de instância não irão desenvolver problemas, pois só uma thread de cada vez acede a estes mesmos. Também com o lock, garantimos o funcionamento da variável de condição leilao_server_ocup. Neste método é realizado lock hierárquico com a classe ValoresLeilao de forma a que o acesso a esta mesma ocorra corretamente.

- **Número de servidores em uso para reservas a pedido/leilão** : Utilizando uma variável volatile garantimos que as alterações realizadas nestas variáveis de instância são escritas em memória em vez de permanecer na cache. O contrário faria com que pudessem haver problemas na leitura destas variáveis, visto a memória ser partilhada.

- **Incremento do ticket** : Utilizando uma variável volatile garantimos que cada thread incrementa corretamente o ticket. Isto acontece porque de cada vez que há um incremento esta alteração é escrita na memória em vez de permanecer na cache. Assim, não existe hipótese de existirem problemas.

- **Atribuição a pedido** : Utilizando locks explícitos neste método garantimos que os decrementos das variáveis de instância neste realizados não irão desenvolver problemas, pois só uma thread de cada vez acede a estas mesmas. Também com o lock, garantimos o funcionamento da variável de condição pedido_server_ocup.

- **Atribuição a leilão** : Utilizando locks explícitos neste método garantimos que os decrementos das variáveis de instância, assim como os métodos neste realizados, não irão desenvolver problemas, pois só uma thread de cada vez acede a estes mesmos. Também com o lock, garantimos o funcionamento da variável de condição leilao_server_ocup.

Ao nível da classe ValoresLeilao:
Utilizando locks explícitos para esta classe, garantimos que todos os métodos nela presentes são executados sem problemas.

- **Adição, Remoção e Cálculo da maior licitação** : Utilizando controlo de concorrência garantimos que estas operações funcionam corretamente, visto só se poder adicionar, ou remover, um utilizador de cada vez. Quanto

ao cálculo, imaginando a existência de mais do que uma thread a realizar tal operação, as atribuições poderiam ser trocadas durante o funcionamento, e o resultado final seria errado.

Além disto, como o lock é da classe, garantimos que nenhum utilizador é adicionado ou retirado da estrutura enquanto ocorre o cálculo da maior licitação, por exemplo.

6 Criação de Threads e Partilha de Estado

No nosso projeto existem dois momentos onde que se criam threads.

Primeiro, no estabelecimento da ligação Cliente-Servidor (Socket) na classe MainServer, onde se cria a thread que vai possuir o lado de comunicação do Servidor. Tal acontece devido à conexão de Clientes. A thread inicializada (MainThread) implementa a interface Runnable, e portanto, funciona em concorrência com as outras threads criadas para os restantes Clientes conectados ao Servidor. Nesta fase, são também criados e passados como parâmetro a cada thread os objetos Utilizadores e Servers usados em todo o funcionamento do programa. São estes dois objetos que dão origem a um conceito de Estado Partilhado, pois é algo que as threads, que funcionam em concorrência, utilizam.

Segundo, quando é atribuido um Servidor a um Cliente, e se cria a thread (ThreadServer) que vai possuir o lado de comunicação do Servidor. Neste caso, o Estado Partilhado são os Utilizadores.

7 Conclusão

Ao longo do desenvolvimento do projeto fomo-nos deparando com várias situações de decisão e escolha de diferentes abordagens. A abordagem final foi a que temos vindo a falar ao longo deste relatório.

Existem sempre algumas particularidades de implementação, como por exemplo, o uso de variáveis **final**, para que não fossem mudados espaços de memória associados a certas variáveis. Mas de todas essas particularidades não podemos falar, dado a maior relevância de outros temas perante estes detalhes.

Duma forma muito geral o grupo está satisfeito com o projeto realizado, dado que praticamente todas as funcionalidades que eram requisitadas foram implementadas. Excetuando apenas a funcionalidade que cita que *"Em caso de indisponibilidade de servidores do tipo pedido, poderão ser canceladas reservas concedidas em leilão para obter o servidor pretendido."*.

Não podemos deixar de dizer que tentamos implementar esta funcionalidade. Para tal criamos uma estrutura (List<Thread>) que guardava todas as threads que estavam a providenciar reservas a leilão, e quando chegasse uma reserva a pedido acedíamos a esta estrutura e aplicávamos o método interrupt(), ou stop(), ou destroy() a uma thread random. De qualquer forma este método parecendo-nos o mais lógico não funcionava e consequentemente não implementamos esta funcionalidade.