Universidade do Minho

LEI 3ºAno 1ºSemestre

Sistemas Distribuídos

Warehouse

Daniel Caldas a67691
José Cortez a67716
Marcelo Gonçalves a67736
Ricardo Silva a67728

4 de Janeiro de 2015

Conteúdo

1 Estrutura do relatório **1**

2 Introdução **1**
2.1 Objetivos 1

3 Detalhes gerais de implementação **2**

4 Servidor e comunicação Cliente-Servidor **3**

5 Controlo de concorrência **4**

6 Interface do utilizador IU **5**

7 Conclusão **6**

1 Estrutura do relatório

Este relatório obedece à estrutura que se segue:

- Introdução e apresentação dos **objetivos**;
- Estrutura/organização do projeto, *detalhes gerais de implementação* (**diagrama de classes**);
- Como é feita a **comunicação** entre servidor e clientes;
- **Zonas críticas** do código no contexto do controlo de concorrência;
- Aspeto da interface do utilizador **IU**;
- Conclusão;

2 Introdução

Foi proposto ao grupo de trabalho o desenvolvimento na linguagem **JAVA de um Armazém que faz controlo de um stock de ferramentas**, às quais funcionários (*clientes*) **acedem concorrentemente** e usam no contexto de uma **tarefa pré-definida** que prentem realizar.

2.1 Objetivos

- Implementar um servidor em JAVA que faz o controlo dos funcionário e do armazém;
- Implementar um cliente em JAVA que aceda ao servidor via sockets (TCP) que possa abastecer o armazém, usar ferramentas, definir tarefas e realizar as mesmas;
- Controlar o acesso ao armazém e execução de tarefas utilizando mecanismos de concorrência que permitam ao servidor transmitir uma imagem de coerência e resposta a todos os clientes e os pedidos efetuados;

3 Detalhes gerais de implementação

Nesta secção queremos essencialmente mostar como funcionam as coisas *nos bastidores* do nosso código.
Podemos obter uma vistas geral sobre as classes que implementá-mos no diagrama de classes em baixo.

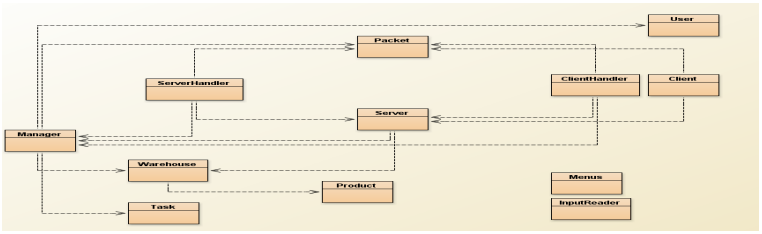

Figura 1: Diagrama de classes onde são visualmente explícitos as interações das diferentes classes do projetos.

De seguida fazemos um pequena e sucinta descrição de cada classe:

- **Cliente** representa um funcionário que interage com o servidor via socket;
- **ClientHandler** é classe que faz o tratamento de um cliente ligado ao servidor;
- **Manager** é a *"capital"* das classes, aqui guardamos a referência para utilizadores, tarefas e armazém. É onde são implementados os métodos que permitem o controlo de concorrência ao acesso desses recursos;
- **Packet** é um pequeno pacote de dados que serve de **estafeta**, encapsulando os dados que queremos transmitir entre *servidor-cliente* e vice-versa;
- **Product** é a classe que define uma ferramenta que é na essência a sua quantidade em stock;
- **Server**, classe onde é definido o servidor e onde são lançadas threads para atender aos diversos clientes;
- **ServerHandler** classe que faz o tratamento da interface do servidor;
- **Task**, classe que define um tipo de tarefa definida por um funcionário (*um cliente*);
- **User**, classe utilizador onde agrupamos os dados de uma entidade que interage com o servidor;
- **Warehouse**, armazém onde são guardadas as ferramentas e feito o controlo de acesso às mesmas;

4 Servidor e comunicação Cliente-Servidor

Para cumprir com o enunciado nós implementamos um servidor que também oferece uma interface de cliente, daí no método **public void start()** da classe **Server** como podemos ver na figura. Para comunicar entre *Servidor*

```
try {
    ServerHandler console = new ServerHandler(m);
    console.start();

    while(true){
        client = server.accept();
        ClientHandler thread = new ClientHandler(client,m);
        thread.start();
    }
} finally { server.close(); }
```

Figura 2: Pedaço de código do ficheiro *Server.java*

e *Cliente* usamos um pacote de dados um *HashMap<String,String>* que contém os dados de uma determinada operação a efetuar, operação essa que é definida pela **String action** que nos indica se o Cliente quer abastecer o armazém ou definir uma tarefa entre outras. Através do métod **void sendPacket(Packet p)** usamos o método **writeObject** para enviar através de um *socket* a ação definida pelo utilizador.

```
private void sendPacket(Packet p) throws IOException {
    ObjectOutputStream writer = getSocketOWriter();
    writer.writeObject(p);
    writer.flush();
}
```

Figura 3: Método **sendPacket** das classes **Cliente** e *ClientHandler*.

5 Controlo de concorrência

O principal objetivo deste trabalho é controlar o acesso ao armazém e execução de tarefas, de maneira a que o programa consiga lidar com a informação de vários clientes em simultâneo. Para isso, tivemos que utilizar mecanismos de controlo concorrência, que permitam ao servidor transmitir uma imagem de coerência em resposta a todos os clientes.

No código por nós desenvolvido, existem quatro classes, a Warehouse, a Product, a Task e a Manager, onde é armazenada a informação relevante relativa a cada utilizador, objetos e tarefas, informações essas que os utilizadores podem alterar à medida que utilizam a aplicação. Para garantir que esta informação está sempre atualizada quando é pretendida por cada utilizador, implementamos um sistema que faz com que só um utilizador possa aceder a informação partilhada por todos os utilizadores de cada vez, devido ao facto dessa informação tratar-se da secção crítica do nosso código.

Sendo assim, na classe Warehouse, onde são armazenados os objetos, temos um lock, que bloqueia o acesso aos outros utilizadores quando já está um a aceder aos objetos. Como por vezes é necessário alertar utilizadores quando um objeto fica disponível, temos associado a esse lock na classe Product, uma condição para cada produto do armazém.

Na classe Manager, onde é se encontra o armazém e as tarefas, utilizamos a mesma técnica. Temos um lock, que bloqueia o acesso aos outros utilizadores quando já está um a aceder à informação da classe. Como por vezes é necessário alertar utilizadores quando termina uma tarefa, temos associado a esse lock na classe Task, uma condição para cada tarefa.

```
public boolean endTask(String id) throws InterruptedException {
    Map<String,Integer> objectsToSupply;
    String type;
    lock.lock();
    try{
        if(!this.tasksRunning.containsKey(Integer.valueOf(id))){
            return false;
        } else{
            type = this.tasksRunning.get(Integer.valueOf(id));
            objectsToSupply = this.tasks.get(type).getObjects();
        }
    } finally{ lock.unlock(); }

    // Supply de todos os objetos
    for(Map.Entry<String,Integer> entry : objectsToSupply.entrySet()){
        warehouse.supply(entry.getKey(),entry.getValue());
    }

    lock.lock();
    try{
        this.tasksRunning.remove(Integer.valueOf(id));
        this.tasks.get(type).signalP();
    } finally{ lock.unlock(); }

    return true;
}
```

Figura 4: Exemplo da utilização de Lock's

Em suma, o programa por nós desenvolvido não deixa que dois utilizadores acedam a informação partilhada em simultâneo, permitindo assim que a informação esteja coerente no momento em que é consultada.

6 Interface do utilizador IU

Pelas seguintes imagens podemos observar claramente como se comporta a aplicação de ambos os lados, Servidor e Cliente.
Na primeira imagem temos as opções disponibilizadas pelo **menu principal** do servido, na segunda imagem em que a temos **sessão iniciada** com o o utilizador de *username* **utilizador3000**, podemos ver as operações que o servidor disponibiliza a todos os clientes, desde abastecer o armazém (opção 1) até listar todo o stock presente no armazém (opção 8).

```
######################## Menu: Servidor ########################
#                                                              #
        1 - Registar
        2 - Entrar
        0 - Sair
#                                                              #
################################################################
Opção:
```

Figura 5: Menu principal do servidor.

```
######################## Menu: utilizador3000 ################
#                                                              #
        1 - Abastecer armazém
        2 - Definir nova tarefa
        3 - Iniciar tarefa
        4 - Concluir tarefa
        5 - Esperar por conclusão de tarefas
        6 - Listar tarefas
        7 - Listar tarefas a decorrer
        8 - Listar Stock
        0 - Sair
#                                                              #
################################################################
Opção:
```

Figura 6: Painel de operações disponíveis na consola para um utilizador.

7 Conclusão

Após a implementação do projeto Warehouse ficamos familiarizados com os conceitos introdutórios do controlo de concorrência, de como estes são parte crucial de qualquer sistema distribuído atual, sobretudo tivemos contacto com:

- Problemas que a complexidade destas implementações levantam;
- Vantagens e importância que trazem aos sistemas de informação atuais e onde podemos encontrá-los no dia à dia;
- Técnicas rotina para contornar os problemas frequentes que surgem quando usamos mecanismos de concorrência como **locks** e **variáveis de condição**;