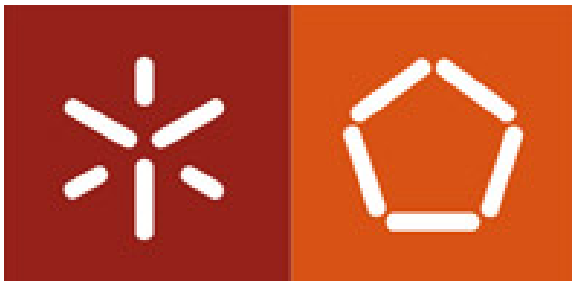


Universidade do Minho

Mestrado Integrado em Engenharia Biomédica


# Aplicações Distribuídas


**Trabalho realizado por:**

**Fábio França 61785**

**Filipe Fernandes 61754**

**Docente:**

**Professor António Sousa**


**Dezembro de 2013**

**Resumo**

No âmbito da Unidade Curricular de Aplicações Distribuídas ,desenvolveu-se um trabalho prático com diferentes objetivos. Numa primeira fase, permitiu uma maior familiarização com a linguagem de programação JAVA, mais especificamente com estruturas de dados, arquitetura cliente/servidor utilizando JAVA RMI e controlo de concorrência.

Com o presente trabalho prático é desenvolvido um Centro Hospitalar no qual se pretende criar um sitema integrado de gestão de atos médicos, consultas, atos de enfermagem e gestão de farmácia hospitalar.

Ao longo deste relatório serão especificados e justificados todos os elementos do nosso trabalho, desde a estrutura da base de dados a arquitetura utilizada.

# Conteúdo

# 1 Introdução

Com o objetivo de complementar todos os requisitos necessários para a execução deste trabalho recorreu-se ao apoio da ferramenta de programação Netbeans, devido à sua versatilidade e familiaridade com a ferramenta por parte dos elementos do grupo.

Foi proposto que se elaborasse um programa em JAVA capaz de lidar com diferentes problemáticas ocorrentes a nível hospitalar, ultrapassando problemas de concorrência e capaz de criar, verificar e controlar estes diferentes fatores.

Além do sistema de gestão implementado foi ainda incentivada a aquisição de estatísticas decorrentes da Administração Hospitalar.

A descrição das entidades, interfaces e estruturas de dados utilizadas bem como a arquitectura proposta encontra-se desenvolvida neste relatório.

# 2 Descrição do Trabalho Prático e Análise de Resultados

## 2.1 Descrição do Trabalho

### 2.1.1 Arquitetura RMI

Com base no enunciado fornecido pretendeu-se criar uma arquitetura cliente/servidor RMI. O RMI (Remote Method Invocation) é uma tecnologia suportada pela plataforma Java que facilita o desenvolvimento de aplicações distribuídas. Como o próprio nome indica, o RMI permite ao programador invocar métodos de objectos remotos, ou seja, que estão alojados em máquinas virtuais Java distintas, duma forma muito semelhante às invocações a objectos locais. De certa forma, à custa de algum esforço adicional de engenharia de software, o programador pode desenvolver aplicações totalmente distribuídas como se de aplicações locais se tratassem, sendo quase toda a comunicação entre máquinas virtuais Java assegurada transparentemente pelo próprio RMI.

Evidentemente, há algumas diferenças fundamentais entre uma aplicação distribuída e uma aplicação não-distribuída. A principal é a de que numa aplicação distribuída há uma infra-estrutura de comunicação subjacente que impõe um conjunto de condicionalismos à própria aplicação: risco de falhas de comunicação, latências variáveis, limites de banda, falha de servidores, entre outros aspectos. O RMI permite lidar com de forma facilitada com algumas destas contingências.

Uma noção central ao RMI é o da separação entre interface e implementação de uma classe.A particularidade mais relevante desta separação é que o RMI permite que a interface e a respectiva implementação se localizem em JVM's diferentes. O RMI torna possível que uma determinada aplicação cliente adquira uma interface (que define o comportamento) referente a uma classe (que contém a implementação) que corre numa JVM diferente. Relembre-se que uma interface em Java não possui código de implementação o que significa que neste caso toda a computação subjacente à interface corre numa JVM distinta. A comunicação entre interface e a implementação é assegurada pelo RMI recorrendo a TCP/IP.

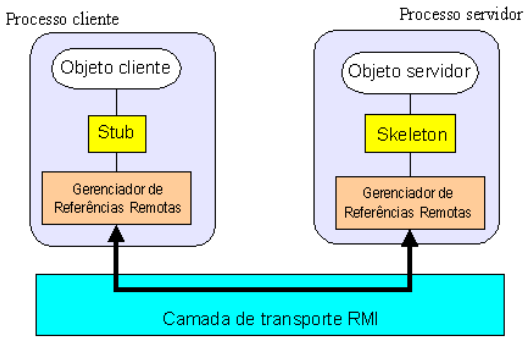


Figura 1: Arquitetura Cliente/Servidor RMI

Uma vez que a interface remota esteja definida e a classe que implementa o serviço remoto tenha sido criada, o próximo passo no desenvolvimento da aplicação distribuída é desenvolver o servidor RMI, uma classe que crie o objeto que implementa o serviço e cadastre esse serviço na plataforma de objetos distribuídos. O desenvolvimento de um cliente RMI requer essencialmente a obtenção de uma referência remota para o objeto que implementa o serviço, o que ocorre através do cadastro realizado pelo servidor. Uma vez obtida essa referência, a operação com o objeto remoto é indistinguível da operação com um objeto local. Para que um serviço oferecido por um objeto possa ser acedido remotamente através de RMI, são precisas as classes auxiliares internas de stubs e skeletons, responsáveis pela comunicação entre o objeto cliente e o objeto que implementa o serviço, conforme descrito na apresentação da arquitetura RMI.

### 2.1.2 Entidades e Interfaces da aplicação

Após a criacão da arquitetura cliente/servidor desenvolveu-se diferentes entidades, mais especificamente, médico, enfermeiro, gestor, utente, receita, medicamento, produto, funcionário, consulta, ato de enfermagem, encomendas de produtos e medicamentos e farmácia. Importa assim referir que cada uma destas interfaces apresenta a sua consequente implementação.Uma interface pode ser vista como um protocolo de comportamento. Ela é simplesmente uma lista de métodos abstratos, podendo também incluir variáveis. Para utilizá-la, é criada uma classe que implemente-a. Esta classe será obrigada a definir todos métodos das interfaces que está implementando. Um exemplo, utilizado neste trabalho é a interface Médicos e correspondente implementação Médicos_Impl.

```
package Interface;

import java.rmi.Remote;
import java.rmi.RemoteException;


public interface Medicos extends Remote {
    public String getMoradaMedico() throws RemoteException;

    public void setMoradaMedico(String morada) throws RemoteException;

    public void setNomeMedico(String nome ) throws RemoteException;

    public String getNomeMedico() throws RemoteException;

    public String getdataMedico() throws RemoteException;

    public void setdataMedico() throws RemoteException;
```

```
    public String getEspecialidade() throws RemoteException;

    public void setEspecialidade(String especialidade) throws RemoteException;


}


public class MedicosImpl extends UnicastRemoteObject implements  Medicos,
Serializable {

 private String nome ;
    private String morada;
    private String especialidade ;
    private HashSet<Utentes> utentes = new HashSet<Utentes>();
    private HashMap<Integer,Medicamentos> medicamentos =
new HashMap<Integer,Medicamentos>();
    private HashSet<Consulta> consultas = new HashSet<Consulta>();
    private Date data;

    public Medicos_Impl(String nome, String morada,String especialidade)
 throws RemoteException{
        this.especialidade = especialidade;
        this.medicamentos = new HashMap<Integer,Medicamentos>();
        this.morada = morada;
        this.nome = nome;
        this.utentes = new HashSet<Utentes>();

}
    public String getMoradaMedico() throws RemoteException{
        return this.morada;
    }


    public void setMoradaMedico( String morada) throws RemoteException {
        this.morada = morada;
    }


    public void setNomeMedico(String nome) throws RemoteException {
       this.nome= nome;
    }
```

```
    public String getNomeMedico() throws RemoteException {
       return this.nome;
    }


    public String getdataMedico() throws RemoteException {
        return this.data.toString();
    }

    public void setdataMedico() throws RemoteException {
        System.out.println("Introduza um dia(dd) ,depois o Mes(mm) e o ano(yyyy) ");
        Scanner s = new Scanner(System.in);
        int dia = s.nextInt();
        int mes = s.nextInt();
        int ano = s.nextInt();
        Date d = new Date(ano-1900,mes-1,dia);
        this.data= d;
    }


    public String getEspecialidade() throws RemoteException {
        return this.especialidade;
    }


    public void setEspecialidade(String especialidade) throws RemoteException {
       this.especialidade = especialidade;
    }

}
```

Tal como referenciado, interfaces Java são especificações de tipos de dados. Estas podem ser vistas como o conjunto de operações que um objeto pode realizar, e que são definidas como acessíveis ou invocáveis do exterior. Especificam portanto, o conjunto de operações que podem ser realizadas com as instâncias de um certo tipo de dados. As classes *Impl* tal como o próprio nome indica, implementam uma determinada interface. A implementção GestorImpl.java possui a capacidade de gerir todas as entidades envolvidas no processo, desde a criação de médicos, consultas, utentes e receitas. Além da criação de novos processos o gestor possui a capacidade de consultar diferentes dados e estatísticas.

Neste tipo de classe, como por exemplo *MedicoImpl* existe um construtor da class, que é um bloco declarado com o mesmo nome da classe e sem valor de retorno, automaticamente "chamado"sempre que um novo objeto é criado. O construtor é geralmente

responsável pela alocação de recursos necessários ao funcionamento do objeto além da definição inicial das variáveis de estado. Além do construtor existem métodos dentro deste tipo de classe que permitem trabalhar o objeto e obter informações deste.

Os métodos do tipo *get* permitem obter informação acerca de um objeto anteriormente criado. Um exemplo será o método getEspecialidade, que posteriormente permitirá ao gestor obter a especialidade de um Médico dado o seu ID. Os métodos do tipo *set* permitem por outro lado, inserir um determinado valor. Desta forma, um exemplo deste tipo de método será setNomeMedico, que mais uma vez permitirá ao gestor introduzir o Nome de determinado médico caso este ainda não se encontre na "Base de dados"ou alterá-lo caso já se encontre criado.

Face às suas potencialidades, podemos considerar que a classe *GestorImpl* funciona como um motor de todo o algoritmo. Este, tal como referenciado, recorre aos diferentes métodos criados nas diversas interfaces e implementações, para dotar o programa com métodos de criação e consulta de processos. Por forma, a melhor se compreender o seu funcionamento desta classe, apresenta-se o método cria_consultaMedica.

```
@Override
    public String cria_consultaMedica(int idconsulta, int idmedico, int idutente)
throws RemoteException {
        if (!medicos.containsKey(idmedico)) {
            return "Médico Inexistente";
        } else if (consultas.containsKey(idconsulta)) {
            consultas.get(idconsulta);
            return "Consulta já marcada";
        } else if (!utentes.containsKey(idutente)) {
            return "Utente Inexistente";

        } else {
            consultas.get(idconsulta).setidutente(idutente);
            consultas.get(idconsulta).setmedico(idmedico);
            return "Consulta Marcada Com Sucesso";
        }
    }
```

Tal como evidenciado no código acima apresentado, o método *criaconsultamedica* faz uso de métodos *get* e *set* provenientes de diferentes interfaces. Desta forma será criada uma nova consulta médica caso exista determinado Médico e Utente e se tal consulta ainda não tiver sido criada.

Na implementação do programa em Java foi utilizada uma estrutura de dados denominada *HashMap* extremamente útil, uma vez que permite associar chaves de pesquisa a determinados valores. As tabelas de hash são um tipo de estruturação para o armazenamento de informação, de uma forma extremamente simples, fácil de se implementar e intuitiva de se organizar grandes quantidades de dados. Possui como ideia central a divisão de um universo de dados a ser organizado em subconjuntos mais gerenciáveis. A estruturação da informação em tabelas de hash, visa principalmente permitir armazenar e procurar rapidamente grande quantidade de dados.

Face às suas características e objetivos do Trabalho prático, foi ampla a utilização de HashMap uma vez que permitiu a organização de dados de forma simples e com facilidade de procura. Um exemplo de utilização de HashMap encontra-se no método *criaMedico*.

```
@Override
    public String cria_medico( int idmedico,String nome, String morada,
String especialidade) throws RemoteException {

 if (medicos.containsKey(idmedico)) {
            return "Médico com esse id já existe";
        } else {
            Medicos m = new Medicos_Impl();
            medicos.put(idmedico, m);
            medicos.get(idmedico).setNomeMedico(nome);
            medicos.get(idmedico).setMoradaMedico(morada);
            medicos.get(idmedico).setEspecialidade(especialidade);
            medicos.get(idmedico).setdataMedico();

            return "Médico adicionado com sucesso";
        }
    }
```

Como observável, a utilização de HashMap, está presente nos diversos métodos implementados pelo Gestor. No caso específico acima apresentado é utilizado o HashMap medicos, através do qual se insere os valores nome, morada, especialidade e data de um novo Médico a ser criado.

### 2.1.3 Criação e Obtenção de Dados

Nesta secção pretende-se explicar um dos objetivos do Trabalho prático que consiste na automatização do processo de carregamento e manipulação de dados para a aplicação. Por forma a concretizar este objetivo foram criadas inúmeras classes, desde a criação e actualização de uma dada entidade, a consulta de informação estatística necessária à Administração Hospitalar.

Uma das classes implementadas foi a classe *CriarEditarMedico* da qual se apresenta uma porção.

```
public class CriarEditarMedico {

 public static void main(String args[]) throws RemoteException, NotBoundException {
            Gestor g;
            Medicos m;
            Registry registry = LocateRegistry.getRegistry("127.0.0.1", 1099);
            g = (Gestor) registry.lookup("GestaoUtentes");
            m = (Medicos) registry.lookup("Medicos");
        int x = 0;

 while (x != 4) {
     System.out.printf("Medico:\nIntroduza o numero da operação que deseja:
              \n1- Adicionar\n2-Editar\n3-Consultar\n4-Sair\n");
      Scanner s = new Scanner(System.in);
       x= s.nextInt();
       if (x == 1) {
         System.out.println("Insira o id do Medico");
         int id = s.nextInt();
         System.out.println("Insira o nome do Utente");
         String nome = s.next();
         System.out.printf("Insira a morada do Utente\n\n");
         String morada = s.next();
         System.out.println("Especialidade");
         String especialidade = s.next();
         System.out.println("Introduza dia(dd) mes(mm)
         e ano(yyyy) de nascimento");
          int dia = s.nextInt();
          int mes = s.nextInt();
          int ano = s.nextInt();
String resultado = g.cria_medico(id, nome, morada,especialidade,dia,mes,ano);
                System.out.printf(resultado+"\n\n");
            }
            else if (x == 2) {
                System.out.printf("Indique a Informação que quer Alterar
                    \n1-Morada\n2-Nome\n3-Especialidade");
                int j = s.nextInt();
                if (j == 1) {
                    System.out.println("Insira o id do Medico");
                    int id = s.nextInt();
                    System.out.println("Insira a morada do Medico");
                    String morada = s.nextLine();
                    System.out.println(g.ins_moradaMedico(id, morada));
```

Apesar de não ser apresentado a totalidade do código correspondente à classe *CriarEditarMedico*, este excerto permite evidenciar o funcionamento da arquitetura cliente/servidor RMI. Por forma a fornecer um serviço remoto, o servidor cria um objeto que implementa a interface de serviço e que posteriormente será exportada para o sistema RMI. Este, cria um serviço que escuta por conexões e invocações a métodos do objeto. O objeto então é registado pelo servidor no RMI Registry com um nome público.Pelo cliente, o RMI Registry é consultado para descobrir a referência ao objecto por nome do serviço.

Além da implementação da arquitetura cliente/servidor RMI é visível o código que permite a criação de um novo Médico bem como a alteração de morada para um já existente. Para o primeiro caso, este é estabelecido recorrendo ao método *criaMedico* implementado na classe *GestorImpl*. No segundo caso apenas a morada é alterada sendo utilizado o método *insMoradaMedico*, também ele logicamente presente na classe *GestorImpl*.

Tal como pretendido na realização do trabalho prático, a Administração Hospitalar necessita de um vasto conjunto de dados e informação estatística. Alguns exemplos implementados são:

- Quais os medicamentos que se encontram abaixo do stock mínimo;
- Quais os actos de enfermagem agendados;
- Para cada medicamento qual o médico que mais o prescreve.

Uma vez mais, a estrutura de dados HashMap foi de extrema utilidade, uma vez que permitiu a organização da informação implementada no sistema. Assim sendo, foi facilitada a aquisição de dados necessários à Administração Hospitalar, como apresentado no método *consprevIDMEDICO*.

```
public HashSet<String> consprevIDMEDICO() throws RemoteException {

    HashSet<String> cons = new HashSet<>();
    Set<Integer> x = new HashSet<>();
    x.addAll(consultas.keySet());
    Iterator<Integer> it = x.iterator();
    while(it.hasNext()){
        int a = it.next();
        cons.add(("ID Médico : "+medicos.get(consultas.get(a).getidmedico())+
            " Nome: "+medicos.get(consultas.get(a).getidmedico())
            .getNomeMedico()+
            " Data :"+ consultas.get(a).getData().toString() ));
        }
        return cons;
    }
```

Utilzando este método, o programa é capaz de encontrar as consultas previstas para um Médico dado o seu ID.

Tendo em conta uma melhor estruturação do programa foram criados 4 principais serviços:

**-Consultas Médicas**

- Marcação de consultas( Médicos atendem em intervalos de 1 hora);
- Passar receita, sendo que esta fica atribuída ao utente;
- Consultar as consultas previstas;
- Consultar consultas por data;

**-Criar, Editar e Consultar dados de Entidades**

- Médicos;
- Utentes;
- Enfermeiros;
- Funcionários;

**-Enfermaria**

- Marcação de Acto de Enfermagem;
- Consulta de Actos agendados;
- Adicionar gastos a um Acto de Enfermagem(Medicamentos/Produtos;

**-Farmácia**

- Consultar Stock;
- Encomendar Medicamentos e Produtos(Encomenda tem prazo de entrega previsto de 7 dias);
- Levantamento de receita;
  Levantamento este realizado através da identificação do utente. Uma vez atingido o stock mínimo de um medicamento, será sugerida a encomenda do mesmo;
- Receber encomenda( Adicionar ao Stock);
- Consultar encomendas previstas;
- Consultar medicamentos/produtos abaixo do Stock mínimo.

### 2.1.4 Controlo de Concorrência

Uma problemáticas inseridas na realização do trabalho prático é o controlo de concorrência. Se bem utilizado, o paralelismo resulta num melhor desempenho de programas. No entanto o acesso concorrente a dados e recursos pode gerar problemas, uma vez que dados estes se podem tornar inconsistentes ao serem acessados concorrentemente.

Os mecanismos e Controlo de concorrência limitam o acesso concorrente a dados e recursos compartilhados, evitando inconsistências nos dados causadas pelo acesso concorrente. Alguns mecanismos de Controle de Concorrência disponíveis em Java são:

- Monitor - protege trechos de código/métodos que manipulam dados/recursos compartilhados, impedindo o acesso concorrente;
- Lock (ou Mutex) - cria uma fila de acesso a um dado compartilhado, impedindo o acesso concorrente;
- Semáforo - limita o número de utilizadores que acessam simultaneamente um recurso, criando filas de acesso se este número for excedido.

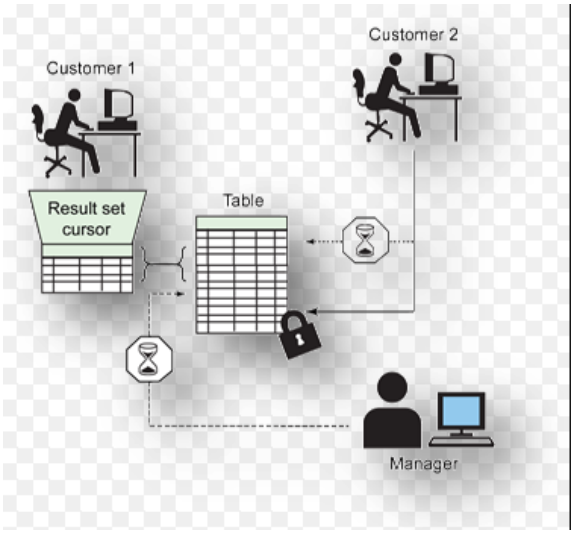


Figura 2: Controlo de Concorrência

Na implementação de controlo de concorrência no trabalho prático, utilizou-se sincronização a diferentes niveis. No primeiro caso, apenas se "sicncronizou "a *HashMap*, por forma a limitar o acesso ao *HashMap* a um cliente de cada vez.

```
public int getidMedico() throws RemoteException {
        int id_medico = 0;
        if (medicos.isEmpty()) {
            synchronized(medicos){
            id_medico =  1;
```

```
            }
        } else {
            synchronized(medicos){
                id_medico = medicos.size() + 1;
            }

        }
        return id_medico;
    }
```

No segundo caso, a sincronização é exercida ao nível dos métodos e não apenas em *HashMap*. No caso específico do método *criaConsultaMedica*, uma vez que este implica a verificação de Agendas, Médicos disponíveis e inserção de dados em diversos objetos, considerou-se satisfatório a sua sincronização. Desta forma, garante-se a criação à vez de cada consulta n a possibilidade de existir mais do que um cliente. Apesar de aumentar um pouco o tempo de resposta da aplicação, garante-se uma maior eficiência, evitando inconsistências nos dados causadas pelo acesso concorrente.

```
public synchronized  String cria_consultaMedica
(int idmedico, int idutente, int dia, int mes,
int ano, int hora) throws RemoteException {
        int idconsulta = getidConsulta();
        if (!medicos.containsKey(idmedico)) {
            return "Médico Inexistente";
        } else if (consultas.containsKey(idconsulta)) {
            consultas.get(idconsulta);
            return "Consulta já marcada";
        } else if (!utentes.containsKey(idutente)) {
            return "Utente Inexistente";

        } else {

            Date horacons = new Date(ano - 1900, mes - 1, dia, hora, 00);
            horasconsultas.put(idmedico, inserir_hora(idmedico, horacons));

            if ((confirma_hora(idmedico, horacons) == false) ||
            ((horacons.getHours() + 1) >= 22 && horacons.getHours() <= 9)) {
                horasconsultas.put(idmedico, remover_hora(idmedico));

                return "Hora indisponivel para o medico";
            } else {
                Consulta c = new Consulta_Impl();
                consultas.put(idconsulta, c);
                System.out.println("Introduziu dados");
                consultas.get(idconsulta).setidutente(idutente);
                System.out.println("id utente");
                consultas.get(idconsulta).setmedico(idmedico);
                consultas.get(idconsulta).setEspecialidade
                (medicos.get(idmedico).getEspecialidade());
                consultas.get(idconsulta).setdata(dia, mes, ano, hora);
                medicos.get(idmedico).setConsulta(idconsulta);
                consespec.put(idconsulta, medicos.get(idmedico)
```

```
                .getEspecialidade());
                utentes.get(idutente).setConsulta(idconsulta);

                return "Consulta Marcada Com Sucesso";
            }

        }

    }
```

## 2.2 Análise de Resultados e Estatísticas

A aplicação final resultante deste primeiro trabalho prático cumpre todos os objetivos propostos pelo docente contando ainda com mais umas funcionalidades implementadas, tais como as que permitem ver uma listagem completa de todos os indivíduos de uma certa entidade e a capacidade responder automaticamente à falta de medicamentos em stock fazendo uma encomenda automaticamente.

A aplicação fornece também um leque de estatísticas de modo a proporcionar uma melhor avaliação sobre esta , ou seja, se estas estiverem de acordo com as nossas acções podemos, numa primeira fase, dizer que a aplicação se encontra coerente na sua extenção, tendo todos os seus objetos a interagirem entre eles.

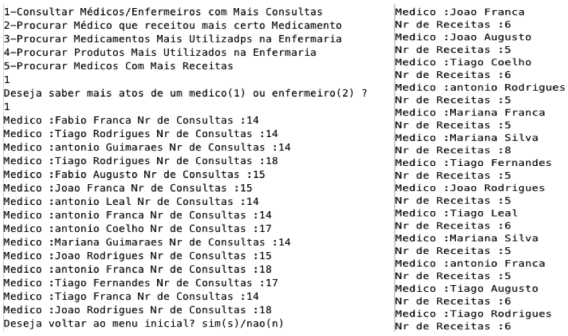


Figura 3: Screenshots de exemplos de dados estatísticos dados pela aplicação.

Na figura encontram-se dois exemplos de dados estatísticos dados pela aplicação . O primeiro é uma amostra dos médicos com mais consultas médicas marcadas.Podemos observar que só aparece os medicos com um numero significativo de consultas comparativamente ao médico(s) com maior número destas. No segundo temos os médicos que mais receitas prescriveram tendo como metodologia de seleção a mesma do primeiro exemplo.

# 3 Conclusões

A criação de um sistema de gestão Hospitalar apresentou-se como um desafio motivavador e de desenvolvimento de capacidades ao nível da linguagem de programação Java. A implementação de uma arquitetura cliente/servidor RMI, utilização de interfaces, HashMap e controlo de concorrência foram os pontos essenciais da dinâmica de trabalho do grupo.

Ao longo do desenvolvimento do trabalho o grupo deparou-se com uma série de dificuldades que serviram para aumentar a autonomia e execução de um programa em Java. O objetivo principal do grupo centrou-se na criação de uma aplicação, seguindo as especificações e requisitos propostos no enunciado prático.

O resultado final foi satisfatório permitindo ao seu utilizador a manipulação de dados de gestão hospitalar, bem como a eficente consulta de informação estatística, necessária à Administração. Posto isto considera-se que os objetivos principais do projeto foram atingidos.

# 4 Bibliografia

[1] Pitt, Esmond, and Kathy McNiff. Java. rmi: The Remote Method Invocation Guide. Addison-Wesley Longman Publishing Co., Inc., 2001.

[2] Downing, Troy Bryan. Java RMI: remote method invocation. IDG Books Worldwide, Inc., 1998.

[3] Savitch, Walter, and Frank M. Carrano. Java: Introduction to Problem Solving and Programming. Prentice Hall Press, 2008.