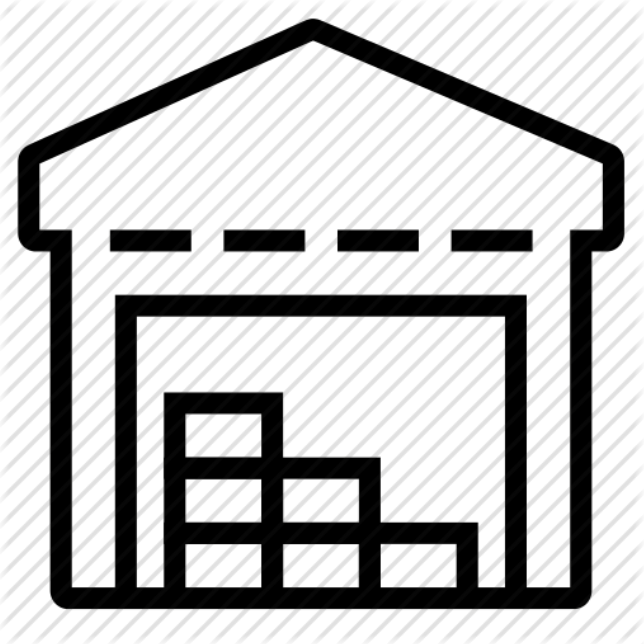
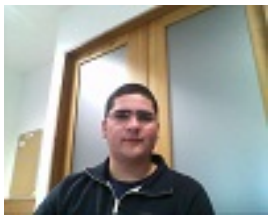
Sistemas Distribuidos

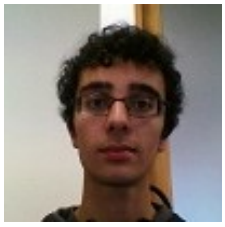


Warehouse

(Cliente - Servidor)


José Pereira(67680)


Rui Oliveira(a67661)


Tomás Ferreira(67701)


Jorge Ferreira(64293)

Índice

Anexos

Introdução

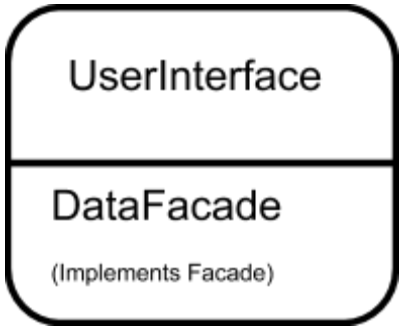
Neste relatório iremos apresentar nosso o projecto de Sistemas Distribuídos.

O objectivo deste projecto é desenvolver uma aplicação cliente-servidor que permita gerir um armazém através de requisições de ferramentas para desempenhar tarefas.
Os funcionários interagem através de um cliente intermediados por um servidor multi-threaded recorrendo a comunicação via TCP. Este servidor deverá disponibilizar para uso local todas as funcionalidades permitidas aos clientes.
Para além dos serviços de requisição e de devolução de ferramentas a aplicação deverá suportar registar novos utilizadores e listar as tarefas em curso.

Arquitetura Geral

O presente capítulo servirá como introdução geral à arquitetura da aplicação cliente-servidor. Esta é constituída pelas seguintes camadas: Interface do utilizador (UserInterface), Camada de comunicação (Client/ClientHandler) e camada de dados (DataFacade).
Para que a aplicação corresse no servidor, tal como corre no cliente. Quer a camada do Client/ClientHandler quer o DataFacade implementam uma interface Facade. Depois, quando é instanciada uma nova UserInterface tem que se dar um Facade, aqui decidimos se damos um Client ou um DataFacade, mediante seja no Cliente ou no Servidor.

(Camadas Cliente/Servidor)



(Camadas Servidor)

Interface Facade

O facade é uma interface onde se declaram todas as possíveis ações permitidas pela aplicação. Existem duas implentações do Facade: o Client para acesso remoto e o DataFacade para aceder diretamente aos dados.

Camada Interface

Esta camada é utilizada para os utilizadores interagirem com o cliente, caso se trate de um utilizador "normal", ou diretamente com o servidor, caso se trate de um administrador.
A cada utilizador é fornecida uma lista de opções, cada uma executando um comando diferente, estes comandos executam diversas tarefas, sendo que, no caso dos clientes, estas tarefas utilizam a camada de comunicação. Já no caso dos administradores, cada tarefa é executada diretamente no servidor, estes possuem acesso a todas as opções dadas aos utilizadores normais e a mais algumas exclusivas.

Camada Comunicação

Esta camada é usada quando o cliente é remoto. Do lado do cliente existe o Client que implementa o Facade, e do lado do servidor existe o ClientHandler. Cada ClientHandler atende um único Client. Estas duas classes comunicam entre si utilizando um protocolo que será explicado mais à frente , sobre um socket.
O ClientHandler ao ser instanciado recebe um Facade, que utiliza para reencaminhar os pedidos que interpretou. No atual cenário este Facade é do tipo DataFacade mas, caso fosse necessário, podia existir outra camada intermédia.

Camada Dados

A camada de dados é a camada que o servidor utiliza para fazer a operações sobre o armazém e também manter e organizar os dados tanto do armazém como dos utilizadores. A classe principal da camada de dados e a classe DataFacade, esta classe implementa a interface Facade que tem os métodos que são utilizados pelo utilizador e dois objetos: um objeto do tipo WareHouse e outro do tipo Clients.
A classe WareHouse é a classe que tem os dados sobre os itens armazenados e faz o controlo dos mesmo, enquanto a classe Clients é um HashMap de todos os clientes, esses clientes são representados pela classe Client que têm um HashMap das suas tarefas e um HashMap das suas tarefas em execução assim como o username e password.

Protocolo de Comunicação

O protocolo de comunicação utilizado entre o cliente e o servidor para comunicarem sobre o socket, é todo feito em texto. e segue o seguinte formato:

$$< codMessagem >,< N\ Atributos >,< Atributo_{1} >,< Atributo_{2} >,...,< Atributo_{N} >$$

Ou seja, tudo separado por virgulas, em primeiro lugar o codigo que identifica a mensagem (um inteiro único), seguido do número de atributos, e por fim os atributos que têm de corresponder ao número de atributos.

Cada ClientHandler e Client tem uma classe chamada *ComunicationSocket* que abstrai da escrita direta em forma de Strings. Este ComincationSocket tem métodos como sendMessage ou readMessage, e tratam da conversão dos objetos para Strings utilizando uma class chamada ComuncationSerializer que será explicada a seguir.

Sempre que o cliente envia uma mensagem, fica a escuta à espera da resposta. Mesmo que seja para saber se correu tudo como esperado, e caso contrario receber a exceção.

Serialização

A serialização é executada pela classe ComuncationSerializer, esta classe serializa arrays, inteiros, booleans e Objetos que implementem a interface Serializer.
A interface serializer permite que qualquer objeto possa ser passado para texto para ser enviado, e depois reconstruído na recepção. Esta interface declara os métodos *serialize*, *descerialize*, e por uma questão de operações, o *clone*.

As vírgulas e outros caracteres usados na serialização, são caracteres proibidos nos atributos para o bom funcionamento da aplicação.

Controlo de Concorrência

Na aplicação do lado do servidor existe situações em que as várias threads podem aceder aos mesmo dados ao mesmo tempo é preciso portanto fazer controlo de concorrência no acesso a esses dados. As classes que têm controlo de concorrência são, a classe WareHouse e Item, pois pode haver várias threads a aceder o armazém e/ou aceder ao mesmo item, e as classes Client e Clients que também têm controlêo de concorrência para o caso em que várias threads tentam aceder ao mesmo utilizador ao mesmo tempo.
Na classe **WareHouse** o controlo de concorrência é feito ao nível dos acessos do HashMap, ou seja, só pode haver a qualquer altura uma única thread a pesquisar, inserir ou remover no HashMap, este controlo é feito usando um ReentrantLock.
Na classe **Item** o controlo é feito quando temos de retirar ou inserir um novo item usando também um ReentrantLock, quando não existem mais itens é feito um wait usando uma Condition para que a thread fique em modo “sleep” até que haja itens, a thread é depois acordada com um signalAll que é feito durante a inserção de novos itens.
Na classe **Clients** existe um Map de utilizadores, que é protegido utilizando também um ReentratLock. Este lock é utilizando em todos os métodos como um monitor. Com a particularidade de no metodo *getCliente,* antes de fazer o retorno, fazer lock do cliente que irá retornar, devendo-se ter o respetivo cuidado para garantir que esse lock é desbloqueado.

A classe **Client** implementa a interface Lock utilizando para isso internamente um RentrantLock, e implementa também a interface AutoCoseable para permitir utilizar a sintaxe do java 7: *try with reource*. Isto é suficiente para o controlo de concorrência no facade utilizar:

try(Client c = clients.getClients()) { / statement */ }*

É garantido que quando sair de dentro do *try* irá ser executado o close que faz o unlock do client, que, tal como dito anteriormente, foi bloqueado pela classe Clients.

Controlo de Exceções

Foi criada um tipo de exceção chamado *SimpleExeption* para que o controlo das exceções foce mais simples.
Assim sempre que existe uma exceção no servidor essa exceção era convertida num SimpleExecption com os seguintes parâmetros: nível - representa o grau de gravidade da exceção, um código - identificador não único para se identificar o contexto, e a uma mensagem curta a explicar o contexto.
O grau de gravidade tem a seguinte escala: 1 - muito grave, 3 - pouco grave.
As exeções nivel 2, 3 são encaminhadas para o cliente na resposta, e o cliente mostrará ao utilizador
As exceções nível 1 provocam o encerramento da comunicação.

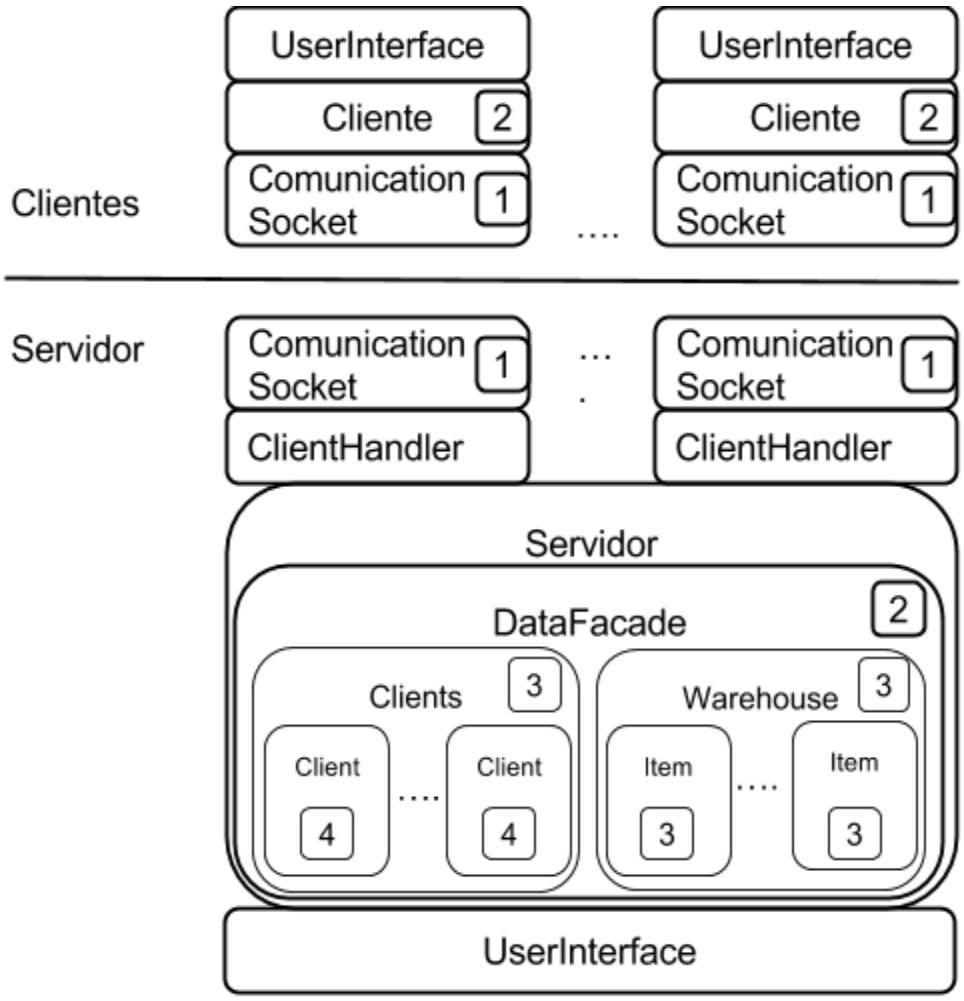
Conclusões

Neste trabalho continuamos o estudo das aulas práticas sobre a aplicação de multi-threading e comunicação cliente-servidor em programas desenvolvidos na linguagem java. No trabalho utilizamos técnicas de multi-threading para poder responder pedidos de vários clientes ao mesmo tempo, para isso cada cliente tem o seu socket de comunicação com o servidor. Para resolver os problemas de concorrência que o servidor multi-threading tive-se que detetar as áreas críticas e resolver tantos os problemas de múltiplos acessos como os de dead-lock. Este trabalho prático serviu como ponte para ligar a matéria das aulas praticas com aplicações práticas de sistemas distribuídos.
Por fim o nosso trabalho atingiu todos os objetivos referidos no enunciado, mas pensamos que ele pode ser sujeito a melhorias futuras.

Trabalhos futuros

Existem possíveis melhorias que podiam ser feitas neste trabalho. Começando por corrigir o problema de caracteres especiais como a virgula não possam passar por o socket possivelmente utilizando um caráter de escape tal como a shell de linux faz.
Implementar o nível de Zero que iria implicar o encerramento da aplicação do lado do servidor e por consequência de todos os clientes.
E implementar controlo para prevenir starvation, possivelmente utilizando uma queue ou um estrutura parecida.

Anexo 1: Esquema da aplicação



1- Usa ComunicationSerializer
2- Implementa Facade
3- Usa um lock como monitor
4- Implementa Interface Lock