

Relatório de Comunicações de Dados

Plataforma de Comunicação

# Licenciatura em Engenharia de Sistemas Informáticos

---

Hélder Martins Carvalho – 15310

João Paulo Figueiredo Carvalho – 15314

Barcelos, maio de 2020



## RESUMO

O objetivo deste relatório é apresentar a informação relevante relativamente ao primeiro trabalho prático realizado para a Unidade Curricular de Comunicações de Dados.

Este trabalho consiste no desenvolvimento de um programa que permita que os alunos do IPCA se autenticuem, com recurso às suas contas académicas, e conversem em tempo real entre si, em grupo ou individualmente.

São feitas descrições sobre como foi interpretado o enunciado, quais as decisões tomadas durante a implementação assim como explicação de algum do código desenvolvido.

No desenvolvimento deste trabalho foi utilizada a linguagem de programação C# e o subsistema gráfico *Windows Presentation Foundation (WPF)*.

Por fim, pretendia-se que este trabalho, além de bem desenvolvido, fosse concluído com sucesso, o que devido à dedicação se veio a concretizar.



## ÍNDICE

INTRODUÇÃO	3
COMO FOI INTERPRETADO O ENUNCIADO	5
DECISÕES TOMADAS DURANTE A IMPLEMENTAÇÃO	7
DESENVOLVIMENTO	8
Funcionamento geral da solução	8
Estrutura do Projeto	9
Estruturas de Dados	10
Comunicação cliente – servidor	10
Estrutura	10
Envio	11
Envio de bytes	11
Preparação e envio de mensagens	12
Envio de ficheiros	13
Receção	14
Receção de bytes	14
Preparação e receção de mensagens	15
Receção de ficheiros	15
Interface visual <i>WPF</i>	17
CONCLUSÃO	19

## LISTA DE FIGURAS

Figura 1 - Estrutura do Projeto	9
Figura 2 - Interface do cliente	17



## INTRODUÇÃO

Este relatório foi concretizado no âmbito do trabalho prático realizado para a Unidade Curricular de Comunicações de Dados da Licenciatura em Engenharia de Sistemas Informáticos, lecionada na Escola Superior de Tecnologia do Instituto Politécnico do Cávado e do Ave.

As diversas tarefas realizadas neste trabalho proporcionaram a oportunidade de, não só aplicar as competências lecionadas/adquiridas nas aulas, mas também o desenvolvimento de novas.

O presente relatório encontra-se dividido em três partes e aborda a forma como o enunciado foi interpretado, quais as decisões tomadas durante a sua implementação e demonstrações de código com explicação.

Repositório GitHub: [https://github.com/joao99c/TP1\\_CD](https://github.com/joao99c/TP1_CD)





## COMO FOI INTERPRETADO O ENUNCIADO

A partir do enunciado e devido á sua clareza, facilmente percebemos quais funcionalidades teríamos de implementar e quais estruturas de dados seriam necessárias para o fazer. Adicionalmente, facilmente decidimos o caminho para o qual queríamos enveredar que era desenvolver um *Front-end* em *WPF*.

Dito isto, graficamente, sabíamos que teríamos de ter uma local onde colocar os utilizadores online, os separadores de chat e o campo de escrita de mensagens.

Quanto ao servidor, este resume-se a uma simples linha de comandos que ficaria à escuta de todas e quaisquer mensagens que lhe seriam enviadas e tratá-las-ia da correta forma de acordo com a sua utilidade.



## **DECISÕES TOMADAS DURANTE A IMPLEMENTAÇÃO**

A partir da interpretação anterior, foi decidido que a primeira tarefa a realizar seria planear as estruturas que iríamos usar assim como idealizar o funcionamento geral do projeto.

Após isso, foram criadas as estruturas de dados necessárias para que fosse possível não só armazenar dados, mas também comunicar entre cliente e servidor.

Depois, foi iniciado o desenvolvimento da interface visual básica necessária para o envio e receção de mensagens, juntamente com o código básico relativo à troca de pacotes entre cliente e servidor.

Por fim, quando estas funções ficaram completamente funcionais, avançamos para o desenvolvimento de todas as outras funcionalidades do projeto.

## DESENVOLVIMENTO

### Funcionamento geral da solução

Resumidamente, o servidor fica à escuta de novas conexões. Sempre que recebe uma, o utilizador é registado/identificado, enviado para todos os outros utilizadores *online* e todos os outros utilizadores *online* são enviados para ele, assim garantimos que todos os utilizadores sabem quem está *online* a qualquer momento.

A partir deste momento, o servidor fica à espera que o cliente lhe envie mensagens.

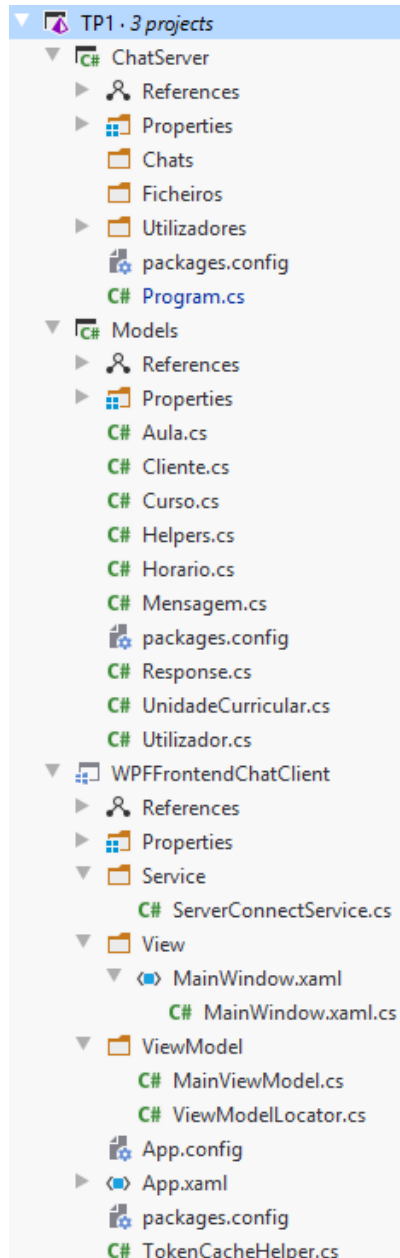
Quando um utilizador abre um separador de chat, este envia uma mensagem ao servidor a pedir o histórico daquele chat. O servidor envia esse histórico que é apresentado no respetivo separador no cliente.

Sempre que um utilizador envia uma mensagem para um chat, o servidor recebe essa mensagem, encaminha-a para o/os destinatário/os e guarda-a no histórico de chat.

Por fim, sempre que um ficheiro é enviado para o servidor, este recebe-o, guarda-o, guarda uma mensagem no histórico de chat com referência a esse mesmo ficheiro e envia essa mensagem para o/os destinatário/os. Desta maneira, os utilizadores podem descarregar o ficheiro clicando na respetiva mensagem. No momento do clique, o cliente envia para o servidor uma mensagem com o nome do ficheiro que quer descarregar e o servidor simplesmente envia esse ficheiro que será guardado na máquina do utilizador.

## Estrutura do Projeto

Para que a solução ficasse modular e mais facilmente navegável, esta foi dividida em vários projetos em que cada um deles tem o seu conjunto de funções específico. Pode-se ler abaixo uma descrição detalhada das pastas e dos ficheiros mais relevantes.



- ChatServer – projeto que realiza a função de servidor. Todo o seu código está no ficheiro `Program.cs`;
- Models – projeto dedicado às estruturas de dados. Todas elas são importantes, mas existe uma que facilita a comunicação entre cliente e servidor que é a `Response.cs`. Quanto ao ficheiro `Helpers.cs`, este não representa nenhuma estrutura de dados, mas sim uma “biblioteca” de funções partilhadas entre cliente e servidor;
- WPFFrontendChatClient – projeto destinado a ser utilizado pelos utilizadores finais;
  - o `ServerConnectService.cs` – contém o código de conexão, envio, receção e tratamento de mensagens;
  - o `MainWindow` – é constituída por dois ficheiros, o “`.xaml`” que tem o código gráfico da janela e o “`.cs`” que tem o código lógico de interligação entre a parte gráfica e a `ViewModel`.
  - o `MainViewModel.cs` – camada intermédia entre a `MainWindow` e o `ServerConnectService`. Pode não só invocar eventos da `MainWindow`, mas também criar comandos que serão executados a partir da interação do utilizador com a interface gráfica;
  - o `ViewModelLocator.cs` – ficheiro onde se registam classes que podem ser injetadas como `ViewModel` em `Views`.

Figura 1 - Estrutura do Projeto

## Estruturas de Dados

Em resumo, um Utilizador é constituído por um id, nome, email, horário, lista de unidades curriculares (se for professor esta lista é das unidades curriculares lecionadas, se for aluno são as unidades curriculares extras), curso e tipo de utilizador.

Um Curso é constituído por nome e lista de unidades curriculares e só é associado a um utilizador do tipo “Aluno”.

Uma Unidade Curricular é constituída por um id e um nome.

As Mensagens são constituídas por id, nome e email de remetente, id e nome de destinatário, conteúdo, data e hora de envio e um indicador de ficheiro (para o cliente o poder transferir).

## Comunicação cliente – servidor

### Estrutura

Para comunicar entre cliente servidor foi desenvolvido um tipo de dados somente para esse efeito, a classe Response.

Esta classe é composta por:

- enumerável que representa o tipo de operação;
- mensagem de chat;
- lista de mensagens que forma o histórico de chat (utilizada quando um utilizador abre um separador de chat);
- utilizador que envia a mensagem;

```
1. [Serializable]
2. public class Response
3. {
4.     public enum Operation
5.     {
6.         Login,
7.         BlockLogin,
8.         EntrarChat,
9.         LeaveChat,
```

```

10.         SendMessage,
11.         SendMessageFile,
12.         PedirFile,
13.         GetUserInfo,
14.         NewUserOnline,
15.         SendFile
16.     }
17.
18.     public Operation Operacao { get; set; }
19.     public Mensagem Mensagem { get; set; }
20.     public List<Mensagem> HistoricoChat { get; set; }
21.     public Utilizador Utilizador { get; set; }
22.
23.     ... Construtor ...
24.
25. }

```

## Envio

A funcionalidade de envio, está dividida em 3 passos, preparação para envio, envio e envio de ficheiros.

### Envio de bytes

A função de envio recebe os bytes a enviar e a sua quantidade e envia-os para o destino.

```

1.  /// <summary>
2.  /// Envia bytes pelo TCP Client
3.  /// </summary>
4.  /// <param name="socket">Socket do TCP Client</param>
5.  /// <param name="buffer">Buffer de bytes a enviar</param>
6.  /// <param name="offset">Posição inicial de envio de bytes</param>
7.  /// <param name="size">Quantidade de bytes a enviar</param>
8.  /// <param name="timeout">Tempo máximo para o envio</param>
9.  /// <exception cref="Exception">Erro</exception>
10. private static void Send(Socket socket, byte[] buffer, int offset, int size,
    int timeout)
11. {
12.     int startTickCount = Environment.TickCount, bytesEnviados = 0;
13.     do
14.     {
15.         if (Environment.TickCount > startTickCount + timeout)
16.         {
17.             throw new Exception("Timeout.");
18.         }
19.         try
20.         {
21.             bytesEnviados += socket.Send(buffer, offset + bytesEnviados, size
- bytesEnviados, SocketFlags.None);
22.         }
23.         catch (SocketException ex)
24.         {

```

```
25.         if (ex.SocketErrorCode == SocketError.WouldBlock ||
26.             ex.SocketErrorCode == SocketError.IOPending || ex.SocketErrorCode ==
27.             SocketError.NoBufferSpaceAvailable)
28.             {
29.                 // Buffer cheio, esperar
30.                 Thread.Sleep(30);
31.             }
32.         else
33.         {
34.             // Erro real
35.             throw;
36.         }
37.     } while (bytesEnviados < size);
```

## Preparação e envio de mensagens

A função de preparação para envio recebe a Response a enviar, transforma-a em bytes e calcula o seu tamanho. Envia esse tamanho para o servidor e depois é que realmente envia os dados.

É necessário enviar o tamanho antes dos dados porque o protocolo TCP, a partir de um certo tamanho, divide os dados em vários pacotes e se o servidor não souber quantos dados vai receber, irá parar a receção com dados a menos ou a mais (de outra mensagem que vem logo de seguida). Desta maneira, garantimos que o servidor recebe apenas a mensagem em questão.

```
1.  /// <summary>
2.  /// Envia mensagem serializada em Json
3.  /// </summary>
4.  /// <param name="tcpClient">Conexão TCP para onde enviar</param>
5.  /// <param name="response">Dados a enviar</param>
6.  public static void SendSerializedMessage(TcpClient tcpClient, Response
7.  response)
8.  {
9.      byte[] enviar =
10.         Encoding.Unicode.GetBytes(JsonConvert.SerializeObject(response));
11.      byte[] tamanhoEnviar = BitConverter.GetBytes(enviar.Length);
12.      try
13.      {
14.          {
15.              Send(tcpClient.Client, tamanhoEnviar, 0, tamanhoEnviar.Length,
16.                  10000);
17.              Send(tcpClient.Client, enviar, 0, enviar.Length, 10000);
18.          }
19.      }
20.      catch (Exception ex)
21.      {
22.          Console.WriteLine("\t" + ex.Message);
23.      }
24.  }
```



## Envio de ficheiros

Para enviar ficheiros, a metodologia é a mesma da função anterior, enviar o tamanho da extensão do ficheiro, enviar a extensão do ficheiro, enviar o tamanho do ficheiro e por fim enviar o ficheiro. Como esta função é utilizada tanto pelo servidor como pelo cliente, existe uma diferença nos dados enviados (para o cliente não é necessário enviar a extensão porque este já a sabe quando inicia a descarga do ficheiro).

```
1.  /// <summary>
2.  /// Envia um ficheiro e a sua informação para o Servidor
3.  /// <para>- Envia a extensão;</para>
4.  /// - Envia o tamanho do ficheiro;
5.  /// <para>- Envia o ficheiro.</para>
6.  /// </summary>
7.  /// <param name="tcpClient">Cliente que envia o ficheiro</param>
8.  /// <param name="extensao">Extensão do ficheiro</param>
9.  /// <param name="caminhoFicheiro">Caminho do ficheiro</param>
10. /// <param name="sendToWpf">
11. ///     Indica para onde é que vai enviar o ficheiro:
12. ///     <para>
13. ///         - true ☐ vai enviar para o WPF (não envia tanta informação
14. ///         sobre o ficheiro porque não é necessário)
15. ///     </para>
16. ///         - false ☐ vai enviar para o servidor (envia toda a informação
17. ///         necessária sobre o ficheiro)
18. ///     </para>
19. public static void SendFile(TcpClient tcpClient, string extensao, string
20. ///     caminhoFicheiro, bool sendToWpf = false)
21. {
22.     byte[] extensaoBytes = null, extensaoSizeBytes = null;
23.     if (!sendToWpf)
24.     {
25.         extensaoBytes = Encoding.Unicode.GetBytes(extensao);
26.         extensaoSizeBytes = BitConverter.GetBytes(extensaoBytes.Length);
27.     }
28.     byte[] ficheiroBytes = File.ReadAllBytes(caminhoFicheiro);
29.     byte[] ficheiroSizeBytes = BitConverter.GetBytes(ficheiroBytes.Length);
30.     NetworkStream networkStream = tcpClient.GetStream();
31.     if (!sendToWpf)
32.     {
33.         networkStream.Write(extensaoSizeBytes, 0, extensaoSizeBytes.Length);
34.         networkStream.Write(extensaoBytes, 0, extensaoBytes.Length);
35.     }
36.     networkStream.Write(ficheiroSizeBytes, 0, ficheiroSizeBytes.Length);
37.     tcpClient.Client.SendFile(caminhoFicheiro);
38. }
```

## Receção

Por consequência do envio, a receção está dividida nas mesmas 3 partes. Preparação para receção, receção e receção de ficheiros.

### Receção de bytes

Esta função recebe uma quantidade de dados definida e guarda-os num array de bytes.

```
1.  /// <summary>
2.  /// Recebe bytes do TCP Client
3.  /// </summary>
4.  /// <param name="socket">Socket do TCP Client</param>
5.  /// <param name="buffer">Buffer onde guardar os bytes</param>
6.  /// <param name="offset">Posição inicial de receção de bytes</param>
7.  /// <param name="size">Quantidade de bytes a receber</param>
8.  /// <param name="timeout">Tempo máximo para a receção</param>
9.  /// <exception cref="Exception">Erro</exception>
10. private static void Receive(Socket socket, byte[] buffer, int offset, int
    size, int timeout)
11. {
12.     int startTickCount = Environment.TickCount, bytesRecebidos = 0;
13.     do
14.     {
15.         if (Environment.TickCount > startTickCount + timeout)
16.         {
17.             throw new Exception("Timeout.");
18.         }
19.         try
20.         {
21.             bytesRecebidos += socket.Receive(buffer, offset + bytesRecebidos,
size - bytesRecebidos, SocketFlags.None);
22.         }
23.         catch (SocketException ex)
24.         {
25.             if (ex.SocketErrorCode == SocketError.WouldBlock ||
ex.SocketErrorCode == SocketError.IOPending || ex.SocketErrorCode ==
SocketError.NoBufferSpaceAvailable)
26.             {
27.                 // Buffer vazio, esperar
28.                 Thread.Sleep(30);
29.             }
30.             else
31.             {
32.                 // Erro real
33.                 throw;
34.             }
35.         }
36.     } while (bytesRecebidos < size);
37. }
```

## Preparação e receção de mensagens

Em resumo, esta função faz o oposto da “preparação e envio”. Recebe a quantidade de bytes da mensagem e depois recebe exatamente essa quantidade. Converte esses bytes num objeto `Response` para poder ser utilizado na execução dos programas.

```

1.  /// <summary>
2.  /// Recebe Mensagem serializada em Json
3.  /// <para>Assim que tiver conteúdo disponível para receber recebe e
    transforma em objeto</para>
4.  /// </summary>
5.  /// <param name="tcpClient">Conexão TCP que vai receber os dados</param>
6.  /// <returns>
7.  ///     "Response" com os dados dentro (objeto des-serializado)
8.  /// </returns>
9.  public static Response ReceiveSerializedMessage(TcpClient tcpClient)
10. {
11.     while (true)
12.     {
13.         if (tcpClient.Available <= 0) continue;
14.         byte[] tamanhoReceber = new byte[4];
15.         try
16.         {
17.             Receive(tcpClient.Client, tamanhoReceber, 0, 4, 10000);
18.             byte[] buffer = new byte[BitConverter.ToInt32(tamanhoReceber,
19. 0)];
20.             Receive(tcpClient.Client, buffer, 0, buffer.Length, 10000);
21.             string jsonString = Encoding.Unicode.GetString(buffer, 0,
22. buffer.Length);
23.             Response response =
24.             JsonConvert.DeserializeObject<Response>(jsonString);
25.             return response;
26.         }
27.         catch (Exception ex)
28.         {
29.             Console.WriteLine("\t" + ex.Message);
30.         }
31.     }
32. }

```

## Receção de ficheiros

Resumidamente, esta função faz o oposto do “envio de ficheiros”. Recebe o tamanho do ficheiro e o ficheiro. Adicionalmente, pode receber a extensão (se for o servidor a receber o ficheiro). Caso seja o cliente a receber o ficheiro, será aberta uma janela de diálogo a pedir para escolher o local onde guardar. Se for o servidor a receber o ficheiro, este é guardado numa pasta predefinida.

```

1.  /// <summary>
2.  /// Recebe um ficheiro
3.  /// <para>- Recebe a extensão;</para>

```

```

4.  /// - Recebe o tamanho do ficheiro;
5.  /// <para>- Recebe o ficheiro;</para>
6.  /// - Guarda o ficheiro.
7.  /// </summary>
8.  /// <param name="tcpClient">Cliente que recebe o ficheiro</param>
9.  /// <param name="mensagem">Mensagem que irá aparecer no chat com o nome do
    ficheiro</param>
10. /// <param name="mensagemModificada">Parâmetro de saída de Mensagem com o
    nome do ficheiro no seu conteúdo</param>
11. /// <param name="nomeFicheiroWpf">Nome do ficheiro que vai ser recebido no
    WPF</param>
12. /// <param name="receiveInWpf">
13. ///     Indica quem é que vai receber o ficheiro:
14. ///     <para>
15. ///         - true ☐ é o WPF que recebe o ficheiro (recebe apenas o tamanho
            do ficheiro e abre a janela para o guardar)
16. ///     </para>
17. ///     <para>
18. ///         - false ☐ é o servidor que recebe o ficheiro (recebe toda a
            informação necessária sobre o ficheiro)
19. ///     </para>
20. /// </param>
21. public static void ReceiveFile(TcpClient tcpClient, Mensagem mensagem, out
    Mensagem mensagemModificada, string nomeFicheiroWpf = null, bool receiveInWpf
    = false)
22. {
23.     byte[] extensaoBytes = null;
24.     NetworkStream networkStream = tcpClient.GetStream();
25.     if (!receiveInWpf)
26.     {
27.         byte[] extensaoSizeBytes = new byte[4];
28.         networkStream.Read(extensaoSizeBytes, 0, extensaoSizeBytes.Length);
29.         extensaoBytes = new byte[BitConverter.ToInt32(extensaoSizeBytes, 0)];
30.         networkStream.Read(extensaoBytes, 0, extensaoBytes.Length);
31.     }
32.     byte[] ficheiroSizeBytes = new byte[4];
33.     networkStream.Read(ficheiroSizeBytes, 0, ficheiroSizeBytes.Length);
34.     int ficheiroSizeInt = BitConverter.ToInt32(ficheiroSizeBytes, 0);
35.     string nomeFicheiro = null;
36.     if (!receiveInWpf)
37.     {
38.         nomeFicheiro = FilesFolder + "\\\" +
            Path.GetFileNameWithoutExtension(Path.GetRandomFileName()) +
            Encoding.Unicode.GetString(extensaoBytes, 0, extensaoBytes.Length);
39.     }
40.     using (MemoryStream memoryStream = new MemoryStream())
41.     {
42.         byte[] buffer = new byte[ficheiroSizeInt];
43.         Receive(tcpClient.Client, buffer, 0, buffer.Length, 10000);
44.         memoryStream.Write(buffer, 0, buffer.Length);
45.         if (!receiveInWpf)
46.         {
47.             File.WriteAllBytes(nomeFicheiro, memoryStream.ToArray());
48.         }
49.         else
50.         {
51.             SaveFileDialog saveFileDialog = new SaveFileDialog {FileName =
                nomeFicheiroWpf, Filter = "All files (*.*)|*.*"};
52.             if (saveFileDialog.ShowDialog() == true)
53.             {
54.                 File.WriteAllBytes(saveFileDialog.FileName,
                    memoryStream.ToArray());
55.             }
56.         }
57.     }
58.     mensagemModificada = null;
59.     if (receiveInWpf) return;

```

```
60.     mensagem.Conteudo = Path.GetFileName(nomeFicheiro);  
61.     mensagemModificada = mensagem;  
62. }
```

## Interface visual WPF

Este foi um dos principais obstáculos no início do desenvolvimento deste projeto visto que nunca tínhamos trabalhado com *Extensible Application Markup Language (XAML)*.

*XAML*, baseado em *XML*, é a linguagem utilizada pelo *WPF* para criar interfaces.

A maior parte da interface foi criada diretamente no ficheiro “.xaml” sendo que os separadores de chat, o chat e as listas de aulas e utilizadores online são criados/as dinamicamente por código no decorrer da execução do programa.

Quando clicamos numa aula ou utilizador, um separador é criado com o histórico de mensagens existentes. Esse separador pode ser fechado clicando no “X” vermelho.

Na parte de baixo da janela existe um campo de texto para escrever as mensagens e um botão para as enviar. Adicionalmente, existe um botão para enviar ficheiros para o chat. Estes botões, quando clicados, detetam qual separador está selecionado e executam funções para que as mensagens e ficheiros sejam enviados para o chat desse separador.



Figura 2 - Interface do cliente



## CONCLUSÃO

Durante este trabalho prático, como já falado anteriormente, foi-nos proposto desenvolver uma plataforma de comunicação. Neste desenvolvimento tivemos a oportunidade de aplicar os conteúdos e matérias lecionadas durante as aulas. Com base nas mesmas, na pesquisa em grupo e no conhecimento já adquirido foi possível resolver o enunciado proposto.

Praticamente todo o código desenvolvido foi um desafio para nós, ou até uma aventura, visto que nunca tínhamos desenvolvido nenhum programa com recurso à tecnologia *WPF*. No que toca à troca de mensagens entre cliente e servidor, aproveitamos a lógica lecionada nas aulas assim como algum do código. Esse código sofreu algumas alterações necessárias para atingirmos as nossas metas de desenvolvimento. Adicionalmente, e voltando ao assunto *WPF*, a partir das bases lecionadas em aula (*bindings* e conexões), a principal dificuldade foi criar elementos gráficos dinamicamente a partir de código, assim como atribuir ações (*commands*) a esses elementos. Foi também desafiante trabalhar com eventos, que podem ser invocados e capturados em diferentes partes do programa e que nos permitem transportar dados através de diferentes “camadas de execução”, ex.: um evento é invocado no serviço de conexão sempre que uma mensagem é recebida, esse evento é capturado na *ViewModel* que por sua vez invoca outro evento que será por fim capturado pela *MainWindow* que apresentará a mensagem recebida.

Estamos perfeitamente mentalizados que algumas das técnicas praticadas nesta solução não serão as mais eficientes, mas para uma primeira vez a desenvolver nesta tecnologia até correu bem.

Para terminar, este trabalho prático foi uma peça essencial não só para o nosso desenvolvimento técnico e profissional, mas também para o nosso sucesso na Unidade Curricular.