Relatório do Trabalho Prático de Sistemas Distribuídos

André Gonçalves (a80368) Diogo Gonçalves (a81860)
Luís Alves (a80165) Rafaela Rodrigues (a80516)

6 de Janeiro de 2019

Resumo

Este relatório pretende identificar a arquitetura da solução desenvolvida bem como indicar algumas decisões tomadas, terminando com um conjunto de conclusões sobre o trabalho realizado.

Conteúdo

1 Introdução

Para este trabalho prático, foi sugerida a criação de um sistema de aluguer de *Clouds* (similar a serviços disponibilizados por Amazon, Google, etc), que permitisse o acesso concorrente de vários utilizadores, recorrendo para isso a Sockets TCP em Java. A arquitetura da solução é baseada no modelo cliente-servidor, sendo o servidor *multi-threaded* para lidar com os vários utilizadores simultaneamente.

2 Arquitetura

De forma a organizarmos melhor o código produzido e conseguirmos dividir melhor o trabalho a realizar pelos diversos elementos do grupo, dividimos a arquitetura em 4 *packages*:

- Client
- CloudAlloc
- Server
- Exceptions

Para cada um destes *packages* está destinada uma subsecção deste capítulo, com exceção do *package* `Exceptions`, cujas classes apenas foram criadas para distinguir os diversos tipos de exceções no código, mas não acrescentam métodos relevantes que mereçam menção neste relatório.

2.1 Client

2.1.1 Client e ClientListener

Este *package* contém as classes `Client` e `ClientListener`, que representam o executável que um utilizador do sistema utilizará para poder requisitar ou leiloar *Clouds*. A classe `Client` recebe input do teclado via terminal e cria uma *Thread* que executa o método `run` da classe `ClientListener`, que se limita a receber todas as mensagens recebidas do Servidor e as expõe também no terminal.

É um cliente muito similar ao uso do `nc`.

2.2 CloudAlloc

Este *package* contém as classes que albergam a principal lógica da aplicação.

2.2.1 AuctionRequest e CloudRequest

Ambas as classes implementam a interface `Runnable`, uma vez que é criada uma nova instância sempre que um utilizador deseja obter uma *cloud* através de pedido ou leilão. Assim sendo, cada instância chama os métodos `auctionCloud`

ou `requestCloud` da classe `CloudAlloc` (de acordo com ser um pedido ou leilão), sendo que essas chamadas podem ser bloqueantes nesses dois métodos. Após ser retornado o valor do ID da *cloud* atribuída, é adicionada uma mensagem ao `MessageLog` do Utilizador que requisitou/leiloou a *cloud*, de forma a ser notificado quando possível.

2.2.2 Cloud

Esta classe define o objeto a ser comprado, tendo por isso como variáveis de instância um ID, um tipo, um preço nominal, uma data em que foi criada e uma *flag* que indica se foi leiloada ou não. Como todos estes campos nunca são alterados após ser criada uma nova instância, não há quaisquer métodos sincronizados ou se recorre ao uso de *locks*.

2.2.3 CloudAlloc

Esta classe armazena em memória toda a informação comum à aplicação, e é partilhada a mesma instância dela por todos as *Threads* associadas a cada utilizador que se conecta ao servidor. Em termos gerais, armazena as *clouds* em funcionamento e os utilizadores registados, pelo que cada um destes tem associado um *lock* distinto.

Assim sendo, tem 3 métodos associados à gestão de *clouds*, e 2 associados à autenticação dos utilizadores.

Relativamente à gestão das *clouds*, de forma a requisitar uma pelo seu preço nominal, é necessário fornecer ao método `requestCloud` o utilizador que realiza o pedido e o tipo da *cloud* que pretende requisitar. De forma a não termos inconsistências na leitura de dados relativamente ao estado das *clouds* do sistema, são usados *locks* para a consulta de informação dos *Maps* onde estão armazenadas, sendo que caso não exista nenhuma *cloud* disponível (incluindo leiloadas), a *Thread* que se encontra a executar o método fica adormecida até a condição associada ao tipo de *cloud* pretendida receber um sinal de que foi liberta uma *cloud* desse tipo. Assim sendo, a execução pode prosseguir e os *Maps* são atualizados, podendo por fim ser libertado o *lock* associado ao armazenamento de *clouds*.

Quanto ao leilão de *clouds*, o comportamento é semelhante, sendo que apenas não é verificada a disponibilidade de outras *clouds* caso o número de *clouds* reservadas tenha atingido o limite definido.

Relativamente à libertação de nuvens, esta pode ser feita tanto pelo sistema (quando é feito um pedido de *cloud* e existem *clouds* desse tipo leiloadas), quer pelo utilizador. Assim sendo, é também utilizado o *lock* associado às nuvens de forma a verificar se é possível remover a *cloud* e não provocar inconsistência nos dados. De seguida, verifica-se quem é o verdadeiro dono da *cloud* e é adicionada uma mensagem ao `MessageLog` do dono, de forma a ser notificado da alteração ao estado da nuvem.

Em relação à autenticação, ambos os métodos de `loginUser` e `registerUser` limitam-se a utilizar os métodos respetivos na classe `User`, sendo utilizado o

lock associado à gestão de utilizadores de forma a poder adicionar/verificar a existência de utilizadores com e-mails repetidos de forma segura e concorrente.

2.2.4 CloudTypes

Esta classe define diversas constantes importantes para o funcionamento da aplicação, nomeadamente os tipos de *clouds* existentes, o seu preço nominal e o número máximo de nuvens por tipo. Para além disso, possui também métodos de classe que permitem obter essa informação de forma mais organizada.

2.2.5 Counter

Esta classe é utilizada para definir novos IDs na criação de *clouds* na classe `CloudAlloc`, tendo para isso um *getter* sincronizado que atualiza uma variável de acordo com o número de vezes que foi chamado (sendo esse número o próximo ID retornado).

Para além disso, é usada também para definir o número de *requests* pendentes, informação utilizada para saber se é possível atribuir *clouds* a pedidos em leilão ou não (é sempre dada prioridade aos *requests*).

2.2.6 MessageLog

Esta classe permite armazenar um conjunto de mensagens e alertar aqueles que detêm o *lock* de cada instância se uma nova mensagem foi adicionada. Através do método `writeMessage` é adicionada uma nova mensagem ao *log*, sendo uma Condição associada ao seu *lock* sinalizada da chegada de uma nova mensagem. Através do método `readMessage`, é retornada a primeira mensagem não lida caso haja uma mensagem pendente e *null* caso não haja mensagens por ler. O adormecimento de *Threads* à espera de novas mensagens relativamente a este log é feito na classe `NotificationCenter`, que fica adormecida na condição referida anteriormente.

2.2.7 User

Esta classe define um utilizador no sistema, sendo que tem por isso associado um e-mail, uma palavra-passe, um conjunto de *clouds* que reservou e um `MessageLog`. Aquando do login/logout, é atualizada uma variável booleana que define se o utilizador está autenticado ou não. Quanto aos métodos de gestão de *clouds*, são sincronizados de forma a garantir que não houve alterações das *clouds* armazenadas (podem ter sido libertadas a meio da chamada do método). Permite também obter os valores em dívida de acordo com as nuvens atualmente ainda em reserva, bem como o total desde a criação da conta.

2.3 Server

Este *package* contém as classes que estão associadas à comunicação direta com o utilizador, através da definição da interface via linha de comandos e da utilização

de *sockets* para comunicação.

2.3.1 Menu

A classe `Menu` apenas contém métodos relativos às mensagens enviadas pelo Servidor ao Utilizador, como os menus da aplicação e mensagens de sucesso/insucesso relativas às ações do Utilizador.

2.3.2 NotificationCenter

A classe `NotificationCenter` implementa a interface `Runnable`, sendo que é iniciada uma nova *Thread* com uma nova instância desta classe sempre que um utilizador se autentica na aplicação. Esta classe trata de enviar ao utilizador notificações sobre reservas ou leilões concluídos ou libertação de *clouds*. Está ligada à classe `MessageLog`, uma vez que a *Thread* com a instância do `NotificationCenter` fica adormecida enquanto não for sinalizada a chegada de uma nova mensagem ao `MessageLog` do utilizador, usando para isso *Conditions*.

2.3.3 Server

A classe `Server` aguarda por novas conexões e, assim que uma nova é estabelecida, inicia uma nova *Thread* com uma instância da classe `ServerThread`.

2.3.4 ServerThread

A classe `ServerThread` é a que contém a maior parte da interação entre o Utilizador e o Servidor, sendo que recebe como parâmetro a instância partilhada por todos os utilizadores da classe `CloudAlloc`. É nesta classe que são processadas as decisões do utilizador face àquilo que lhe foi apresentado, quer seja no momento de autenticação, quer seja no momento de reservar um servidor. O único controlo de concorrência feito nesta classe, é quando um utilizador reserva/leiloa uma *cloud*, uma vez que é iniciada uma nova *Thread* de acordo com o pedido, de forma ao utilizador poder realizar outras ações entretanto, caso não seja possível ser-lhe alocada uma *cloud* assim que realizou o pedido (ver classes `CloudRequest` e `AuctionRequest`).

3 Conclusões

Um dos primeiros desafios deste trabalho foi a forma de notificar os utilizadores independentemente de eles estarem autenticados ou sequer conectados ao servidor. A solução inicialmente desenvolvida apenas funcionava caso o utilizador se mantivesse conectado, pelo que apenas posteriormente desenvolvemos a solução final que se assemelha ao funcionamento do *chat* criado para o último guião das aulas práticas.

Quanto a desenvolvimento futuro, este poderá passar pela definição de uma GUI com recurso a *JavaFX*, que não implicará grandes mudanças na arquitetura da aplicação, dada a sua modularização.