

# Loudmouth

Uma aplicação chat com arquitectura REST desenvolvida no âmbito da unidade curricular de SDIS@FEUP



**Universidade do Porto**

---

**Faculdade de Engenharia**

**FEUP**

## **Autores:**

Diogo Henrique Marques Cruz	201105483
Hélder Manuel Mouro Antunes	201406163
Pedro Daniel Oliveira Pacheco	201406316
Ricardo Ferreira da Silva	201305163

28/05/2017

# Índice

<b>Loudmouth</b>	<b>1</b>
<b>Índice</b>	<b>2</b>
<b>Introdução</b>	<b>3</b>
<b>Arquitectura</b>	<b>4</b>
Versão 1	4
Cliente	4
Login	4
Canais	5
Chat	6
Servidor	6
Versão 2	7
Cliente	7
Servidor	8
<b>Implementação</b>	<b>9</b>
Versão 1	9
Cliente	9
Servidor	10
Versão 2	12
Cliente	12
Servidor	12
<b>Questões Relevantes</b>	<b>13</b>
Versão 1	13
Segurança	13
Versão 2	14
Segurança	14
<b>Conclusão</b>	<b>15</b>

# Introdução

Loudmouth é uma aplicação full stack de chat construída com a arquitectura REST no âmbito da disciplina de Sistemas Distribuídos na Faculdade de Engenharia da Universidade do Porto. No desenrolar da unidade curricular, tivemos a oportunidade de desenvolver não uma, mas duas versões da mesma aplicação utilizando tecnologias diferentes. Isto foi feito tanto por motivos de valorização extra como de exploração e aprendizagem das API's em questão.

A primeira versão foi desenvolvida em javascript utilizando Electron + React (Node.js) e com recurso a XMLHttpRequest e Socket.io<sup>1</sup> (chat sem polling). O código relativo a esta versão encontra se na pasta electron do projeto.

A segunda versão foi desenvolvida em Java com recurso às APIs com.sun.net.HttpServer e HttpURLConnection, ambas sugeridas no lab6. O código relativo a esta versão encontra se na pasta java/src do projeto.

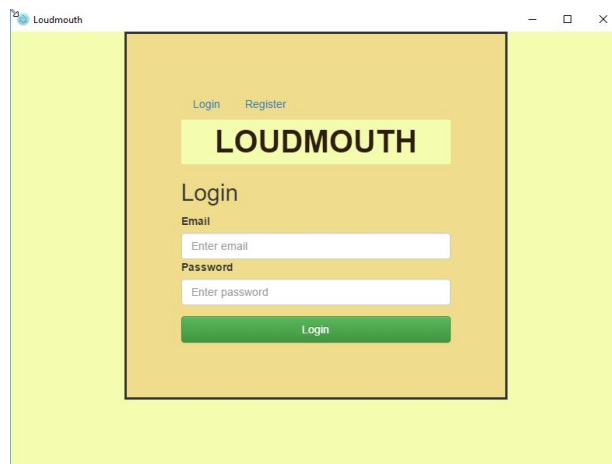
---

<sup>1</sup> "Socket.IO." <https://socket.io/>. Data de acesso: 28 mai. 2017.

# Arquitectura

Neste capítulo, vamos descrever a arquitectura dos clientes e servidores de ambas as versões.

## Versão 1



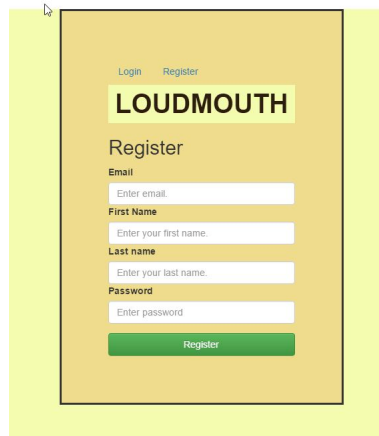
A versão 1 da aplicação é constituída por cliente, um servidor que lida com os pedidos HTTP (REST) e um servidor que recebe e envia mensagens através de sockets, Socket.IO.

## Cliente

O cliente é constituído por 3 components/views principais: Login, Canais e Chat.

### Login

No componente do Login, um utilizador pode fazer login, ou registar-se na aplicação caso ainda não o tenha feito. O email com que se regista terá que ser único na base de dados.

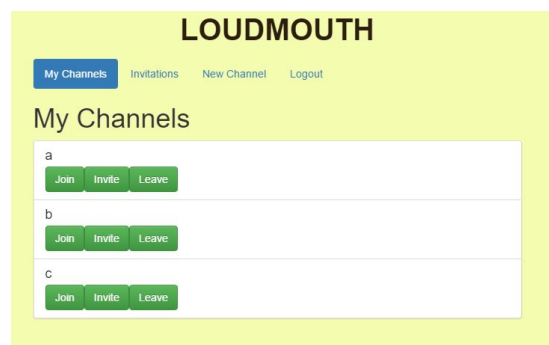


Quase toda a troca de informações entre cliente e servidor é feita com pedidos POST utilizando XMLHttpRequest. É importante notar que a aplicação utiliza token-based authentication para comprovar a fidedignidade de um pedido. O que acontece é que após um utilizador fazer login com sucesso, este recebe um token aleatoriamente gerado pelo servidor. Este token é guardado, pois será sempre enviado no corpo de cada pedido a partir deste momento. Isto serve para garantir que cada pedido provém sempre de um utilizador autenticado e não é um pedido forjado.

## Canais

No componente dos canais, um utilizador pode efectuar várias acções, tudo feito com pedidos POST ao servidor:

- Ver os canais em que está inscrito;
- Entrar nesses canais;
- Desinscrever-se desses canais;
- Criar novos canais;
- Convidar outros utilizadores para esses canais;
- Fazer logout.



## Chat

Por fim, temos o componente do chat. Neste componente, é onde se vê a troca de mensagens entre os utilizadores que nele estão inscritos e presentes (dentro do chat).

Ao contrário do resto da aplicação (que utiliza pedidos POST), a troca de mensagens no chat foi implementada com Socket.IO. Isto permitiu ter uma solução mais sublime, em que em vez de cada cliente ter que estar periodicamente a pedir novas mensagens ao servidor, cada cliente envia a sua mensagem ao servidor e o servidor distribui a mensagem por todos os outros.



## Comunicação entre Componentes:

A arquitectura baseada em REACT permite-nos organizar as nossas views de uma forma hierárquica como se fosse uma árvore. Cada componente pai, pode passar argumentos a um componente filho. Desta forma, à medida que um utilizador explora a aplicação de cima para baixo, a informação recolhida é passada para níveis mais baixos da hierarquia conforme necessário. O oposto é também possível e é feito passando uma função como argumento de um componente para outro, o que efectivamente permite que um componente-filho chame uma função que afecta o estado do componente-pai.

## Servidor

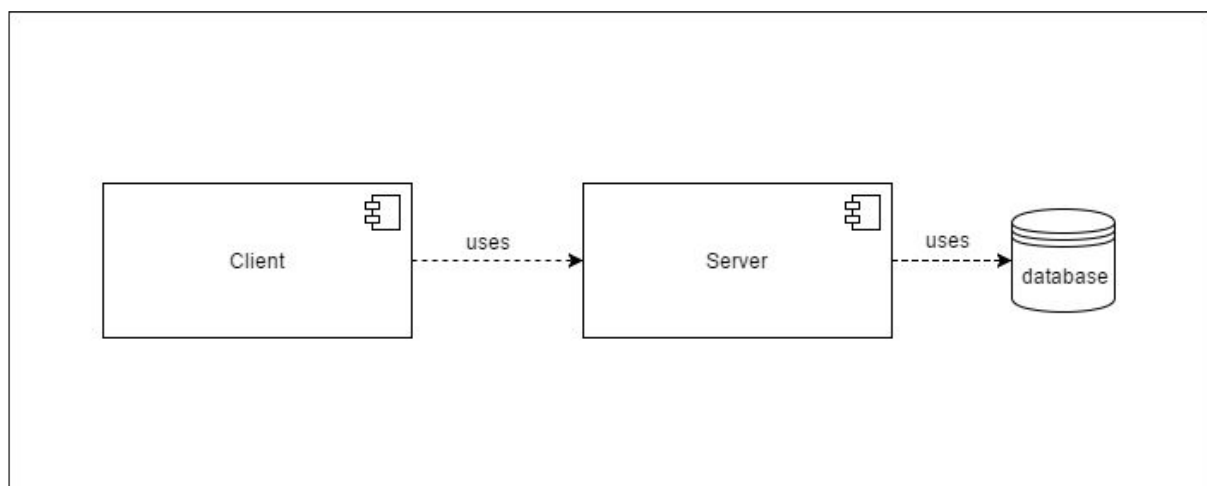
O servidor é constituído por dois ficheiros. O ficheiro `server.js`, é o servidor responsável por processar a comunicação REST na forma de pedidos HTTP e HTTPS. Para efeitos de debugging, as rotas estão disponíveis tanto em HTTP como HTTPS. Neste ficheiro, temos a conexão à base de dados, várias funções auxiliares de processamento, e várias funções associadas a rotas que permitem efectivamente encaminhar e processar os pedidos.

O outro ficheiro, `server-socket.js` é responsável pela comunicação de socket.io. Isto é feito de uma forma semelhante ao outro ficheiro. Cada mensagem

tem um “contexto”, que no fundo é um campo de texto que pode ser interpretado como o “tipo” de mensagem. A cada tipo de mensagem, associamos uma função que é responsável por extrair os argumentos do corpo da mensagem e processar qualquer operação lógica, e por fim efectuar respostas a um, ou mais utilizadores caso seja necessário.

## Versão 2

A aplicação em Java baseia-se numa arquitetura cliente-servidor. O cliente envia pedidos HTTP ao servidor que os trata e, se necessário, envia uma resposta ao cliente. O servidor faz uso da base de dados para responder aos pedidos do cliente.



### Cliente

O cliente possui um package `client.gui` que é responsável pela interface com o utilizador. A parte da comunicação com o servidor está no package `client.network`.

No package `client.network` contém a class `HttpClient` que fornece ao package `client.gui` as seguintes funções:

- `String sendGet(String path, String urlParameters)`
- `String sendPost(String path, String urlParameters)`
- `String sendPostBasicAuthentication(String path, String urlParameters, String username, String password)`
- `String sendGetBasicAuthentication(String path, String urlParameters, String username, String password)`



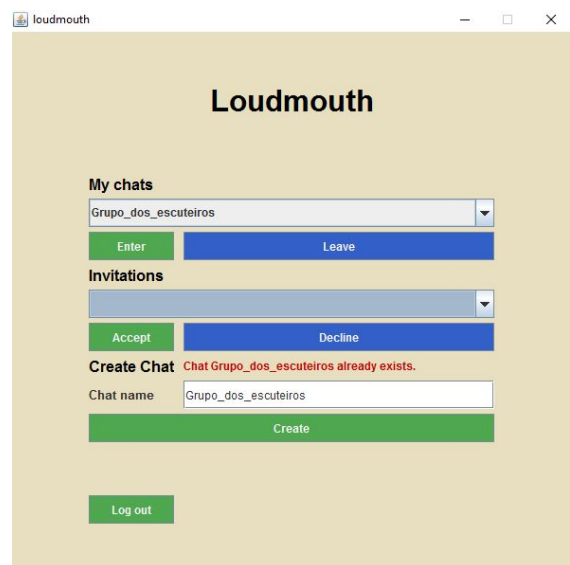
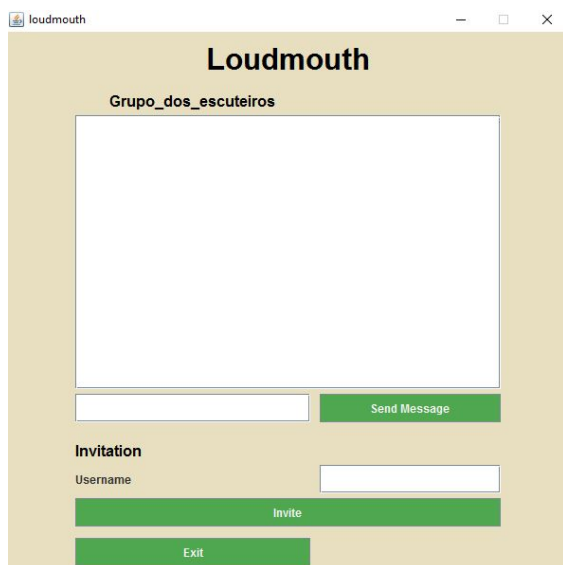
## Servidor

O servidor contém as classes `Server` e `Handler`. A primeira guarda informação na base de dados, cria os contextos do serviço HTTP e associa-os a uma instância da classe `Handler`. A segunda trata os pedidos recebidos pelo cliente.

Contextos criados pelo servidor HTTP:

- `/login`
- `/register`
- `/info`
- `/createChat`
- `/getMyChats`
- `/getInvites`
- `/invitation`
- `/acceptInvite`
- `/declineInvite`
- `/leaveChat`
- `/getMessages`
- `/addMessage`

A arquitetura desta versão é semelhante à anterior. O chat mostra quem enviou as mensagens, a data e hora destas. Depois é possível adicionar pessoas ao chat através do “Username”. O menu contém os chats criados por uma conta e a possibilidade de entrar ou sair de um chat. Tem um campo que contém os convites feitos por outras pessoas para chats. No final tem uma secção para criar novos chats.



# Implementação

## Versão 1

Como foi descrito no capítulo anterior, a versão 1 é constituída por um cliente e dois servidores. Neste capítulo vamos apresentar as tecnologias e APIs utilizadas, sendo que todas são baseadas em Node.js.

*"Node.js® is a JavaScript runtime built on [Chrome's V8 JavaScript engine](#). Node.js uses an event-driven, non-blocking I/O model that makes it lightweight and efficient. Node.js' package ecosystem, [npm](#), is the largest ecosystem of open source libraries in the world."*<sup>2</sup>

### Cliente

O cliente foi escrito em Electron e com recurso ao framework React<sup>3</sup> para Electron.

Electron é uma framework desenvolvida pelo GitHub que permite construir aplicações desktop utilizando ferramentas web. Utiliza javascript como linguagem de programação e oferece um webkit através do browser Chromium para o frontend e utiliza o Node.js para o backend em runtime<sup>4</sup>.

React é uma framework de frontend para javascript que simplifica actions e data management. A framework oferece um DOM virtual para o developer modificar e fazer render que depois o React irá converter no DOM real com o mínimo de operações, otimizando o processo e correndo um menor risco de erros<sup>5,6</sup>.

Através do React, podemos desenhar a interface com uma linguagem de Markup e utilizar javascript para toda lógica, organizando a aplicação em componentes reusáveis.

Relativamente à comunicação, o cliente usa a API default de XMLHttpRequest do javascript e Socket.io. Demonstramos aqui, a sintaxe de um pedido POST em javascript e de uma mensagem em Socket.io.

---

<sup>2</sup> "Node.js." <https://nodejs.org/>. Data de acesso: 28 mai. 2017.

<sup>3</sup> "React - A JavaScript library for building user interfaces - Facebook Code." <https://facebook.github.io/react/>. Data de acesso: 28 mai. 2017.

<sup>4</sup> "Electron | Build cross platform desktop apps with JavaScript, HTML ...." <https://electron.atom.io/>. Data de acesso: 28 mai. 2017.

<sup>5</sup> "React (JavaScript library) - Wikipedia." [https://en.wikipedia.org/wiki/React\\_\(JavaScript\\_library\)](https://en.wikipedia.org/wiki/React_(JavaScript_library)). Data de acesso: 28 mai. 2017.

<sup>6</sup> "What does react.js try to solve? Can you provide a practical example ...." <https://www.quora.com/What-does-react-js-try-to-solve-Can-you-provide-a-practical-example>. Data de acesso: 28 mai. 2017.

```

tryRegister() {
  var loginData =
  {
    "email": this.state.registerEmail,
    "firstname": this.state.registerFirstname,
    "lastname": this.state.registerLastname,
    "password": this.state.registerPassword,
  };
  var request = new XMLHttpRequest();
  request.open('POST', this.props.serverURL + 'register');
  request.setRequestHeader("Content-type", "application/json");
  request.onreadystatechange = () => {
    if (request.readyState !== 4) {
      return;
    }
    if (request.status === 200) {
      localStorage.setItem("username", this.state.registerEmail);
      alert("Register Sucessfull");
      this.setState({onLogin: "true"});
    } else {
      alert("Register error.");
      this.setState({onLogin: "true"});
    }
  };
  request.send(JSON.stringify(loginData));
}

```

```

componentDidMount()
{
  this.socket = io(this.props.serverSocketURL);
  console.log(this.props.serverSocketURL);
  this.socket.on('message', this.handleMessages);
  this.socket.on('connect', function (m) { console.log("socket.io connection open");});
  var token = localStorage.getItem("token");
  this.socket.emit('join', this.props.chatName, token);
}
handleMessages(messages)

```

Socket.io, sendo que utilizámos como uma WebSocket, é um pouco mais alto nível do que puro TCP, no entanto achamos mais interessante e construtivo utilizar uma tecnologia nova adicional para obter uma solução mais elegante do que estar a utilizar mais REST e a fazer polling ao servidor, o que o poderia saturar. No entanto, na versão 2 da aplicação, tudo foi feito com REST a 100%.

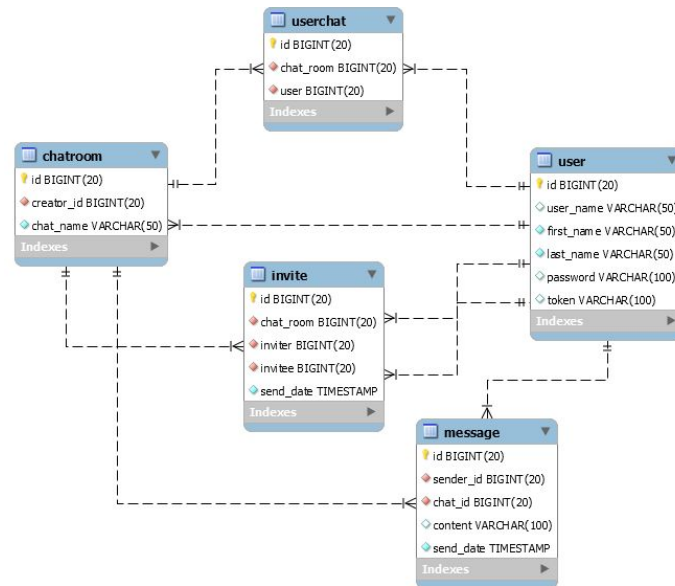
## Servidor

O servidor foi implementado também em javascript, utilizando a framework Express<sup>7</sup> que permite associar funções de processamento a rotas HTTPS de uma forma mais simplificada<sup>8</sup>, e com conexão a uma base de dados MySQL onde são guardados os dados da aplicação.

Apresentamos de seguida o esquema da base de dados e a sintaxe do processamento de um pedido de login.

<sup>7</sup> "Express.js." <https://expressjs.com/>. Data de acesso: 28 mai. 2017.

<sup>8</sup> "Express basic routing - Express.js." <https://expressjs.com/en/starter/basic-routing.html>. Data de acesso: 28 mai. 2017.



```

app.post('/login', function (req, res) {
  var email = req.body.email;
  var password = req.body.password;
  var query = connection.query('SELECT * FROM user WHERE user_name = ?', [email], function (error, results, fields) {
    if (error){
      res.sendStatus(500);
      throw error;
    }
    else {
      if(results.length > 0)
      {
        var tk = randtoken.generate(16);
        var response = {token: tk};
        if(passwordHash.verify(password,results[0].password))
        {
          connection.query('UPDATE user SET token = ? WHERE user_name = ?', [tk,email], function (error, results, fields) {
            if (error){
              res.sendStatus(500);
              throw error;
            }
            else {
              console.log("Login sucessfull.");
              res.send(JSON.stringify(response));
            }
          });
        }
      }
    }
  });
});

```

O servidor utiliza também um certificado SSL que foi gerado e assinado de forma a servir os pedidos utilizando HTTPS, garantindo que os pacotes são encriptados e que não é possível fazer packet sniffing da nossa informação.

Relativamente às mensagens, servidor e cliente usam Socket.io, como foi explicado [anteriormente](#). O cliente envia mensagens que têm um contexto e uma lista de argumentos ao servidor. O servidor canaliza as mensagens para diferentes funções de processamento dependendo do seu contexto e, por fim, efectua operações lógicas e reenvia a mensagem recebida para todos os utilizadores do canal. O estado dos canais, além de ser guardado e monitorizado na nossa base de dados, é também monitorizado internamente dentro do objecto do Socket.io. Isto é o que permite distribuir uma mensagem por todos os utilizadores de um determinado canal.

## Versão 2

A linguagem utilizada tanto para o cliente como para o servidor foi Java.

Foi utilizada a biblioteca externa json-simple para fundamentalmente organizar a informação de modo a que o parser desta fosse mais fácil.

### Cliente

O cliente envia pedidos para o servidor através da API `URLConnection`. Exemplo de pedidos deste tipo podem ser vistos no ficheiro `client/network/HttpClient.java`.

Na parte do chat, utilizou-se a técnica de polling sendo necessário enviados pedidos periodicamente (períodos de 400 ms) para verificar a existência de novas mensagens. Para isso, criou-se um thread responsável apenas por fazer esses pedidos. O thread começa a correr quando um utilizador entra numa sala de chat e interrompe a execução quando o utilizador sai da sala de chat, fazendo-se uso de uma variável volátil do tipo booleana chamada *pollingActive*. Esta parte pode ser vista no ficheiro `client/gui/Chat.java`.

```
161      public class GetMessagesPolling implements Runnable {
162          @Override
163          public void run() {
164              while (pollingActive) {
165                  setMessages(false);
166                  try {
167                      Thread.sleep(400);
168                  } catch (InterruptedException e) {
169                      e.printStackTrace();
170                  }
171              }
172          }
173      }
```

### Servidor

O servidor possui um handler que responde aos pedidos do cliente. Para que vários handlers pudessem correr paralelamente foi aplicado um executor ao servidor HTTP. O tipo de executor escolhido, `CachedThreadPool`, cria uma pool de threads dinâmica, na medida em que cria threads consoante as necessidades. Esta execução é feita no ficheiro `server/Server.java`.

```
60      httpServer.setExecutor(java.util.concurrent.Executors.newCachedThreadPool());
```

# Questões Relevantes

## Versão 1

### Segurança

Relativamente a segurança, a versão 1 da aplicação Loudmouth utiliza HTTPS para garantir que os pedidos são encriptados, e utiliza token based authentication para garantir que os pedidos vem de uma fonte fidedigna. Cada vez que um utilizador faz login, esse utilizador recebe um token. Esse token deve ser utilizado em cada pedido, para confirmar a sua identidade. Caso o token não esteja associado ao utilizador na base de dados (ou seja um token desactualizado), o pedido é rejeitado, pois é impossível verificar que o pedido vem de uma fonte fidedigna.

O facto de utilizarmos HTTPS não só no registo e na autenticação mas em todos os pedidos, previne que além de não ser possível escutar informação de autenticação, não é também possível descobrir o token, o que impede o eventual “roubo” de identidade do utilizador mesmo sem ter as suas credenciais.

```
function tokenChecker(req, res, next) {
  if(req.url == "/login" || req.url == "/register") next();
  else
  {
    var token = req.body.token;
    connection.query('SELECT * from user where token = ?', [token], function(err, rows, fields)
    {
      if (!err)
      {
        if(rows.length > 0)
        {
          //Token is valid.
          next();
        }
      }
      else
      {
        console.log('Error while performing Query.');
      }
    });
  }
}
```

Esta função de pré-processamento é efectuada para todos os pedidos com a excepção do login e do registo.

## Versão 2

### Segurança

Foi utilizado um método de autenticação chamado “basic authentication”. Neste método, o username e password são combinados na string “username:password” que é codificada pelo método Base64. Posteriormente é adicionado ao header uma linha com “Authorization: Basic stringEncoded”.

Para isto, colocou-se uma instância de um BasicAuthenticator em alguns contextos criados pelo server, server/Server.java, linhas 49 a 65.

O cliente apenas tem de adicionar a propriedade “Authorization” ao header do pedido com a string “username:password” codificada (client/network/HttpClient.java).

```
52 public String sendPostBasicAuthentication(String path, String urlParameters, String username, String password)
53     throws Exception {
54     String url = base_url + path;
55     URL obj = new URL(url);
56     HttpURLConnection con = (HttpURLConnection) obj.openConnection();
57     String authorizationEncoded = getAuthorizationStringEncoded(username, password);
58
59     con.addRequestProperty("Authorization", authorizationEncoded);
60     con.setRequestMethod("POST");
61     con.setDoOutput(true);
62     con.setDoInput(true);
63     setPostParameters(con, urlParameters);
64
65     int responseCode = con.getResponseCode();
66     if (responseCode != 200)
67         return "Error: code " + responseCode + ".";
68
69     return readResponse(con);
70 }
```

```
72 public String sendGetBasicAuthentication(String path, String urlParameters, String username, String password)
73     throws Exception {
74     String url = base_url + path + "?" + urlParameters;
75     URL obj = new URL(url);
76     HttpURLConnection con = (HttpURLConnection) obj.openConnection();
77     String authorizationEncoded = getAuthorizationStringEncoded(username, password);
78
79     con.addRequestProperty("Authorization", authorizationEncoded);
80     con.setRequestMethod("GET");
81
82     int responseCode = con.getResponseCode();
83     if (responseCode != 200)
84         return "Error: code " + responseCode + ".";
85
86     return readResponse(con);
87 }
```

# Conclusão

O trabalho foi concluído conforme o plano estipulado na apresentação do projeto.

Poderia ter uma interface mais complexa com várias janelas de canais diferentes em simultâneo.

Poderia ter suporte para envio de fotos ou outros ficheiros.

O desenvolvimento deste projecto, permitiu que o contacto com várias tecnologias e protocolos de comunicação.