

Índice

1- Arquitetura do sistema	2
1.1 – Componentes:.....	2
Java <i>Sockets</i>	2
Java RMI	3
1.2 – Interações entre os componentes:	3
Java <i>Sockets</i>	3
Java RMI	3
1.3 – Protocolos de aplicação:.....	4
2 – Arquitetura dos serviços.....	5
2.1 – <i>Server</i> :	5
2.2 – Serviço de Identificação:	5
2.3 – Serviço de <i>Ticketing</i> :.....	8
3 – Diagrama de classes	0
4 – Execução do código	0
.....	2
5 – Questões de Revisão.....	3

1- Arquitetura do sistema

O sistema descrito funciona segundo um modelo *Peer-to-peer* (P2P), uma vez que, é um sistema ponto a ponto (cliente e servidor) onde cada um dos nós que se encontram na rede podem desempenhar a função de cliente ou de servidor sem que exista um servidor mãe que organiza as comunicações todas.

Este modelo permite uma escalabilidade global para que seja possível a troca ou a conexão com múltiplos utilizadores dentro da rede, garante a segurança dos dados e depende da performance da Internet ou dos recursos distribuídos, como o espaço de armazenamento, a latência da rede, etc.

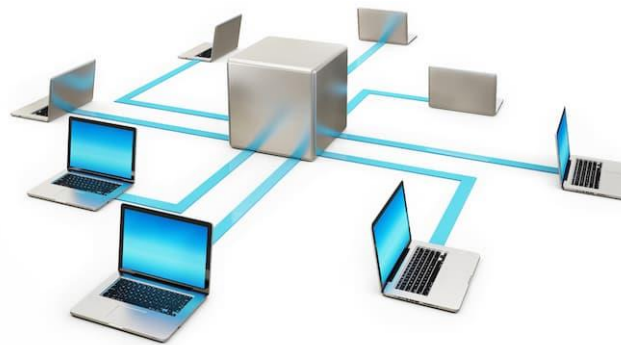


Figura 1 - Modelo P2P

1.1 – Componentes:

Java Sockets

Permite estabelecer um ponto de comunicação entre duas máquinas, por exemplo, um cliente e um servidor.

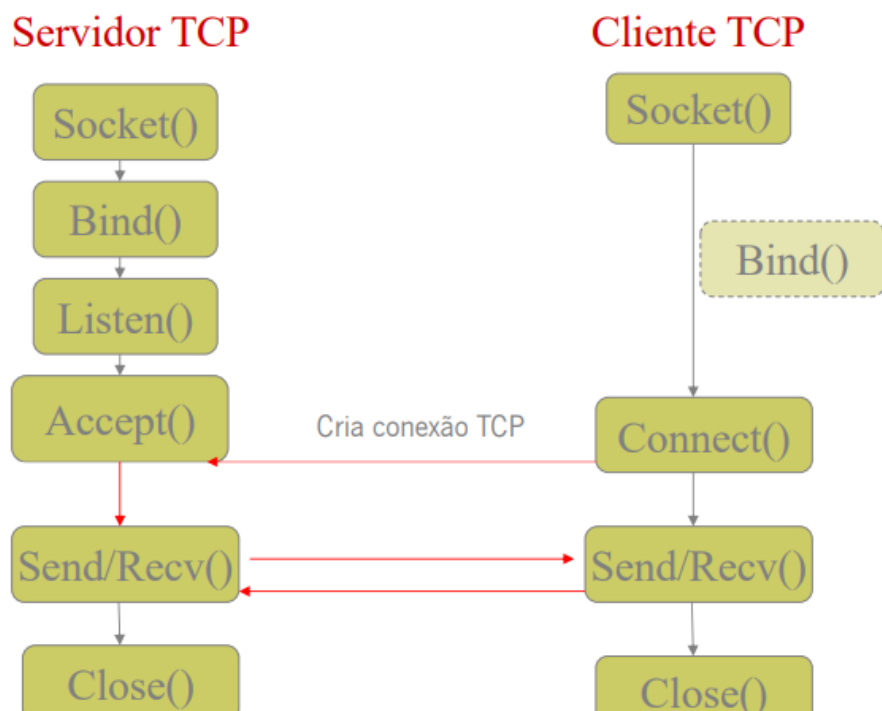


Figura 2 - Comunicação em Java Sockets

Java RMI

Possui módulos que permitem a comunicação entre os intervenientes (cliente e servidor) e, de referências remotas (*skeleton*) onde é feito o tratamento das ligações, por exemplo, para onde deve encaminhar a invocação remota.

De notar, que os métodos invocados pelo RMI são métodos que residem em diferentes máquinas virtuais de Java.

Os *stubs* são classes usadas do lado do cliente que recebem como parâmetros os parâmetros dos métodos utilizados pelo objeto remoto e reencaminha-os para o lado do servidor onde serão interpretados por um *skeleton*.

O *skeleton* recebe os parâmetros do *stub*, executa os métodos do objeto remoto e direciona a informação para os *stubs* dos clientes correspondentes.

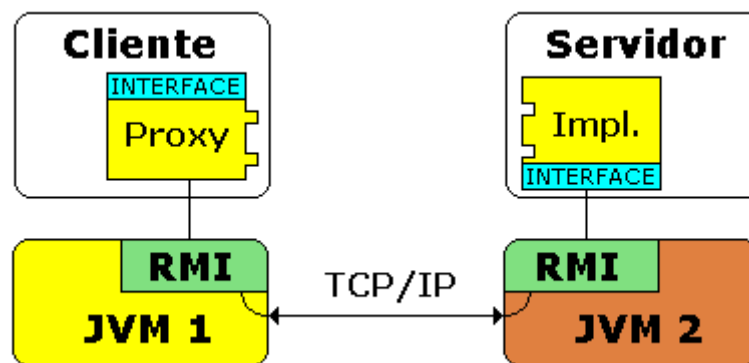


Figura 3 - Comunicação em Java RMI

1.2 – Interações entre os componentes:

Java Sockets

Do lado do servidor, este espera uma conexão vinda do cliente de um dado IP e de uma dada porta. De seguida, o servidor escuta a conexão feita e se encontrar algum pedido aceita-o e retorna um objeto *Socket* e passa a haver uma comunicação entre processos através do protocolo TCP. Assim que é estabelecida uma ligação, o método *accept()* bloqueia todos os outros pedidos até que a conexão acabe (*close()*).

Do lado do cliente, é criado um *Socket* com a *system call socket()* e, de seguida, é estabelecida a conexão a um servidor para que seja possível a troca de mensagens entre processos através do protocolo TCP.

Java RMI

Do lado do servidor, este cria um objeto remoto e regista-o no *rmiregistry()*.

O cliente pede ao *rmiregistry* a referência do mesmo objeto e quando lhe é fornecida a informação é criado um *stub/proxy*. De seguida, o cliente contacta com o *skeleton* por meio do módulo de comunicação

através da invocação de métodos feitos no *stub*. Por fim, o *skeleton* acaba por invocar o mesmo método mas no objeto anteriormente criado.

1.3 – Protocolos de aplicação:

O sistema utilizado implementa os protocolos TCP/IP e HTTP.

O protocolo TCP efetua o serviço de comunicação de bytes entre processos garantindo que os dados são entregues de maneira ordeira e inteiros.

O protocolo HTTP está presente ao nível da web que permite o envio de dados através do protocolo TCP entre o cliente e o servidor.

Desta maneira, o cliente inicia uma ligação TCP e o servidor aceita-a, onde a troca de mensagens entre os dois são transmitidas pelo canal TCP com a ajuda do protocolo HTTP, por fim, finalizadas as trocas, fecha-se a conexão TCP.

2 – Arquitetura dos serviços

2.1 – Server:

Esta Classe implementa o Servidor, onde é criado um *ServerSocket*, cujas aplicações deverão interagir através de *Sockets* de rede, dando uso ao protocolo TCP. Permite assim a conexão do Cliente, possibilitando o de fazer a consulta de um determinado Serviço a partir de uma chave de Acesso que lhe é fornecida após o *Login*.

```
1 package Server;
2
3 public class Server {
4     public static void main(String[] args) throws Exception {
5         Thread t = new Thread(new SI());
6         t.start();
7
8         Thread st = new Thread(new ST());
9         st.start();
10    }
11 }
12
```

Figura 4 - Class Server

2.2 – Serviço de Identificação:

Esta Classe implementa um Serviço de Identificação que permite que os Clientes se identifiquem e obtenham informação sobre o Serviço de *Ticketing*, sendo que esta Classe irá dar uso à função de Autenticação que passa como parâmetros *Username* e NIF, após a autenticação do Cliente o Serviço de Identificação fornece uma Chave de Acesso que o Cliente terá que guardar para poder aceder ao Serviço de *Ticketing*.

A arquitetura deste serviço consiste em inicialmente abrir um *socket* para receber clientes na porta 5000. Quando é recebido um novo cliente, lê o seu pedido e executa um dos serviços: criar, autenticar. O serviço começa por ler o tipo de pedido efetuado, de seguida é o *username* e por fim o NIF. Caso o pedido seja criar um utilizador é instanciada um utilizador do serviço e é adicionado à lista de utilizadores. Caso pedido seja autenticar verifica se existe algum utilizador com o NIF introduzido na lista de utilizadores, e calcula a sua hash baseada no seu NIF.

```

1 package Server;
2
3 import java.io.IOException;
4 import java.net.ServerSocket;
5 import java.net.Socket;
6
7 public class SI implements Runnable {
8
9     private ServerSocket server;
10
11     public SI() {
12         try{
13             this.server = new ServerSocket(5000);
14         } catch (IOException e) {
15             e.printStackTrace();
16         }
17     }
18
19     public void handlerClient() throws Exception{
20
21         String tag;
22         UserList l = new UserList();
23
24         while(true){
25             Socket connect = server.accept();
26
27             new Thread(new SIHandler(connect, l)).start();
28         }
29     }
30
31     @Override
32     public void run() {
33         try {
34             handlerClient();
35         } catch (Exception e) {}
36     }
37 }
38
39

```

Figura 5 – Class SI (do lado do servidor)

```

1 package Server;
2
3 import static Server.HashEnc.getMd5;
4 import java.io.BufferedReader;
5 import java.io.IOException;
6 import java.io.InputStreamReader;
7 import java.io.PrintWriter;
8 import java.net.InetAddress;
9 import java.net.Socket;
10 import java.util.StringTokenizer;
11 import java.util.logging.Level;
12 import java.util.logging.Logger;
13
14 public class SIHandler implements Runnable {
15
16     Socket client;
17     UserList userList;
18
19     BufferedReader input;
20     PrintWriter output;
21
22     public SIHandler(Socket client, UserList userList) {
23         this.client = client;
24         this.userList = userList;
25     }
26
27     public void HandlerNewUser(StringTokenizer tokens) {
28
29         String username = tokens.nextToken();
30         String nif = tokens.nextToken();
31         userList.addUser(new UserS(username, nif));
32         String response;
33         response = "Utilizador criado com sucesso";
34         output.println(response);
35     }
36
37     public void HandlerAuth(StringTokenizer tokens) throws IOException {
38
39         String username = tokens.nextToken();
40         String nif = tokens.nextToken();
41

```

```

42         if(!userList.checkUser(nif)) {
43             output.println("User doesn't exist");
44         }
45         else {
46             InetAddress myIP = InetAddress.getLocalHost();
47
48             String result = getMd5(nif);
49             System.out.println("[SERVER SI CONSOLE] Your hash key: "+result);
50
51             output.println(result);
52         }
53     }
54
55     @Override
56     public void run() {
57         try {
58             //Cria um alocador para Receber a Mensagem
59             this.input = new BufferedReader(new InputStreamReader(client.getInputStream()));
60             //Cria um alocador para Enviar a Mensagem
61             this.output = new PrintWriter(client.getOutputStream(), true);
62
63             //Leitura da mensagem recebida
64             String request = input.readLine();
65             System.out.println("[SERVER SI CONSOLE] " +request);
66
67             while(request != null && !request.equals("quit")) {
68                 StringTokenizer tokens = new StringTokenizer(request,"");
69                 String tag = tokens.nextToken();
70
71                 //Close connection
72                 if(tag.equals("New User")){
73                     System.out.println("[SERVER SI CONSOLE] NEW USER ");
74                     HandlerNewUser(tokens);
75                 }
76
77                 else if(tag.equals("Authentication")){
78                     System.out.println("[SERVER SI CONSOLE] Authentication");
79                     HandlerAuth(tokens);
80                 }
81
82                 else {
83                     output.println("Erro 404 - Not found");
84                 }
85
86                 request = input.readLine();
87             }
88             client.close();
89
90         } catch (IOException ex) {
91             Logger.getLogger(SIHandler.class.getName()).log(Level.SEVERE, null, ex);
92         }
93     }
94 }
95

```

Figura 6 - Class SIHandler (do lado do servidor)

```

1 package Client;
2
3 import java.io.BufferedReader;
4 import java.io.IOException;
5 import java.io.InputStreamReader;
6 import java.io.PrintWriter;
7 import java.net.InetAddress;
8 import java.net.Socket;
9 import java.util.logging.Level;
10 import java.util.logging.Logger;
11
12 public class SIStub {
13
14     private static SIStub instance = null;
15
16     Socket connect;
17     BufferedReader in;
18     PrintWriter out;
19
20     private SIStub() {
21         try {
22             connect = new Socket(InetAddress.getByName("localhost"), 5000);
23             in = new BufferedReader(new InputStreamReader(connect.getInputStream()));
24             out = new PrintWriter(connect.getOutputStream(),true);
25         } catch (IOException ex) {
26             Logger.getLogger(SIStub.class.getName()).log(Level.SEVERE, null, ex);
27         }
28     }
29
30     public static SIStub getInstance() {
31         if(instance == null) {
32             instance = new SIStub();
33         }
34         return instance;
35     }
36
37     public void register(UserC user) throws IOException {
38         System.out.println("Registrar");
39
40         //Criar Mensagem para enviar ao Server
41         String request = "New User," + user.getUsername() + "," + user.getNif();
42

```

```
42 |
43 |         out.println(request);
44 |         String response= in.readLine();
45 |         System.out.println("Response from server: "+response);
46 |     }
47 |
48 |     public String authenticate(UserC user) throws IOException {
49 |         System.out.println("Autenticar");
50 |
51 |         //Criacao da Mensagem para Envio
52 |         String request = "Authentication," + user.getUsername() + "," + user.getNif();
53 |
54 |         out.println(request);
55 |
56 |         String response= in.readLine();
57 |         System.out.println("Response from server: "+response);
58 |
59 |
60 |
61 |         LoggedInUser c = LoggedInUser.getInstance();
62 |         c.setUsername(user.getUsername());
63 |         c.setNif(user.getNif());
64 |         c.setHash(response);
65 |         return response;
66 |     }
67 |
68 | }
69 |
```

Figura 7 - Class SIStub (do lado do cliente)

2.3 – Serviço de *Ticketing*:

Esta Classe implementa um Serviço de *Ticketing* (Bilhetes), que irá posteriormente ao *Login* fornecer uma interface de registo e consulta dos Serviços (Humidade e Temperatura).

A arquitetura deste serviço consiste em abrir um socket para receber clientes na porta 5001 e inicializar a estrutura de dados que consiste num *HashMap* em que a chave é do tipo *string*, que é o identificador único do serviço de rede, e o *value* contém toda a informação do respetivo serviço (descrição, tipo, ip, nome e porta). Quando o serviço ticketing aceita um cliente, analisa o pedido com o *StringTokenizer* em que o primeiro argumento é tipo do pedido, que pode ser consultar ou registar.

Quando regista adiciona um serviço de rede à *HashMap* com toda a informação introduzida no *value* e na *key* introduz o identificador único (ip+port+name). No caso de consulta é verificado se a hash enviada pelo utilizador é válida, e caso seja é devolvido ao utilizador uma lista de tickets do serviço pedido (humidade, temperatura, etc).


```

1 package Server;
2
3 import java.io.IOException;
4 import java.io.BufferedReader;
5 import java.io.PrintWriter;
6 import java.net.ServerSocket;
7 import java.net.Socket;
8 import java.util.HashMap;
9
10 public class ST implements Runnable{
11
12     private ServerSocket server;
13     BufferedReader input;
14     PrintWriter output;
15
16     HashMap<String, ServicoRece> servicosRece;
17
18     public ST() {
19         try {
20             servicosRece = new HashMap<>();
21             this.server = new ServerSocket(5001);
22         } catch (IOException e) {
23             e.printStackTrace();
24         }
25     }
26
27     public void handlerClient() throws Exception{
28         String tag;
29         STState estado = new STState();
30
31         while(true){
32             Socket connect = server.accept();
33
34             new Thread(new STHandler(connect, estado)).start();
35         }
36     }
37
38     public void run() {
39         try{
40             handlerClient();
41         } catch (Exception e) {
42             e.printStackTrace();
43         }
44     }
45 }
46

```

Figura 8 - Class ST (do lado do servidor)

```

1 package Server;
2
3 import static Server.HashEnc.getMd5;
4 import java.io.BufferedReader;
5 import java.io.IOException;
6 import java.io.InputStreamReader;
7 import java.io.PrintWriter;
8 import java.net.Socket;
9 import java.util.StringTokenizer;
10 import java.util.logging.Level;
11 import java.util.logging.Logger;
12
13 public class STHandler implements Runnable {
14
15     Socket client;
16     STState estado;
17     BufferedReader input;
18     PrintWriter output;
19
20     public STHandler(Socket client, STState estado) {
21         this.client = client;
22         this.estado = estado;
23     }
24
25     public void consultHandler(StringTokenizer tokens){
26         String hash = tokens.nextToken();
27         String id = tokens.nextToken();
28         String type = tokens.nextToken();
29
30         if(hash == null || id == null || type == null){
31             output.println("Bad Request");
32             return;
33         }
34
35         if(!getMd5(id).equals(hash)){
36             System.out.println("Invalid Hash!");
37             output.println("Erro! Invalid Hash!");
38             return;
39         }
40
41         output.println(estado.getTicket(type).toString());
42     }
43
44     public void registHandler(StringTokenizer tokens){
45         String description = tokens.nextToken();
46         String type = tokens.nextToken();
47         String ip = tokens.nextToken();
48         Integer port = Integer.parseInt(tokens.nextToken());
49         String name = "";
50
51         if(!type.equals("Socket")) {
52             name = tokens.nextToken();
53         }
54
55         if(description == null || type == null || ip == null || port == null){
56             output.println("Bad Request");
57             return;
58         }
59
60         ServicoRece sr = new ServicoRece(description, type, ip, name, port);
61         String nPort = String.valueOf(port);
62         String concat1 = ip.concat(nPort);
63         String concat2 = concat1.concat(name);
64
65         estado.put(ip + port + name, sr);
66         output.println("Success creating the service!");
67     }
68
69     @Override
70     public void run() {
71
72         try {
73             //Cria um alocador para Receber a Mensagem
74             this.input = new BufferedReader(new InputStreamReader(client.getInputStream()));
75             //Cria um alocador para Enviar a Mensagem
76             this.output = new PrintWriter(client.getOutputStream(), true);
77
78             //Leitura da mensagem recebida
79             String request = input.readLine();
80             System.out.println("[SERVER ST CONSOLE] " + request);
81
82             while(request != null && !request.equals("quit")){
83                 StringTokenizer tokens = new StringTokenizer(request, ",");
84                 String method = tokens.nextToken();
85
86                 if(method.equals("consultar")){
87                     System.out.println("[SERVER ST CONSOLE] Consultar ");
88                     consultHandler(tokens);
89                 }
90
91                 else if(method.equals("registar")){
92                     System.out.println("[SERVER ST CONSOLE] Registar ");
93                     registHandler(tokens);
94                 }
95
96                 request = input.readLine();
97             }
98         } catch (IOException ex) {
99             Logger.getLogger(STHandler.class.getName()).log(Level.SEVERE, null, ex);
100         }
101     }
102 }
103
104
105
106

```

Figura 9 - Class STHandler (do lado do servidor)

```

1 package Server;
2
3 import java.time.Instant;
4 import java.util.ArrayList;
5 import java.util.HashMap;
6 import java.util.List;
7 import java.util.Map;
8
9 public class STState {
10     HashMap<String,ServiceRede> servicosRede;
11
12     public STState() {
13         servicosRede = new HashMap<>();
14     }
15
16     public synchronized List<String> getTicket(String type) {
17         List<String> response = new ArrayList<>();
18         for (Map.Entry<String,ServiceRede> entry : servicosRede.entrySet()) {
19             if (entry.getValue().type.equals(type)) {
20                 Instant timestamp = Instant.now();
21                 String s = entry.getValue().description + "/" + entry.getValue().ip + "/" + entry.getValue().port + "/" + entry.getValue().type + "/" + entry.getKey() + "/" + timestamp + "/" + entry.getValue().name ;
22                 response.add(s);
23             }
24         }
25         return response;
26     }
27
28     public synchronized void put(String key, ServiceRede z) {
29         servicosRede.put(key, z);
30     }
31 }

```

Figura 10 - Class STState (do lado do servidor)

```

1 package Client;
2
3 import java.io.BufferedReader;
4 import java.io.IOException;
5 import java.io.InputStreamReader;
6 import java.io.PrintWriter;
7 import java.net.InetAddress;
8 import java.net.Socket;
9 import java.util.ArrayList;
10 import java.util.List;
11 import java.util.StringTokenizer;
12 import java.util.logging.Level;
13 import java.util.logging.Logger;
14
15 public class STStub {
16     private static STStub instance = null;
17
18     Socket connect;
19     BufferedReader in;
20     PrintWriter out;
21
22     private STStub() {
23         try {
24             connect = new Socket(InetAddress.getByAddress("localhost"), 5001);
25             in = new BufferedReader(new InputStreamReader(connect.getInputStream()));
26             out = new PrintWriter(connect.getOutputStream(), true);
27         } catch (IOException ex) {
28             Logger.getLogger(STStub.class.getName()).log(Level.SEVERE, null, ex);
29         }
30     }
31
32     public static STStub getInstance() {
33         if (instance == null) {
34             instance = new STStub();
35         }
36         return instance;
37     }
38
39     public List<Ticket> consultar(String type) throws IOException {
40         LoggedInUser c = LoggedInUser.getInstance();
41         String consult = "consultar," + c.getHash() + "," + c.getNif() + "," + type;
42
43         out.println(consult);
44         String response = in.readLine();
45         System.out.println(response);
46
47         response = response.replace("[", "").replace("]", "");
48
49         StringTokenizer tickets = new StringTokenizer(response, ",");
50
51         List<Ticket> ticketsList = new ArrayList<>();
52
53         while (tickets.hasMoreTokens()) {
54             String ticket = tickets.nextToken();
55             StringTokenizer tokens = new StringTokenizer(ticket, "/");
56             String description = tokens.nextToken();
57             String ip = tokens.nextToken();
58             String port = tokens.nextToken();
59             String typep = tokens.nextToken();
60             String uniquekey = tokens.nextToken();
61             String timestamp = tokens.nextToken();
62
63             String name = "";
64             if (!typep.equals("Socket")) {
65                 name = tokens.nextToken();
66             }
67
68             Ticket t = new Ticket(description, ip, port, uniquekey, typep, timestamp, name);
69             c.addTicket(t);
70             ticketsList.add(t);
71         }
72         return ticketsList;
73     }
74
75     public String registrar(String request) {
76         try {
77             out.println(request);
78             return in.readLine();
79         } catch (IOException e) {
80             e.printStackTrace();
81         }
82         return "Error";
83     }
84 }

```

Figura 11 - Class STStub (do lado do cliente)

3 – Diagrama de classes

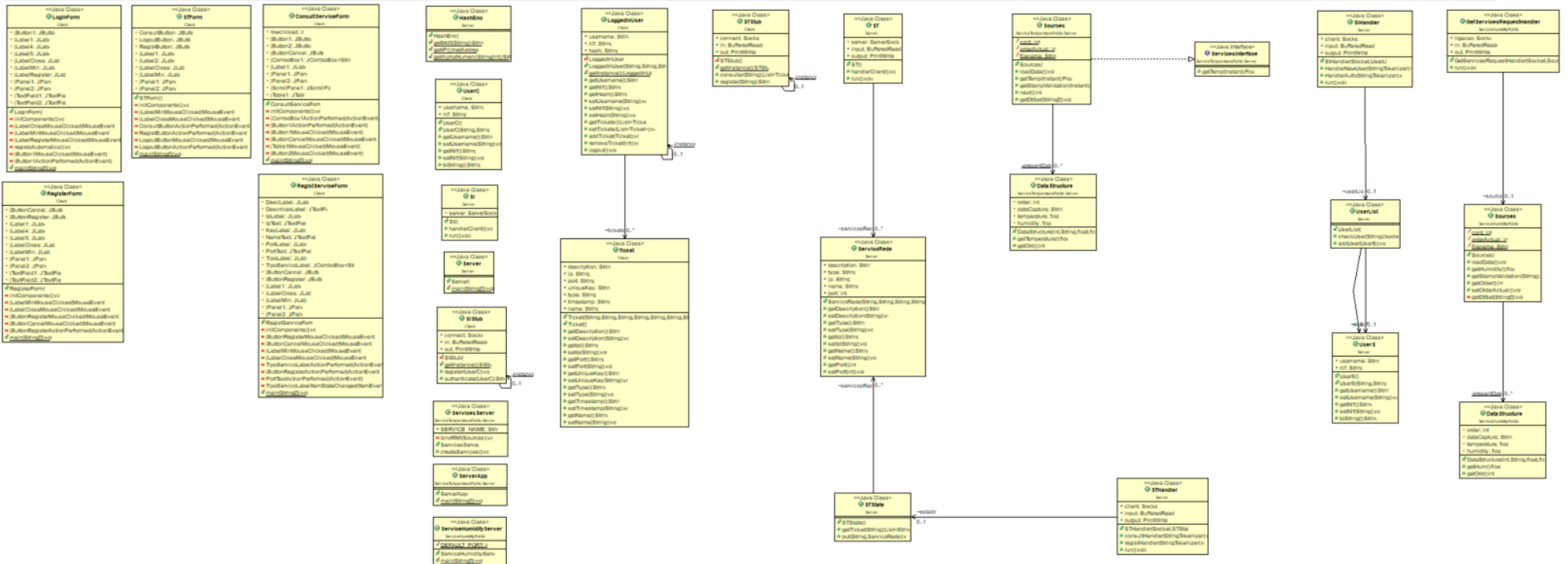
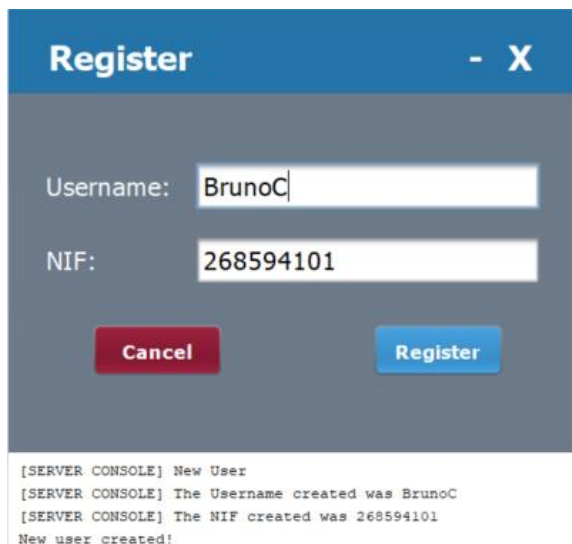


Figura 12 - Diagrama de classes do projeto

4 – Execução do código

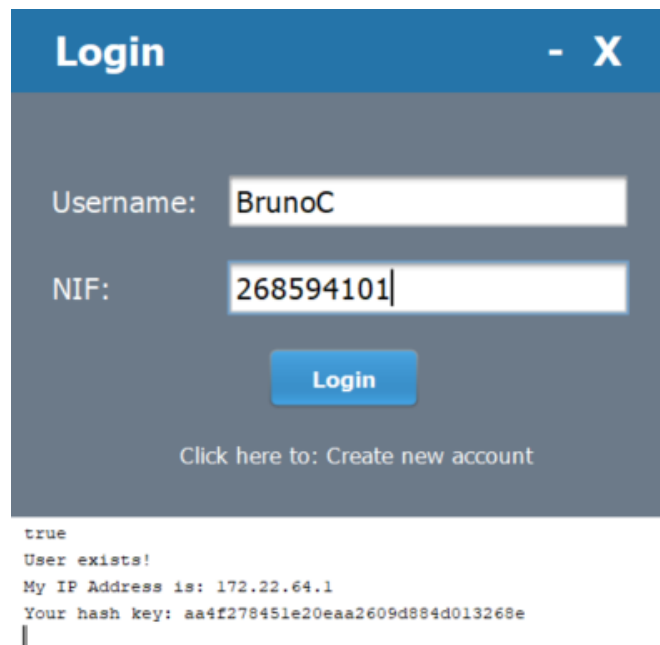
Resposta da parte do servidor quando se dá a criação e o login de um novo utilizador:



The Register interface has a blue header with the title 'Register' and a close button 'X'. It contains two input fields: 'Username:' with the value 'BrunoC' and 'NIF:' with the value '268594101'. Below the inputs are two buttons: a red 'Cancel' button and a blue 'Register' button. At the bottom, a server console log shows the successful creation of the user.

```
[SERVER CONSOLE] New User  
[SERVER CONSOLE] The Username created was BrunoC  
[SERVER CONSOLE] The NIF created was 268594101  
New user created!
```

Figura 13 - Interface de registo

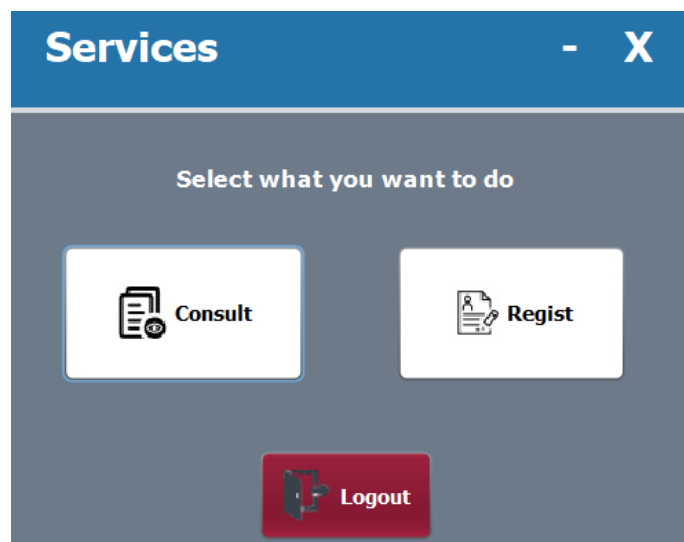


The Login interface has a blue header with the title 'Login' and a close button 'X'. It contains two input fields: 'Username:' with the value 'BrunoC' and 'NIF:' with the value '268594101'. Below the inputs is a blue 'Login' button. At the bottom, there is a link 'Click here to: Create new account'. A server console log shows the successful login with user details.

```
true  
User exists!  
My IP Address is: 172.22.64.1  
Your hash key: aa4f278451e20eaa2609d884d013268e  
|
```

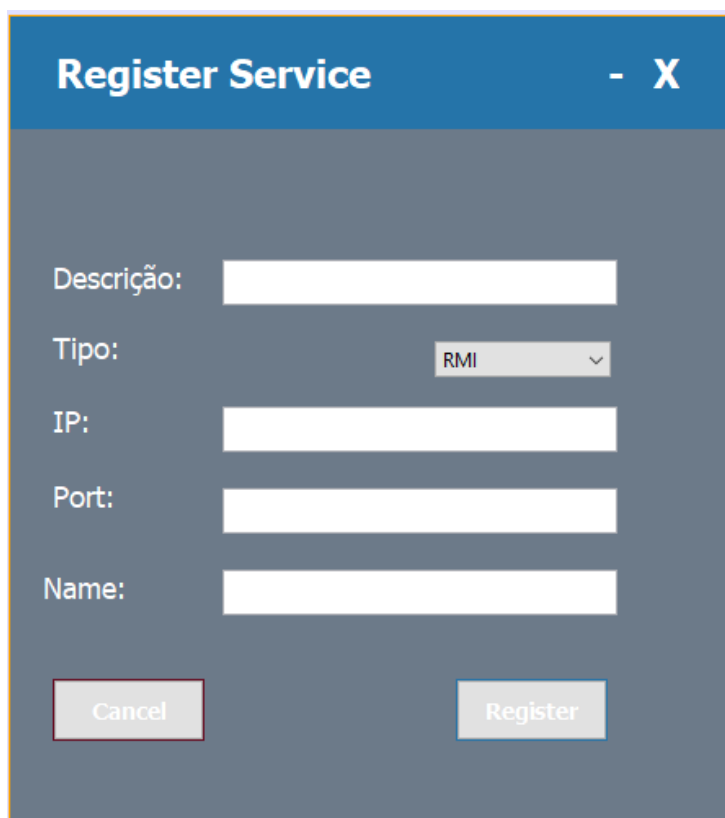
Figura 14 - Interface de autenticação

Depois de ser efetuado o login com um determinado utilizador aparece a seguinte janela que permite ao utilizador conhecer as operações disponíveis (consulta e registo).



The Services interface has a blue header with the title 'Services' and a close button 'X'. It contains the text 'Select what you want to do' and three buttons: a 'Consult' button with a document icon, a 'Regist' button with a document and pencil icon, and a 'Logout' button with a door icon.

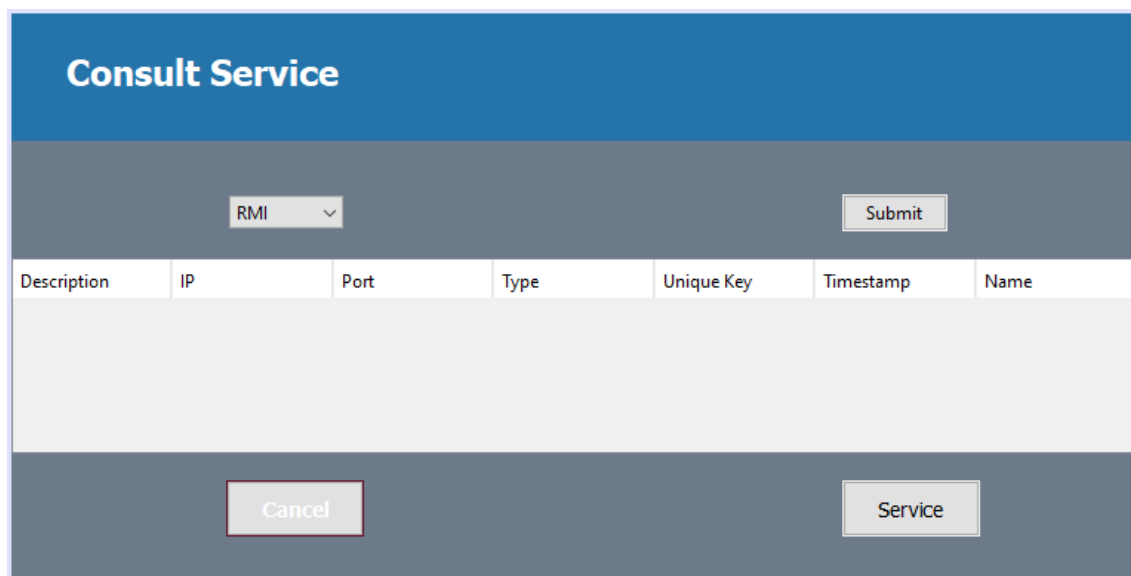
Figura 15 - Interface do ST



The 'Register Service' window has a blue title bar with the text 'Register Service' and a close button 'X'. The main area is dark gray and contains five text input fields labeled 'Descrição:', 'Tipo:', 'IP:', 'Port:', and 'Name:'. The 'Tipo:' field is a dropdown menu currently showing 'RMI'. At the bottom, there are two buttons: 'Cancel' and 'Register'.

Figura 16 - Interface de Registo

A Interface do Registo permite fazer o registo de um Serviço, se este Serviço utilizar Método Remoto de Invocação este terá um TextField por preencher sendo este o Nome, se o Serviço usar comunicação por Sockets este apenas terá de preencher a sua Descrição, IP e Port associado.



The 'Consult Service' window has a blue title bar with the text 'Consult Service'. Below the title bar is a dark gray area containing a dropdown menu for 'Tipo' (currently 'RMI') and a 'Submit' button. Below this is a table with the following columns: Description, IP, Port, Type, Unique Key, Timestamp, and Name. The table body is currently empty. At the bottom, there are two buttons: 'Cancel' and 'Service'.

Figura 17 - Interface de Consulta

Após a seleção do ícone de “Consult” isto remete para um Form onde nos permite aceder a uma lista de Tickets que serão seleccionados para proceder à consulta de um Serviço, tal como, demonstrado em baixo:

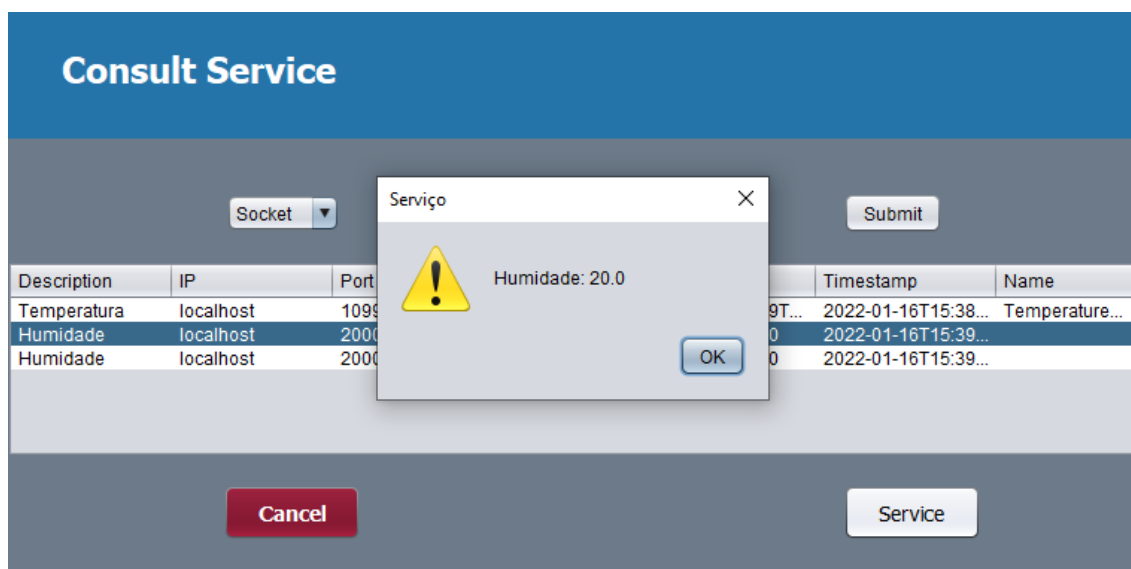


Figura 18 – Consulta de Serviço

Após a seleção do Ticket referente a um Serviço de Humidade este permite visualizar um valor aleatório fornecido através do CSV disponibilizado pela equipa docente.

Este Ticket permitirá fazer consultas deste Serviço durante o tempo estipulado até a invalidação do TimeStamp sendo este 3 minutos.

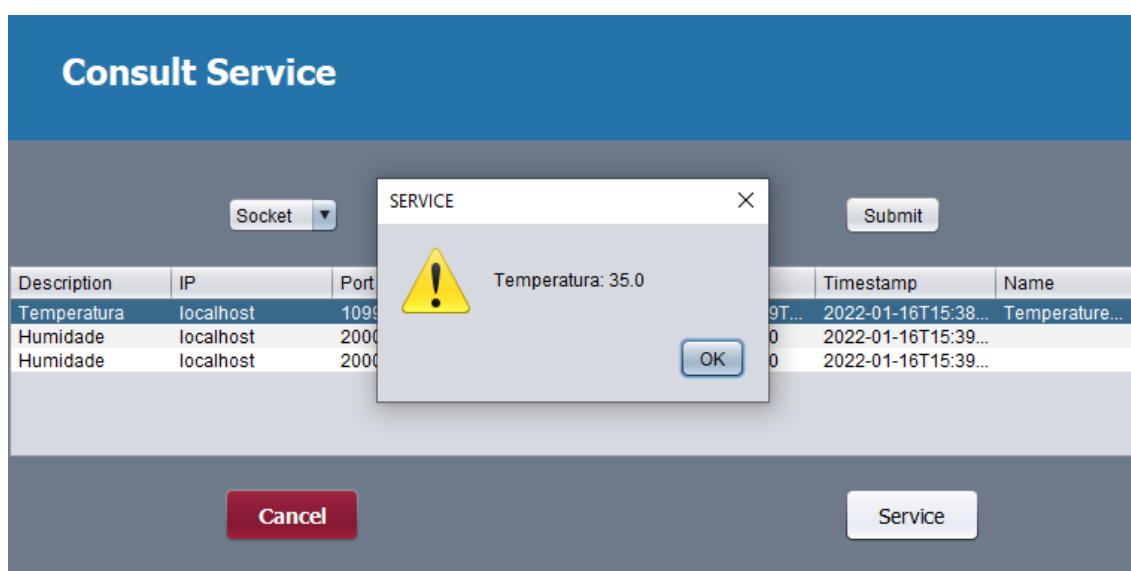


Figura 19 – Consulta de Temperatura

Após a seleção do Ticket referente a um Serviço de Temperatura este permite visualizar um valor aleatório fornecido através do CSV disponibilizado pela equipa docente.

5 – Questões de Revisão

1. O que é necessário garantir para permitir a integração de componentes do sistema desenvolvidos pelos diferentes grupos?

R: Seria necessário possuir um sistema aberto que permita que novos componentes sejam desenvolvidos e acrescentados ao sistema pelas outras entidades (grupos). Um sistema para se tornar num sistema aberto tem que, à partida, dar a conhecer aos desenvolvedores de software as principais interfaces disponíveis. Posto isto, os desenvolvedores de software têm que compreender os inúmeros componentes que foram elaborados por outras entidades e, de maneira a ajudar os mesmos a entender melhor o conceito de certo componente, para isto, usufruem de um método para a documentação, o RFC (possuindo protocolos, discussões, ...). Com isto, o sistema poderá caracterizar-se como um sistema aberto à entrada de novos componentes, assim que, for verificada a compatibilidade de cada componente com o padrão publicado.

2. De que forma é que os componentes desenvolvidos se encontram preparados para lidar com situações de comportamento incorreto ou bloqueio por parte de outros componentes?

R: Os componentes desenvolvidos não se encontram capazes de tolerar falhas parciais do sistema, de momento, se um dos componentes se encontrar inoperável o funcionamento do nosso sistema como um todo irá parar de executar e originar erros de execução.

3. Analise as questões de segurança deste sistema e proponha as alterações necessárias para resolver possíveis faltas.

R: Este sistema possui uma característica de autenticação que permite associar uma hash key (MD5, gerada automaticamente) a um determinado utilizador, garantindo que o sistema confira autenticidade, integridade e confiabilidade. Por outro lado, este sistema não garante que a entidade que está a aceder ao mesmo não seja uma entidade ameaçadora, para isto, podia-se recorrer a um sistema de base dados que passa por criar os utilizadores e armazena toda a informação deles (por exemplo, IP).

4. Discuta outros paradigmas de comunicação que se adequem a este problema.

R: Os sistemas distribuídos possuem 5 principais tipos de comunicação:

- Cliente-servidor (utilizado neste projeto): baseado no conceito em que o servidor oferece serviços e o cliente require-os, onde, para tal, existe uma interface bem definida que é partilhada por todos os clientes, um protocolo que define os pedidos que o servidor deve aceitar e as respostas que deve gerar
- Conjunto de processadores: evita que um processador fique sobrecarregado com a existência de múltiplos utilizadores, garantindo uma otimização do sistema
- Cache de CPU: fornecimento, ao utilizador, de dois níveis de capacidade (baixo e alto), onde o Sistema Operativo é que determina qual o computador que irá executar um dado serviço através de métricas de capacidade e eficiência

- Hierárquico: organiza a arquitetura do sistema de acordo com a capacidade computacional dos processadores, onde os processadores mais afastados da raiz tratam de serviços mais específicos, contrariamente aos processadores mais próximos da raiz
- Orientado ao fluxo de dados: organizado por grafos e cada nó representa um elemento do processador onde as instruções são executadas assincronamente para obter o máximo de desempenho