Universidade do Minho
Licenciatura em Engenharia Informática

Trabalho Prático Sistemas Distribuídos: Gestor Requisições/Tarefas

Autores:

Bruno Vilaça
a55853@alunos.uminho.pt

Pedro Silva
a64345@alunos.uminho.pt

Domingo 4, Janeiro 2015

Conteúdo

Introdução

Este relatório descreve o processo de conceção de um gestor de requisições/tarefas no âmbito do trabalho prático da Unidade Curricular Sistemas Distribuídos.

O objetivo deste trabalho consiste no desenho e implementação de um serviço que permite a gestão de um armazém de forma eficiente. Os utilizadores interagem com recurso a um cliente escrito em Java, contudo intermediados por um servidor multi-threaded em que a comunicação ocorre via TCP.

Servidor

Estrutura

No seu funcionamento o servidor está dividido em duas partes, a primeira, o servidor em si, que aceita as ligações de clientes remotos e uma segunda parte que consiste num cliente local. Existe ainda a inicialização das estruturas e a persistência de dados do servidor, que não serão aprofundados neste relatório.

ConnectionsHandler

Responsável por instanciar o servidor na porta requerida e estabelecer ligação com os clientes remotos.

```
public void run() {
    ServerSocket ss;
    try {
        ss = new ServerSocket(port);
        System.out.println("Server initialized on port : " + port);

        while (true) {
            Socket sc = ss.accept();
            try {
                new ClientHandler(sc, users, manager, logger).start();
            } catch (Exception e1) {
                System.out.println(e1.getMessage());
            }

        }

    } catch (IOException e2) {
        System.out.println(e2.getMessage());
    }
}
```

ClientHandler

A Classe ***ClientHandler*** é a responsável por toda a interação entre cliente e servidor.

```
while (run) {

    InputStreamReader ir = new InputStreamReader(sc.getInputStream());
    BufferedReader br = new BufferedReader(ir);
    PrintWriter pw = new PrintWriter(sc.getOutputStream());

    String s = br.readLine();
    if (s == null) {
        break;
    }

    String response = skeleton.parseMessage(s);

    pw.println(response);
    pw.flush();
}
sc.close();
String r = skeleton.parseMessage("logout:" + username);
//System.err.println("Client " + username + " on : " + address + " disc
```

- O ***ClientHandler*** recebe a mensagem do *Socket* do cliente entregando-a ao ***skeleton*** para ser feito o *parsing* da mesma e consequentemente a execução da mesma no servidor. Depois fica á espera de uma resposta do servidor para ser encaminhada para o cliente.

LocalClient

O Cliente local é apenas uma interface com utilizador em forma de texto que permite aceder sem recurso a ***sockets***, como fazem os clientes remotos, às estruturas partilhadas do servidor.

Estruturas Partilhadas

Manager

A estrutura *manager* é uma abstração a 3 outras estruturas partilhadas, um inventário de objetos (*WareHouse*), uma coleção de tipos de tarefas existentes no sistema e uma lista de tarefas ativas.

```
public class Manager implements ManagerInterface {

    WareHouse wh;
    int nextId = 0;

    HashMap<String, TaskType> taskTypes;
    HashMap<Integer, Task> activeTasks;
```

A estrutura ***wh*** (WareHouse) é a responsável pela gestão dos Objetos no sistema.

```
public class WareHouse {

    private final Lock inv_lock = new ReentrantLock();
    private HashMap<String, Tool> inventory;
```

Users

A estrutura *Users* contem a informação sobre os utilizadores do sistema, mais concretamente utilizadores registados e utilizadores ativos.

```
public class Users implements UsersInterface {

    private final HashMap<String, User> users;
    private final List<String> logged;
```

Controlo de Concorrência

Como um dos Objetivos deste Trabalho Prático era desenvolver um Servidor capaz de atender pedidos de vários clientes ao mesmo tempo, foi necessário implementar controlo de

concorrência nas estruturas partilhadas. Isto permite que o estado do servidor nunca fique comprometido e que a informação acedida pelos clientes se mantenha consistente.

Funcionalidades

Para as diferentes funcionalidades foi necessário implementar controlo de concorrência. Isso foi feito com recurso a ***Locks*** explícitos, para manter a coesão das diferentes estruturas e permitir que o resto de servidor não fique em espera quando se trabalha com uma única estrutura.

Requisitar Objetos/ Inicio de Tarefa

Para esta funcionalidade foi necessário implementar o controlo de concorrência em quatro sítios diferentes:

- Primeiramente ter acesso exclusivo ao contador de identificador, isto é importante na medida em que permite manter sempre uma ordem na requisição dos Objetos/ Tarefas.

```
clock();
try {
    nextId++;
    id = nextId;
} finally {
    cunlock();
}
```

- De seguida foi necessário ter o acesso exclusivo aos tipos de tarefas, para obter o tipo de trefa pretendido.

```
lockTskType();
try {
    t = taskTypes.get(taskType);
} finally {
    unlockTskType();
}
```

- Em terceiro lugar foi necessário assegurar que aquando da requisição de material no inventário, mais ninguém poderia ter acesso ao mesmo. E no caso de não haver disponibilidade de um dos objetos para tal tarefa ficar bloqueado à espera que este fiquem todos disponíveis ao mesmo tempo.

```
lockInv();
try {
    Tool t;
    boolean notready = true;
    while (notready) {
        notready = false;
        for (String s : tools.keySet()) {
            t = getTool(s);

            if (tools.get(s) > t.qtd()) {
                notready = true;
                t.await();
                break;
            }
        }
    }
    for (String s : tools.keySet()) {
        t = getTool(s);
        t.dec(tools.get(s));
    }
} finally {
    unlockInv();
}
```

- Por fim bloqueou-se a estrutura para guardar a instância desta tarefa.

```
lockTsk();
try {
    this.activeTasks.put(id, task);
} finally {
    unlockTsk();
}
```

Devolução de objetos/Conclusão de tarefa

Esta funcionalidade para além de ter acesso exclusivo as estruturas já referidas, era necessário notificar ***threads*** em espera na requisição de material e/ou em espera da terminação da tarefa em causa. Isso foi feito com recurso a variáveis de condição e ao método ***signalAll ().***

- Notificar ***threads*** em espera de material:

```
lockInv();
try {
    Tool t;
    for (String s : tools.keySet()) {
        t = getTool(s);
        if (t.is_returnable()) {
            t.inc(tools.get(s));
            t.signalAll();
        }
    }
} finally {
    unlockInv();
}
```

- Notificar ***threads*** em espera que a tarefa termine:

```
lockTsk();
try {
    Task ta = activeTasks.get(task_id);
    if(!ta.getUsername().equals(username) && !username.equals("logger")) throw new WrongUserException();

    activeTasks.remove(task_id);
    ta.signalAll();
} finally {
    unlockTsk();
}
```

É importante referir que também a funcionalidade de abastecer o armazém notifica as ***threads*** em espera por material, recorrendo às mesmas técnicas utlizadas acima.

Outras Funcionalidades Implementadas

- Registar Utilizador
- Login Utilizador
- Abastecer Armazém
- Definir Tipos de Tarefas
- Esperar pela Conclusão de Tarefas
- Listar Tipo / Tarefas Ativas

Cliente

Para a comunicação entre Servidor e Cliente implementou-se um Protocolo RPC (*Remote Procedure Call*), tendo assim o cliente acesso aos métodos existentes no servidor a partir de Interfaces partilhadas por ambos.

O Cliente tem como interface um menu textual, em que os utilizadores utilizam comandos via texto, que depois são processados pelos diferentes métodos até ser enviado ao servidor uma mensagem.

Parser

A classe ***Parser*** é responsável por tratar do ***input*** do utilizador e da resposta ao mesmo. Ao receber o ***input,*** o ***Parser*** chama o método adequado ao ***Stub***, esses métodos são disponibilizados pela interface referida acima.

Segue um pequeno excerto do método ***parseAndCall*** da classe ***Parser***:

```
case "completed":
    try {
        stub.task_return(Integer.parseInt(message[1]));
    } catch (TaskNotFoundException ex) {
        returnStr = "Task not found!";
    } catch (IOException ex) {
        returnStr = "Connection lost!";
    } catch (WrongUserException ex) {
        returnStr = "Task belongs to other user!";
    }
    break;
case "request":
    try {
        returnStr = "Task ID: " + Integer.toString(stub.task_request(message[1], this.user));
        if(returnStr.equals("-1"))
            returnStr = "Task not available!";
    } catch (InterruptedException ex) {
        ex.printStackTrace();
    } catch (IOException ex) {
        returnStr = "Connection lost!";
    }
    break;
```

Stub

O ***Stub*** é responsável por estabelecer a comunicação com o servidor, enviar e receber mensagens do mesmo. E também responsável por fazer o ***marshalling*** da mensagem a ser enviada e o ***unmarshalling*** da resposta.

```
public WarehouseStub(String host, int port) throws IOException {
    this.socket = new Socket(host, port);
    this.in = new BufferedReader(new InputStreamReader(socket.getInputStream()));
    this.out = new PrintWriter(socket.getOutputStream());
}
```

A ligação ao servidor é feita logo na instanciação do ***Stub***.

Num primeiro momento apenas é permitido ao cliente fazer *login* ou registar um utilizador. Só apos estar com *login* efetuado, o cliente pode executar outras ***queries*** ao servidor.

Conclusão

A topologia ***multi-threaded*** do servidor permite atender pedidos de vários clientes ao mesmo tempo. Isso levantou problemas de concorrência que tiveram que ser resolvido com recurso a exclusão mutua no acesso as estruturas partilhadas do servidor.

Ainda relativamente à concorrência no servidor foi tido em atenção situações de ***deadlock***. Em termos de ***starvation*** o controlo está no ***cpu*** da máquina do servidor e por isso fora do alcance do desenvolvimento do sistema.

Apesar de não ser requerido, foi ainda implementado uma função de persistência de dados, mais concretamente persistência do estado do servidor, que é restaurado assim que o servidor é inicializado e atualizado sempre que um cliente executa uma ação que altera esse mesmo estado.