Trabalho Prático Nº1

Engenharia Informática

Sistemas Distribuídos

17/04/2023

Por:

António Teixeira, 70835

Índice

Protocolo

Neste trabalho definiu-se que o servidor aceitaria todos os pedidos de conexão, ou seja, todos os possíveis clientes.

Para tal o servidor abre a conexão para o cliente e esta é estabelecida.

De seguida o servidor envia mensagem ao cliente confirmando-a.

O servidor envia instruções ao cliente para concretizar um pedido e solicita-lhe o envio de *IP* como forma de identificação.

Após a receção do *IP*, o cliente insere o que pretende de acordo com as instruções anteriormente fornecidas pelo servidor.

É permitido o envio de um ficheiro (*.*csv*) com as colunas “rua, código postal, número da porta, cidade, Município”.

No final para terminar conexão o cliente segue igualmente as instruções.

Implementação

O atendimento aos clientes é feito utilizando uma “*thread”* para cada um.

A comunicação é feita através de “*sockets”,* utilizando a classe “*NetworkStream”*.

```
0 references
static void Main(string[] args)
{
    TcpListener ServerSocket = new TcpListener(IPAddress.Any, 8888);

    ServerSocket.Start();

    Console.WriteLine("Waiting for a client...");

    //
    CheckFiles.StartWatcher();
    //

    while (true)
    {
        TcpClient Client = ServerSocket.AcceptTcpClient();

        //
        MClient mClient = new MClient(Client);

        var ChildSocketThread = new Thread(() => handle_client(mClient));

        ChildSocketThread.Start();

        Clients.Add(mClient);
        //

        Console.WriteLine("Client Connected {0}", mClient.Id);
    }
}
```

```
    1 reference
    private static void handle_client(MClient Client)
    {
        if (Client.Client.Connected)
        {
            NetworkStream Stream = Client.Client.GetStream();


            HandleServer.Welcome(Client, Stream);
            HandleServer.ClientsMessage(Client, Stream);
        }
    }
}
```

Cada cliente tem direito a ter um ficheiro em processamento, modelo assegurado através do uso de "*mutexes*".

```
1 reference
private static void UseResourceClientsMessage(MClient Client, NetworkStream Stream)
{
    while (Client.Client.Connected)
    {
        string Message = ReceiveMessage(Client, Stream);

        Client.Status = "Sending Data";

        mutex.WaitOne();

        if (Message.Equals("Send File"))
        {
            Client.Status = "Sending File";

            ReceiveFile(Client, Stream);
        }
        else
        {
            //Console.WriteLine("Client " + Client.Id + ": " + Message);
        }

        mutex.ReleaseMutex();

        Client.Status = "Data Sent";
    }
}
```

O acesso aos dados de cobertura separados por Município é condicionado, sendo apenas possível um acesso de cada vez.

```
1 reference
private static void ChangesCities(FileSystemWatcher Watcher)
{
    Watcher.Changed += (s, e) =>
    {
        mutex.WaitOne();

        Thread.Sleep(1000);

        CheckCities(e.FullPath);

        mutex.ReleaseMutex();
    };
}
```

Anexo

- *Client*

Todas as funções presentes recebem como parâmetro o cliente conectado.

1. O cliente conecta-se ao servidor caso exista conexão disponível, após a qual se inicia a troca de informação;

```
0 references
class Client
{
    0 references
    static void Main(string[] args)
    {
        TcpClient Client = new TcpClient();

        //Client.Connect("localhost", 8888);

        Connect(Client);

        if (Client.Connected)
        {
            Connected(Client);

            Chat(Client);
        }
    }
```

Figura 1: Procedimento inicial;

A conexão é possível de forma direta ("*Client.Connect("localhost", 8888)*"), ou pedindo ao utilizador o *IP* e porta aos quais se conectar ("*ConnectClient*").

```
1 reference
private static string AskIP()
{
    Console.Write("IP to connect to: ");

    string IP = Console.ReadLine();
    while (!IPAddress.TryParse(IP, out _))
    {
        Console.WriteLine("Another!");
        IP = Console.ReadLine();
    }

    return IP;
}

1 reference
private static int AskPort()
{
    Console.Write("Port to connect to: ");

    string Port = Console.ReadLine();

    while (!int.TryParse(Port, out _))
    {
        Console.WriteLine("Another!");
        Port = Console.ReadLine();
    }

    return Convert.ToInt32(Port);
}

1 reference
private static void Connect(TcpClient Client)
{
    string IP = AskIP();
    int Port = AskPort();
    Client.Connect(IP, Port);
}
```

Figura 2: Conexão do cliente;

```
1 reference
private static void Connected(TcpClient Client)
{
    IPEndPoint Server = (IPEndPoint)Client.Client.RemoteEndPoint;
    Console.WriteLine("Connected to Server: " + Server.Address + " on port number: " + Server.Port + Environment.NewLine);
}
```

Figura 3: Cliente notificado da conexão;

2. É iniciada a recolha de dados. É possível ao cliente enviar mensagens, enquanto conectado ao servidor;

```
1 reference
private static void Chat(TcpClient Client)
{
    if (Client.Client.Connected)
    {
        ThreadReceiveData(Client);

        Thread.Sleep(1000);

        while (Client.Connected)
        {
            SendMessage(Client);
        }
    }
}
```

Figura 4: Troca de dados iniciada;

3. O cliente escreve mensagem, que lhe é solicitada novamente enquanto esta for nula ou vazia. Logo que a mensagem seja válida é transformada e enviada ao servidor. Se a mensagem for "*Quit*" a conexão cessa do lado do cliente, em caso de "*Send File*" é iniciada a recolha de dados que permite o envio de ficheiros.

```
1 reference
private static void SendMessage(TcpClient Client)
{
    if (Client.Connected)
    {
        //Console.WriteLine("Client: ");
        Console.Write("Client: ");
        string message = Console.ReadLine();

        while (string.IsNullOrEmpty(message))
        {
            //Console.WriteLine("Another! \n");
            Console.WriteLine("Another!");

            //Console.WriteLine("Client: ");
            Console.Write("Client: ");
            message = Console.ReadLine();
        }

        byte[] BufferMessage = Encoding.ASCII.GetBytes(message);

        Client.GetStream().Write(BufferMessage, 0, BufferMessage.Length);

        if (message.Equals("Quit")) { Disconnect(Client); }

        if (message.Equals("Send File")) { SendFile(Client); }
    }
}
```

Figura 5: Cliente envia mensagem;

4. Utilizando a classe “*NetworkStream”* é permitida a recolha de dados através de “*sockets”*. A função tem como resultado final a mensagem recebida.

```
2 references
private static string ReceiveMessage(TcpClient Client)
{
    if (Client.Connected)
    {
        NetworkStream networkStream = Client.GetStream();

        byte[] Buffer = new byte[1024];

        int BytesCount = networkStream.Read(Buffer, 0, Buffer.Length);

        //Console.WriteLine(Encoding.ASCII.GetString(Buffer, 0, BytesCount));

        string DataMessage = Encoding.ASCII.GetString(Buffer, 0, BytesCount);

        return DataMessage;
    }
    else
    {
        return "";
    }
}
```

Figura 6: Cliente recebe mensagem;

5. É criada uma “*thread”* que tem como função receber dados após iniciada.

```
1 reference
private static void ThreadReceiveData(TcpClient Client)
{
    if (Client.Client.Connected)
    {
        var thread = new Thread(() => ReceiveData(Client));

        thread.Start();
    }
}
```

Figura 7: *Thread* de recolha de dados;

6. Durante o período em que o cliente estiver conectado recebe mensagens que têm diferentes resultados: no caso de "*Sending Coverage*" é iniciada a recolha de um ficheiro, "*Sending List of Clients*" o cliente é informado do envio da lista de clientes conectados ao servidor, em qualquer outro caso a mensagem enviada pelo servidor é escrita na consola.

```
1 reference
private static void ReceiveData(TcpClient Client)
{
    while (Client.Connected)
    {
        string Message = ReceiveMessage(Client);

        if (Message.Equals("Sending Coverage"))
        {
            ReceiveFile(Client);
        }
        else if (Message.Equals("Sending List Of Clients"))
        {
            Console.WriteLine(Environment.NewLine + "Receiving List Of Clients...");
            Console.Write("Client: ");
        }
        else
        {
            Console.WriteLine("Server: " + Message);
        }
    }
}
```

Figura 8: Cliente recebe dados;

7. É solicitado ao cliente o nome do ficheiro a enviar, o qual é novamente pedido no caso de este não existir. Assim que o ficheiro seja reconhecido é lida, transformada e enviada toda a informação presente no mesmo.

```
1 reference
private static void SendFile(TcpClient Client)
{
    if (Client.Client.Connected)
    {
        string FilePath = @"C:\Users\PhyMo\Source\repos\P1\File Management\Send\";

        Console.Write("Filename: ");
        string filename = Console.ReadLine();

        while (!File.Exists(FilePath + filename + ".csv"))
        {
            Console.WriteLine("Another!");
            filename = Console.ReadLine();
        }

        string FullFilePath = FilePath + filename + ".csv";

        byte[] Data = File.ReadAllBytes(FullFilePath);

        Client.GetStream().Write(Data, 0, Data.Length);
    }
}
```

Figura 9: Cliente envia ficheiro;

8. Após receber mensagem (os dados do ficheiro) é criado um novo "*ServerCoverage.csv*" onde é escrita toda essa informação. De seguida é informado o cliente da cobertura do servidor.

```
1 reference
private static void ReceiveFile(TcpClient Client)
{
    if (Client.Client.Connected)
    {
        string Destination = @"C:\Users\PhyMo\Source\repos\P1\File Management\Server Coverage\";

        string Data = ReceiveMessage(Client);

        using (FileStream fs = new FileStream(Destination + "ServerCoverage.csv", FileMode.Create, FileAccess.Write))
        {
            fs.Write(Encoding.ASCII.GetBytes(Data), 0, Data.Length);
        }

        Console.WriteLine("Server coverage received from Server");
    }
}
```

Figura 10: Cliente recebe ficheiro;

9. "*Disconnect*" tem como objetivo terminar a conexão do cliente ao servidor, é executada por este e logo de seguida é informado do términus da conexão.

```
    1 reference
    private static void Disconnect(TcpClient Client)
    {
        //Client.Client.Shutdown(SocketShutdown.Both);

        Client.Client.Shutdown(SocketShutdown.Send);

        Client.Close();

        Console.WriteLine("Client Disconnected");
    }
}
```

Figura 11: Cliente é desconectado;

- *Address*

A classe “*Address*” é responsável por guardar os dados de todos os endereços pertencentes à cobertura do servidor.

```
4 references
public class Address
{
    2 references
    public string Street { get; set; }
    2 references
    public string Zip { get; set; }
    2 references
    public string DoorNumber { get; set; }
    2 references
    public string City { get; set; }
    2 references
    public string Municipality { get; set; }
    2 references
    public string Ownership { get; set; }

    1 reference
    public Address(string street, string zip, string doorNumber, string city, string municipality, string ownership)
    {
        Street = street;
        Zip = zip;
        DoorNumber = doorNumber;
        City = city;
        Municipality = municipality;
        Ownership = ownership;
    }
    0 references
    public override string ToString()
    {
        return $"{Street}, {Zip}, {DoorNumber}, {City}, {Municipality}, {Ownership}";
    }
}
```

Figura 12: Classe *Address*;

- *MFile*

A classe “*MFile*” é responsável por guardar os ficheiros a processar.

```
5 references
public class MFile
{
    private static int count = 1;
    2 references
    public int Id { get; set; }
    8 references
    public string Name { get; set; }
    1 reference
    public string Size { get; set; }
    1 reference
    public DateTime? DateCreated { get; set; }
    3 references
    public DateTime? DateModified { get; set; }
    1 reference
    public DateTime? DateAccessed { get; set; }

    1 reference
    public MFile(string name)
    {
        Id = count++;
        Name = name;
    }
}
```

Figura 13: Classe *MFile*;

- *Check Files*

1. Inicialmente é criado um "*mutex"*, responsável por permitir que apenas um ficheiro seja acedido de cada vez, uma lista dos ficheiros a processar e dos endereços armazenados.

```
1 reference
class CheckFiles
{
    private static Mutex mutex = new Mutex();

    private static List<MFile> Files = new List<MFile>();

    private static List<Address> Addresses = new List<Address>();
```

Figura 14: *Mutex,* listas de ficheiros e endereços criados;

2. É definida a localização dos ficheiros recebidos pelo cliente (*rPath*) e dos ficheiros separados por município (*cPath*). São iniciadas as funções "*LoadFiles*", "*ThreadHandleFiles*", "*ThreadHandleCities*" e "*ThreadShowFiles*".

```
1 reference
public static void StartWatcher()
{
    string rPath = @"C:\Users\PhyMo\source\repos\P1\File Management\Data";

    string cPath = @"C:\Users\PhyMo\source\repos\P1\File Management\Data\Data Base\Cities";

    LoadFiles(rPath);
    ThreadHandleFiles(rPath);
    ThreadHandleCities(cPath);
    ThreadShowFiles();
}
```

Figura 15: *Watcher* iniciado;

3. A função recebe como parâmetro a localização dos ficheiros recebidos, os quais são adicionados à lista de ficheiros a processar.

```
1 reference
private static void LoadFiles(string rPath)
{
    var newFiles = Directory.GetFiles(rPath, "*.csv");

    foreach (var File in newFiles)
    {
        NewFile(File);
    }
}
```

Figura 16: Ficheiros a processar adicionados à respetiva lista;

4. A função recebe como parâmetro a localização do ficheiro a adicionar. É criado um "*MFile*" onde é guardado o nome do ficheiro, o tamanho, a data de criação e modificação, o qual é adicionado à lista de ficheiros a processar.

```
3 references
private static void NewFile(string File)
{
    MFile mFile = new MFile(File);

    mFile.Name = Path.GetFileName(File);
    mFile.Size = new FileInfo(File).Length.ToString();
    mFile.DateCreated = DateTime.Now;
    mFile.DateModified = DateTime.Now;

    Files.Add(mFile);
}
```

Figura 17: Ficheiro a processar adicionado à respetiva lista;

5. São criadas e iniciadas “*threads”* responsáveis por: permitir ao utilizador saber que ficheiros serão processados, processar os ficheiros recebidos e aceder à informação presente nos ficheiros separados por município.

   As duas últimas funções recebem a localização dos ficheiros a que pretendem aceder.

```
1 reference
private static void ThreadShowFiles()
{
    var thread = new Thread(() => ShowFiles());

    thread.Start();
}

1 reference
private static void ThreadHandleFiles(string rPath)
{
    var thread = new Thread(() => HandleFiles(rPath));

    thread.Start();
}

1 reference
private static void ThreadHandleCities(string cPath)
{
    var thread = new Thread(() => HandleCities(cPath));

    thread.Start();
}
```

Figura 18: *Threads* iniciadas;

6. O utilizador tem a possibilidade de escrever “*Files*” para saber que ficheiros serão processados e “*Quit*” para fechar o ambiente.

```
1 reference
private static void ShowFiles()
{
    while (true)
    {
        string? Input = Console.ReadLine();

        if (!string.IsNullOrEmpty(Input))
        {
            if (Input.Equals("Files"))
            {
                Files.ToList().ForEach(f =>
                {
                    Console.WriteLine(f.Id + " " + f.Name);
                });
            }
            if (Input.Equals("Quit"))
            {
                Environment.Exit(0);
            }
        }
    }
}
```

Figura 19: Ficheiros a processar e *Quit*;

7. Ambas as funções recebem a localização dos ficheiros.

Através de “*FileSystemWatcher”* é permitida a monitorização de novos ficheiros (*.*csv*).

Caso a localização exista, são controlados os ficheiros recebidos enquanto esta existir. Em caso contrário, o utilizador é informado.

“HandleFiles” é responsável pelos ficheiros recebidos.

“HandleCities” é responsável pelos ficheiros separados por Município, nomeadamente verificar e calcular sobreposições que possam existir.

```
1 reference
private static void HandleFiles(string rPath)
{
    bool rPathExists = Directory.Exists(rPath);

    var Watcher = new FileSystemWatcher(rPath)
    {
        Filter = "*.csv",
        NotifyFilter = NotifyFilters.FileName | NotifyFilters.LastWrite,
        EnableRaisingEvents = true
    };

    if (rPathExists)
    {
        Changes(Watcher);

        while (rPathExists) ;
    }
    else { Console.WriteLine("Check Path"); }
}
```

Figura 20: Monitorização de ficheiros recebidos;

```
1 reference
private static void HandleCities(string cPath)
{
    bool cPathExists = Directory.Exists(cPath);

    var Overlaid = new FileSystemWatcher(cPath)
    {
        Filter = "*.csv",
        NotifyFilter = NotifyFilters.FileName | NotifyFilters.LastWrite,
        EnableRaisingEvents = true
    };

    if (cPathExists)
    {
        ChangesCities(Overlaid);

        while (cPathExists) ;
    }
    else { Console.WriteLine("Check Path"); }
}
```

Figura 21: Monitorização de ficheiros separados por Município;

8. A função recebe o objeto responsável pela respetiva monitorização de ficheiros. Após o servidor receber um ficheiro:

    - este é adicionado à lista de ficheiros a processar e o utilizador é informado;

    - é inscrito num ficheiro de histórico que o ficheiro se encontra "O*pen*";

    - é iniciada a função “*StoreFile*”;

- depois de um ficheiro ser eliminado/processado é eliminado da lista de ficheiros a processar e é inscrito “C*ompleted*”;

- no caso de o nome do ficheiro ser alterado é atualizado na lista e é inscrito no histórico essa alteração;

- caso algum seja modificado é alterada a data de acesso e modificação.

```
1 reference
private static void Changes(FileSystemWatcher Watcher)
{
    Watcher.Created += (s, e) =>
    {
        Console.WriteLine("Created: " + e.Name);
        NewFile(e.FullPath);

        WriteLogs("Open", @"C:\Users\PhyMo\source\repos\P1\P1\File Management\Data\Data Base\Logs\Logs.csv", e.Name);
        StoreFile(e.FullPath);
    };
    Watcher.Deleted += (s, e) =>
    {
        Console.WriteLine("Deleted: " + e.Name);

        Files.RemoveAll(n => n.Name.Equals(e.Name));

        WriteLogs("Completed", @"C:\Users\PhyMo\source\repos\P1\P1\File Management\Data\Data Base\Logs\Logs.csv", e.Name);
    };
    Watcher.Renamed += (s, e) =>
    {
        Console.WriteLine("Renamed: " + e.OldName + " to " + e.Name);

        foreach (var c in Files.Where(n => n.Name.Equals(e.OldName)))
        {
            c.Name = e.Name;
            c.DateModified = DateTime.Now;
        }

        bool FileExists = Files.Any(n => n.Name.Equals(e.Name));

        if (!FileExists) { NewFile(e.FullPath); }

        WriteLogs("Renamed " + e.Name, @"C:\Users\PhyMo\source\repos\P1\P1\File Management\Data\Data Base\Logs\Logs.csv", e.OldName);
    };
    Watcher.Changed += (s, e) =>
    {
        Console.WriteLine("Changed: " + e.Name);

        foreach (var c in Files.Where(n => n.Name.Equals(e.Name)))
        {
            c.DateModified = DateTime.Now;
            c.DateAccessed = DateTime.Now;
        }

        //WriteLogs("Changed", @"C:\Users\PhyMo\source\repos\P1\P1\File Management\Data\Data Base\Logs\Logs.csv", e.Name);
    };
}
```

Figura 22: Alterações a ficheiros recebidos;

9. A função recebe o objeto responsável pela respetiva monitorização de ficheiros. Através do uso de um "*mutex*" é regulado o acesso aos ficheiros separados por Município de modo que apenas possa ser acedido um de cada vez.

```
1 reference
private static void ChangesCities(FileSystemWatcher Watcher)
{
    Watcher.Changed += (s, e) =>
    {
        mutex.WaitOne();

        Thread.Sleep(1000);

        CheckCities(e.FullPath);

        mutex.ReleaseMutex();
    };
}
```

Figura 23: Alterações a ficheiros separados por Município;

10. A função recebe como parâmetro a localização do ficheiro.

    É guardado o *ID* do cliente.

    É definida a localização final do ficheiro, sendo esta uma pasta identificada com o *ID* do cliente, que é criada automaticamente caso não exista.

    Previamente à transferência do ficheiro para a pasta respetiva, o mesmo é processado através da função "*UpdateFile*".

    É escrito no ficheiro de histórico que o ficheiro se encontra a ser processado.

```
1 reference
private static void StoreFile(string path)
{
    string Id = path.Split('\\').Last().Split('_').First();

    string destinationFolder = @"C:\Users\PhyMo\source\repos\P1\P1\File Management\Data\Data Base\" + Id + @"\";

    if (!Directory.Exists(destinationFolder))
    {
        Directory.CreateDirectory(destinationFolder);
    }

    WriteLogs("In Progress", @"C:\Users\PhyMo\source\repos\P1\P1\File Management\Data\Data Base\Logs\Logs.csv", Path.GetFileName(path));
    UpdateFile(path, Id);

    File.Move(path, destinationFolder + Path.GetFileName(path));
}
```

Figura 24: Armazenamento de ficheiros;

11. A função recebe como parâmetro a localização do ficheiro tal como o *ID* do cliente que o enviou.

    É definida a localização do ficheiro de cobertura.

    O conteúdo do ficheiro é lido linha-a-linha e guardado num “*array”*. É ignorada a primeira linha, correspondendo esta ao cabeçalho do ficheiro.

    Caso o ficheiro de cobertura já exista é atualizada a informação desse cliente.

    Caso contrário é aceite toda a informação constante no ficheiro.

    O *ID* do cliente é inserido no final de todas as novas linhas.

    Os dados são guardados e passados como parâmetro para a função “*WriteInFile*”.

    É executada a função “*ProcessFile*”.

```
1 reference
private static void UpdateFile(string Path, string Id)
{
    string CoveragePath = @"C:\Users\PhyMo\source\repos\P1\P1\File Management\Data\Data Base\Coverage\Coverage.csv";

    string[] newlines = File.ReadAllLines(Path);
    string Header = newlines.First() + ",Ownership";
    newlines = newlines.Skip(1).ToArray();

    if (File.Exists(CoveragePath))
    {
        List<string> matches = new List<string>();

        string[] lines = File.ReadAllLines(CoveragePath);
        lines = lines.Skip(1).ToArray();

        foreach (string line in lines)
        {
            string[] values = line.Split(',');

            string Ownership = values[5];

            if (Ownership.Equals(Id))
            {
                matches.Add(line);
            }
        }

        List<string> CLines = lines.ToList();

        matches.ForEach(m => CLines.Remove(m));

        CLines.ToArray();

        string Message = string.Join("," + Id + Environment.NewLine, newlines) + "," + Id + Environment.NewLine + string.Join(Environment.NewLine, CLines);

        WriteInFile(Message, CoveragePath, Header);
    }
    else
    {
        string Message = string.Join("," + Id + Environment.NewLine, newlines) + "," + Id;

        WriteInFile(Message, CoveragePath, Header);
    }

    ProcessFile(CoveragePath);
}
```

Figura 25: Atualização de cobertura;

12. A função recebe como parâmetro a localização do ficheiro.

    O conteúdo do ficheiro é lido linha-a-linha e guardado num “*array*”.

    É ignorada a primeira linha, correspondendo esta ao cabeçalho do ficheiro.

    São eliminados todos os ficheiros que previamente se encontravam separados por Município.

    Para cada linha são obtidos os valores dos Municípios e é definida uma localização onde irão ser guardados os dados, sendo essa diferente para cada Município.

    Para cada linha são obtidos os valores “rua, código postal, número da porta, cidade, Município” e se o cliente é, ou não, proprietário do endereço enviado. Com tais elementos é criado um endereço que posteriormente é adicionado à lista e escrito no ficheiro do Município correspondente.

```
1 reference
private static void ProcessFile(string path)
{
    string[] lines = File.ReadAllLines(path);
    string Header = lines.First() + Environment.NewLine;
    lines = lines.Skip(1).ToArray();

    Addresses.Clear();

    string[] files = Directory.GetFiles(@"C:\Users\PhyMo\source\repos\P1\P1\File Management\Data\Data Base\Cities\");
    foreach (string file in files)
    {
        File.Delete(file);
    }

    foreach (string line in lines)
    {
        string[] values = line.Split(',');

        string Municipality = values[4];

        string destinationMunicipality = @"C:\Users\PhyMo\source\repos\P1\P1\File Management\Data\Data Base\Cities\" + Municipality + ".csv";

        //WriteInFile(line + "," + Id, destinationMunicipality);

        using (FileStream fs = new FileStream(destinationMunicipality, FileMode.Create, FileAccess.Write))
        {
            fs.Write(Encoding.ASCII.GetBytes(Header), 0, Header.Length);
        };
    }

    foreach (string line in lines)
    {
        string[] values = line.Split(',');

        string Street = values[0];
        string Zip = values[1];
        string DoorNumber = values[2];
        string City = values[3];
        string Municipality = values[4];
        string Ownership = values[5];

        Addresses.Add(new Address(Street, Zip, DoorNumber, City, Municipality, Ownership));

        string destinationMunicipality = @"C:\Users\PhyMo\source\repos\P1\P1\File Management\Data\Data Base\Cities\" + Municipality + ".csv";

        using (StreamWriter sw = File.AppendText(destinationMunicipality))
        {
            sw.WriteLine(line);
        }
    }
}
```

Figura 26: Processamento de ficheiros;

13. A função recebe como parâmetro a localização do ficheiro a ser analisado.

    Depois dos ficheiros serem separados por Município, são analisados com o intuito de encontrar sobreposições de cobertura.

    O conteúdo de cada ficheiro é lido linha-a-linha e guardado num "*array".*

    São definidas: uma lista dos endereços com sobreposição; uma "*string"*, inicialmente vazia, que representa a linha lida anteriormente; o número de endereços com sobreposição.

    Para cada linha é comparado com a anterior, o conteúdo do endereço, excluindo, portanto, o *IP* do cliente que o enviou e, em caso de ambas serem idênticas, é adicionada uma *match* e adicionado o endereço à lista de endereços com sobreposição caso este não tenha sido adicionado previamente.

    No final é calculado o número de endereços e o utilizador é informado.

    Também no caso de terem sido encontrados endereços com sobreposição, o utilizador é informado do número de sobreposições e estas são apresentadas.

```
1 reference
private static void CheckCities(string path)
{
    string[] lines = File.ReadAllLines(path);

    List<string> matches = new List<string>();
    string previous = string.Empty;
    int match = 0;

    foreach (string line in lines)
    {
        string CLine = line.TrimEnd(',').Remove(line.LastIndexOf(',') + 1);

        if (CLine.Equals(previous))
        {
            match++;

            if (!matches.Any(l => l.Equals(line)))
            {
                matches.Add(line);
            }
        }

        previous = CLine;
    }

    int streets = lines.Length;
    string filename = path.Split('\\').Last().Split('.').First();

    Console.WriteLine();
    Console.WriteLine(filename + ": " + streets + " addresses");

    if (match > 0)
    {
        matches.ToArray();

        Console.WriteLine("{0} repeated address(es): ", match);

        foreach (string line in matches)
        {
            Console.WriteLine(line);
        }
    }
}
```

Figura 27: Sobreposições em cobertura;

14. A função recebe como parâmetro a mensagem a escrever, a localização do ficheiro onde escrever e o nome do ficheiro em questão.

```
4 references
private static void WriteLogs(string Message, string Path, string Filename)
{
    if (!File.Exists(Path))
    {
        using (StreamWriter sw = File.AppendText(Path))
        {
            sw.WriteLine("Filename,Status,Date");
            sw.WriteLine(Filename + "," + Message + "," + DateTime.Now.ToString());
        }
    }
    else
    {
        using (StreamWriter sw = File.AppendText(Path))
        {
            sw.WriteLine(Filename + "," + Message + "," + DateTime.Now.ToString());
        }
    }
}
```

Figura 28: Ficheiro de histórico;

15. A função recebe como parâmetro os dados a escrever no ficheiro, a localização do mesmo e o cabeçalho. Os dados são posteriormente escritos.

```
    2 references
    private static void WriteInFile(string Message, string Path, string Header)
    {
        using (StreamWriter sw = File.CreateText(Path))
        {
            sw.WriteLine(Header);
            sw.WriteLine(Message);
        }
    }
}
```

Figura 29: Cobertura do servidor;

- *MClient*

A classe “*MClient”* é responsável por guardar os dados de um cliente: *ID*, Nome, “*Status”* e permitir a conexão do mesmo ao servidor.

```
20 references
public class MClient
{
    private static int count = 1;

    8 references
    public int Id { get; private set; }

    8 references
    public string? Name { get; set; }

    18 references
    public TcpClient Client { get; private set; }

    6 references
    public string Status { get; set; } = "Offline";

    4 references
    public string Ownership { get; set; }

    1 reference
    public MClient(TcpClient client)
    {
        if (client == null) throw new ArgumentNullException("client");

        Client = client;
        Id = count++;
    }
}
```

Figura 30: Classe *MClient*;

- *CMFile*

A classe “*CMFile”* é responsável por guardar os dados dos ficheiros recebidos pelo cliente, tais como o nome e o conteúdo, de modo a não permitir o processamento de ficheiros iguais e o “*MClient*” que o enviou.

```
4 references
public class CMFile
{
    1 reference
    public string? Name { get; set; }
    2 references
    public string? Data { get; set; }
    1 reference
    public MClient? Client { get; set; }

    1 reference
    public CMFile(string Name, string Data, MClient Client)
    {
        this.Name = Name;
        this.Data = Data;
        this.Client = Client;
    }
}
```

Figura 31: Classe *CMFile*;

- *HandleServer*

Todas as funções presentes recebem como parâmetro o cliente conectado.

1. Inicialmente é criado um "*mutex"*, responsável por permitir que apenas um ficheiro de determinado operador seja processado de cada vez. É criada também uma lista dos ficheiros já processados.

```
2 references
public class HandleServer
{
    //
    private static Mutex mutex = new Mutex();

    private static List<CMFile> Files = new List<CMFile>();
```

Figura 32: *Mutex* e lista de ficheiros;

2. A função recebe como parâmetro o "*MClient"* conectado, tal como "*NetworkStream"*, responsável pelo envio e recolha de dados.

   Após estabelecida conexão, o estado do cliente é atualizado para "*Connected*" e são iniciadas as funções "*SendPredefinedMessage*", a qual envia uma série de instruções ao cliente por forma a interagir com o servidor, "*ReceiveIP*", ou "*Connected*".

   "*ReceiveIP*" pede ao utilizador a introdução do *IP*,

   "*Connected*" recebe o *IP* de forma automática.

   O *IP* é guardado identificando o cliente através do nome.

   Caso o *IP* seja "127.0.0.7", o cliente é identificado como "*Owner*".

```
1 reference
public static void Welcome(MClient Client, NetworkStream Stream)
{
    if (Client.Client.Connected)
    {
        //Client.Status = "Connected";

        Connected(Client);
        SendPredefinedMessage(Client, "100 OK");
        SendPredefinedMessage(Client, Environment.NewLine + "Send File - to send file, Receive Coverage - to receive server coverage");
        SendPredefinedMessage(Client, ", Quit - to quit.");
        //SendPredefinedMessage(Client, Environment.NewLine + "Enter your IP");
        //ReceiveIP(Client, Stream);
    }
}
```

Figura 33: Instruções dadas ao cliente;

```
1 reference
private static void Connected(MClient Client)
{
    if (Client.Client.Connected)
    {
        Client.Status = "Connected";

        IPEndPoint ClientIP = (IPEndPoint)Client.Client.Client.LocalEndPoint;
        Console.WriteLine("Client connected: " + ClientIP.Address + " on port number: " + ClientIP.Port);

        Client.Name = ClientIP.Address.ToString();

        if (Client.Name.Equals("127.0.0.7")) { Client.Ownership = "Owner"; } else { Client.Ownership = Client.Name; }
        Console.WriteLine("Client Ownership: " + Client.Ownership);
    }
}
```

Figura 34: Servidor notificado da conexão;

3. A função recebe como parâmetro o "*MClient*" conectado, tal como a mensagem que de seguida lhe será enviada.

```
8 references
private static void SendPredefinedMessage(MClient Client, string Message)
{
    if (Client.Client.Connected)
    {
        foreach (var C in Server.Clients.Where(n => n.Id.Equals(Client.Id)))
        {
            byte[] BufferMessage = Encoding.ASCII.GetBytes(Message);

            Client.Client.GetStream().Write(BufferMessage, 0, BufferMessage.Length);
        }
    }
}
```

Figura 35: Envio de mensagem pré-definida;

4. É recebida a mensagem enviada pelo cliente, que enquanto não constituir um endereço *IPv4* ou *IPv6* lhe é pedida novamente.

   O *IP* é posteriormente associado ao nome do cliente conectado.

```
0 references
private static void ReceiveIP(MClient Client, NetworkStream Stream)
{
    if (Client.Client.Connected)
    {
        Client.Status = "Connected";

        string ClientsIP = ReceiveMessage(Client, Stream);

        while (!IPAddress.TryParse(ClientsIP, out _) || string.IsNullOrEmpty(ClientsIP))
        {
            SendPredefinedMessage(Client, "IPv4 or IPv6 Adress");

            ClientsIP = ReceiveMessage(Client, Stream);
        }

        Client.Name = ClientsIP;
    }
}
```

Figura 36: Recolha e análise do *IP* enviado pelo cliente;

5. A função invoca o recurso "*UseResourseClientsMessage*".

```
1 reference
public static void ClientsMessage(MClient Client, NetworkStream Stream)
{
    UseResourceClientsMessage(Client, Stream);
}
```

Figura 37: Troca de informação iniciada;

6. A função recebe como parâmetro o "*MClient*" conectado, tal como "*NetworkStream*".

   A mensagem é lida da *stream* e guardada numa "*string"*.

   Dependendo do valor da mensagem, é executada uma determinada ação de entre as ações: "*Quit*" que aciona a função "*Disconnect"*, "*ReceiveCoverage*" que aciona a função "*SendFile"* e "*Clients*" que aciona a função "*SendListOfClients".*

   No final o valor da mensagem é devolvido.

```
4 references
private static string ReceiveMessage(MClient Client, NetworkStream Stream)
{
    if (Client.Client.Connected)
    {
        byte[] Buffer = new byte[1024];

        int BytesCount = Stream.Read(Buffer, 0, Buffer.Length);

        string DataMessage = Encoding.ASCII.GetString(Buffer, 0, BytesCount);

        if (DataMessage.Equals("Quit")) { Disconnect(Client); }

        if (DataMessage.Equals("Receive Coverage")) { SendPredefinedMessage(Client, "Sending Coverage"); SendFile(Client); }

        if (DataMessage.Equals("Clients")) { SendPredefinedMessage(Client, "Sending List Of Clients"); SendListOfClients(Client); }

        return DataMessage;
    }

    else
    {
        return "";
    }
}
```

Figura 38: Recolha e análise de mensagem recebida;

7. A função recebe como parâmetro o "*MClient*" conectado, tal como "*NetworkStream*".

   Enquanto o cliente se encontrar conectado as mensagens por este enviadas são recebidas e o seu estado é atualizado em conformidade.

   Através do uso de um "*mutex"*, cada operador tem direito ao processamento de apenas um ficheiro de cada vez, após o qual o "*mutex"* é libertado.

```
1 reference
private static void UseResourceClientsMessage(MClient Client, NetworkStream Stream)
{
    int nFiles = 0;

    while (Client.Client.Connected)
    {
        string Message = ReceiveMessage(Client, Stream);

        Client.Status = "Sending Data";

        mutex.WaitOne();

        if (Message.Equals("Send File"))
        {
            Client.Status = "Sending File";

            nFiles++;

            ReceiveFile(Client, Stream);
        }
        else
        {
            //Console.WriteLine("Client " + Client.Id + ": " + Message);
        }

        mutex.ReleaseMutex();

        Client.Status = Client.Name + " Sent " + nFiles + " Files";
    }
}
```

Figura 39: Uso de m*utex* condiciona o processamento de ficheiros;

8. É definida a localização do ficheiro de cobertura a enviar, após o qual os dados deste são lidos e enviados ao cliente mas sem a última coluna (identificação do proprietário).

```
1 reference
private static void SendFile(MClient Client)
{
    if (Client.Client.Connected)
    {
        string FilePath = @"C:\Users\PhyMo\Source\repos\P1\P1\File Management\Data\Data Base\Coverage\Coverage.csv";

        if (File.Exists(FilePath))
        {
            string[] Data = File.ReadAllLines(FilePath);

            string newData = string.Empty;

            foreach (string line in Data)
            {
                string[] values = line.Split(',');

                values = values.Take(values.Length - 1).ToArray();

                newData += string.Join(",", values) + Environment.NewLine;
            }

            byte[] newFile = Encoding.ASCII.GetBytes(newData);

            Client.Client.GetStream().Write(newFile, 0, newFile.Length);
        }
        else { Console.WriteLine("File Not Found"); }
    }
}
```

Figura 40: Envio de cobertura do servidor;

9. A função recebe como parâmetro o "*MClient*" conectado, tal como "*NetworkStream*".

   É definido um formado para a data a introduzir como parte do nome do ficheiro e definida a localização onde ficará guardado.

   Através da função "*ReceiveMessage"* são recebidos os dados inerentes ao ficheiro. Caso os dados já tenham sido enviados pelo cliente, é-lhe enviada uma mensagem de aviso. Em caso contrário, é criado um novo ficheiro cujo nome é constituído pelo *IP* do cliente, *ID* e data em que o servidor o recebeu.

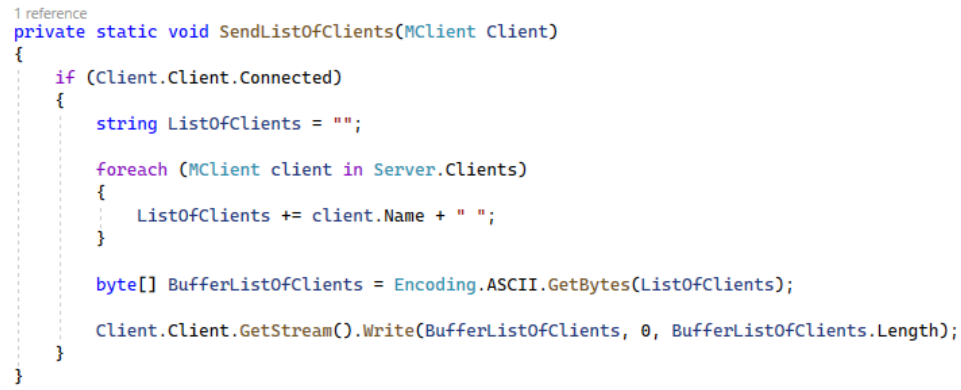
   Os dados do ficheiro são depois adicionados à lista de ficheiros recebidos.

   O servidor recebe aviso de ficheiro recebido.

```
1 reference
private static void ReceiveFile(MClient Client, NetworkStream Stream)
{
    if (Client.Client.Connected)
    {
        string format = "Mddyyyyhhmmsstt";
        string DateAndTime = DateTime.Now.ToString(format);

        string Destination = @"C:\Users\PhyMo\Source\repos\P1\P1\File Management\Data\";

        string Data = ReceiveMessage(Client, Stream);

        //string Municipality = Data.Split(',')[9];

        if (Files.Any(f => f.Data.Equals(Data)))
        {
            SendPredefinedMessage(Client, "File already sent");
        }
        else
        {
            using (FileStream fs = new FileStream(Destination + Client.Ownership + "_" + Client.Id + "_" + DateAndTime + ".csv", FileMode.Create, FileAccess.Write))
            {
                fs.Write(Encoding.ASCII.GetBytes(Data), 0, Data.Length);
            }

            Files.Add(new CMFile(Client.Name + "_" + Client.Id + "_" + DateAndTime, Data, Client));
            // in alternative, one could read every file in the directory and compare the data. The method used is faster. Assumes that the server stays on.

            Console.WriteLine("{0}, Client {1} submitted a file", Path.GetFileName(Client.Name), Client.Id);
        }
    }
}
```

Figura 41: Ficheiro é processado;

10. É criada uma “*string”* inicialmente vazia a qual é preenchida com os nomes (*IP*) dos diversos clientes conectados ao servidor e depois enviada ao cliente.



```
1 reference
private static void SendListOfClients(MClient Client)
{
    if (Client.Client.Connected)
    {
        string ListOfClients = "";

        foreach (MClient client in Server.Clients)
        {
            ListOfClients += client.Name + " ";
        }

        byte[] BufferListOfClients = Encoding.ASCII.GetBytes(ListOfClients);

        Client.Client.GetStream().Write(BufferListOfClients, 0, BufferListOfClients.Length);
    }
}
```

Figura 42: Envio da lista de clientes conectados;

11. Após a utilização da função “*Disconnect*”: o cliente é informado, são terminadas as vias de comunicação, o seu estado é alterado, é removido da lista de clientes conectados, o servidor é notificado.

```
        1 reference
        private static void Disconnect(MClient Client)
        {
            SendPredefinedMessage(Client, "400 BYE");

            Client.Client.Client.Shutdown(SocketShutdown.Both);

            Client.Status = "Disconnected";

            Server.Clients.Remove(Client);

            Client.Client.Close();

            Console.WriteLine("Client {0} Disconnected", Client.Id);
        }
        //
}
```

Figura 43: Conexão terminada;

- *Server*

1. Inicialmente é criada uma lista de "*MClient*" que corresponde aos clientes conectados ao servidor.

```
3 references
class Server
{
    public static List<MClient> Clients = new List<MClient>();
```

Figura 44: Criada lista de clientes conectados;

2. O servidor inicia o seu funcionamento no endereço pré-definido, após o qual é iniciado "*socket"* de comunicação.

   É iniciado o "*StartWatcher"*, anteriormente descrito.

   O servidor aceita qualquer e toda a conexão através do ciclo "*while (true)*".

   Qualquer cliente conectado é adicionado como "*MClient*" à lista de clientes conectados e é criada e iniciada uma "*thread"* para a comunicação entre este e o servidor, após a qual o servidor é informado da conexão.

```
0 references
static void Main(string[] args)
{
    TcpListener ServerSocket = new TcpListener(IPAddress.Any, 8888);

    ServerSocket.Start();

    Console.WriteLine("Waiting for a client...");

    //
    CheckFiles.StartWatcher();
    //

    while (true)
    {
        TcpClient Client = ServerSocket.AcceptTcpClient();

        //
        MClient mClient = new MClient(Client);

        var ChildSocketThread = new Thread(() => handle_client(mClient));

        ChildSocketThread.Start();

        Clients.Add(mClient);
        //

        Console.WriteLine("Client Connected {0}", mClient.Id);
    }
}
```

Figura 45: Comunicação iniciada;

3. São iniciadas as funções "*Welcome*" e "*ClientsMessage*" anteriormente referidas. Ambas recebem como parâmetro a "*stream"* de dados e o cliente conectado.

```
    1 reference
    private static void handle_client(MClient Client)
    {
        if (Client.Client.Connected)
        {
            NetworkStream Stream = Client.Client.GetStream();


            HandleServer.Welcome(Client, Stream);
            HandleServer.ClientsMessage(Client, Stream);
        }
    }
}
```

Figura 46: Cliente atendido;