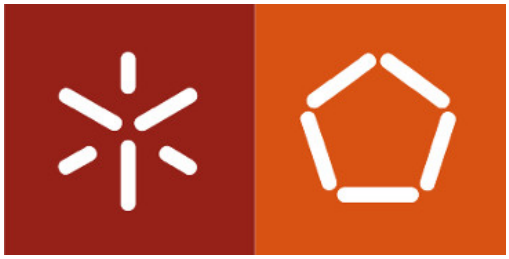

Universidade do Minho

Mestrado integrado em Engenharia Informática

Rede Overlay de Anonimização do Originador

Diogo Ribeiro A84442

José Monteiro A83638

Rui Reis A84930

Comunicações por Computador

2019/2020

Departmento de Informática

19 May 2020

Contents

1 Introdução **1**

2 Arquitetura da Solução **2**

2.1 Início da Ligação 2
2.2 Ligação Peer-to-Peer 3
2.3 Ligação Peer-to-Server 5
2.4 Ligação Server-to-Peer 5
2.5 Ligação Peer-to-Peer 5
2.6 Fim da Ligação 5

3 Especificação do Protocolo UDP **7**

3.1 Formato das mensagens protocolares (PDU) 7
3.2 Interações 8

4 Implementação **9**

4.1 Multiplexação 9
4.2 Encriptação da Comunicação 9
4.3 Controlo de Perdas 10
4.4 Autenticação da Origem 10

5 Testes e Resultados **12**

6 Conclusões e Trabalho Futuro **14**

1 Introdução

À medida que a sociedade humana tem evoluído no sentido de uma *Internet Universal*, a manutenção da privacidade *Online* tem vindo a ser um tópico cada vez mais prevalecente nas nossas vidas.

Tudo o que fazemos na *internet* flui dentro de uma rede complexa, apesar de bem organizada, sendo transmitida e retransmitida várias vezes ao longo de um dado percurso, Facilmente se verifica que se um dado pacote têm a necessidade de passar por vários intermediários, muitos sobre os quais não possuímos soberania alguma, há necessariamente uma questão de segurança implícita nesta matéria.

Sobre esta rede, a internet, são transmitidos todos os tipos de informações, desde mensagens de texto a perguntar sobre a família, até dados empresariais que podem custar a uma empresa, ou governo, milhões de euros. Desta forma, é necessário garantir a segurança de qualquer mensagem transmitida ao longo de uma rede, para que a internet se torne num mecanismo verdadeiramente útil, onde as pessoas possuem algum controlo sobre a privacidade do conteúdo que transmitem, em oposto a uma rede vulnerável e sem confiança.

Por essa razão, pretendemos desenvolver um mecanismo, naturalmente simplificado, para enfrentar esta temática. Atacando assim este problema tão imperioso, com a apresentação de uma solução viável para a garantia de comunicação privada entre duas entidades. Delineando de seguida testes para a garantia de qualidade desta peça de software.

2 Arquitetura da Solução

Antes de realizarmos uma descrição do funcionamento do nosso programa, precisamos de saber como criar os nodos que vão servir para redireccionar as mensagens que percorrem a rede, de forma a manter a privacidade dos utilizadores. Para tal, temos que utilizar o comando

```
AnonGW target-server <TARGET SERVER ADRESS> port <PORT NUMBER> \n
                                        overlay-peers <PEER ADRESS>...
```

partindo do pressuposto que a máquina já possuiu o software necessário.

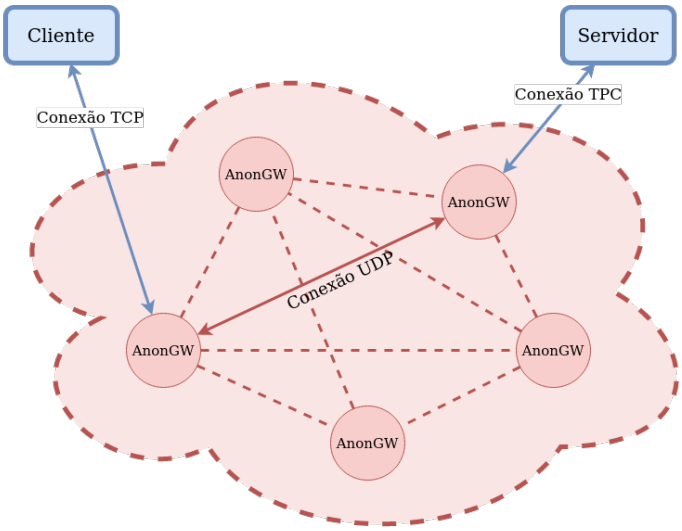

Figure 1: Arquitetura AnonGW

Agora que temos a nossa rede "populada" com *peers* capazes de redirecionar a comunicação do cliente, estamos prontos para a utilizar. Um cliente que queira comunicar com o servidor, necessita somente de utilizar o endereço de um dos peers da nossa rede.

2.1 Início da Ligação

Uma máquina quando é inicializada como `AnonGW`, cria 2 sockets que vamos necessitar ao longo de todo o processo, `internalsocket` e `controlsocket`. Três threads vão estar a funcionar constantemente: `TCPListener`, `TCPSpeaker` e `UDPListener`. Nesta primeira fase, vamos necessitar da primeira thread `TCPListener` que vai utilizar um socket TCP, `external_socket_in`, para comunicar com o cliente, e cuja porta, `outsideport`, definimos como 0.

```
external_socket_in = new ServerSocket(this.outside_port);
```

Figure 2: Iniciar Socket TCP para receber pedido do Cliente

O cliente começa por enviar um pedido a um dos nodos. A thread `TCPListener` recebe esse pedido em formato `BufferedReader` e cria um `Request`, uma classe que criamos, com a capacidade de armazenar informação necessária para fazermos o pedido ao servidor e também armazenar a resposta vinda do servidor.

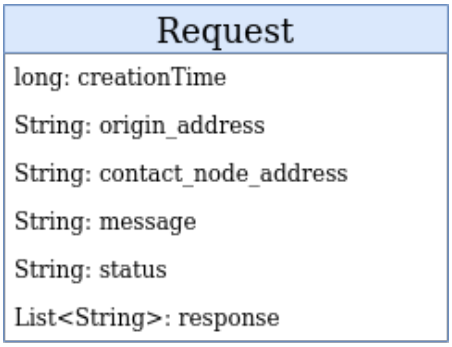

Figure 3: Classe Request

A classe `Request` vai ser inicializada com todos os valores a que já temos acesso, como o endereço do nodo e o tempo de criação, e com estado inicial `NA`, não atendido. Depois, preenche os valores que faltam, exceto a resposta, que só vai ser preenchida pelo servidor. Por fim, inicia uma thread `TCPReplier`, cuja função será responder ao cliente quando a resposta chegar do servidor, e uma thread `RequestHandler` que vai proceder à retransmissão da mensagem para o seguinte nodo. A esta thread, vamos oferecer uma socket UDP, `internalsocket` e uma porta protegida como argumentos.

2.2 Ligação Peer-to-Peer

A thread `RequestHandler` começa por criar uma string, identificador, para atribuir aos pacotes PDU. Esta string é composta pelo endereço do nodo e pelo número global do `Request`.

Através da função `serialize`, vai converter o Request num buffer de bytes. Com o tamanho desse buffer, define o número de pacotes PDU que necessita de criar, procede à criação e divide a carga de bytes entre eles. Vamos estudar a estrutura destes pacotes, numa secção posterior.

```java
public static byte[] serialize(Object obj) throws IOException {
    ByteArrayOutputStream out = new ByteArrayOutputStream();
    ObjectOutputStream os = new ObjectOutputStream(out);
    os.writeObject(obj);
    os.flush();
    os.close();
    byte[] b = out.toByteArray();
    out.flush();
    out.close();
    return b;
}
```

Figure 4: Função Serialize

Volta a converter os pacotes, que acabou de criar, num buffer de bytes, utilizando a mesma função `serialize`, e vai armazenando numa estrutura `Map`, para no caso de ser necessário o reenvio, termos tudo preparado. Por fim, cria um `DatagramPacket` com o endereço de um peer aleatório e a protected port, que definimos como 6666 e envia o pacote através do `internalsocket`.

No nodo escolhido aleatoriamente, a thread `UDPListener` está ativa, por isso, desde que não tenham havido falhas, os pacotes vão chegar sem problemas. No momento de receção, a thread utiliza a função `deserialize` para converter novamente os buffers de byte em pacotes PDU.

```java
private static Object deserialize(byte[] data) {
    Object o = null;
    try {
        ByteArrayInputStream in = new ByteArrayInputStream(data);
        ObjectInputStream is = new ObjectInputStream(in);
        o = is.readObject();
        is.close();
    } catch (Exception e){
        e.printStackTrace();
    }
    return o;
}
```

Figure 5: Função Deserialize

Nesta thread, os pacotes PDU que chegam e pertencem ao mesmo Request são armazenados em grupo dentro de um SortedSet. Escolhemos este tipo de estrutura porque os pacotes são sequênciais e numerados, por isso, é nos útil eles ficarem armazenados de forma ordenada. Depois, coloca o SortedSet de pacotes dentro de um Map e como chave, utiliza o identificador dos pacotes, isto, porque é o mesmo para todos os pacotes vindos do mesmo Request. Com esta estrutura Map, temos fácil acesso ao grupo de pacotes que queremos compilar para Request, visto que eles já estão organizados dentro de um SortedSet e estão agrupados pelo identificador.

2.3 Ligação Peer-to-Server

Quando acaba de receber todos os pacotes PDU que necessita para obter o Request, passar à fase de agregação dos pacotes. Para tal, pega em todos os pacotes armazenados no SortedSet e concatena todos os seus buffers num outro buffer de bytes ao qual aplica novamente a função deserialize. Assim, este nodo passa a ter um Request, mas antes de proceder ao envio para o servidor vai o colocar numa queue há espera da sua vez.

A thread TCPSpeaker está continuamente a verificar o tamanho da queue de Requests. Sempre que haja algum pedido na fila, retira da fila o primeiro, atualiza o seu estado para ad, atendido no destino e procede ao seu envio para o servidor através de um PrintWriter. O PrintWriter é enviado através de uma socket tcp, **external socket out** que tem como argumento o endereço do servidor destino e uma **outside port** cujo valor por norma é 80.

```
external_socket_out = new Socket(target_address, outside_port);
```

Figure 6: Inicializar Socket TCP para comunicar com o Servidor

2.4 Ligação Server-to-Peer

Agora que o servidor recebeu o pedido vindo do cliente, vai ter que fornecer a sua resposta. A thread TCPSpeaker, que permaneceu a correr, recebe a resposta em forma de BufferedReader, através da nesma socket tcp por onde enviou o pedido, **external socket out**. Depois, atualiza o estado do Request para sd, servido no destino, e insere a resposta do servidor na variável List<String> response do mesmo.

Como já tem o Request atualizado com a resposta, remove o pedido da queue e inicia uma thread RequestHandler para proceder ao encaminhamento no sentido inverso.

2.5 Ligação Peer-to-Peer

Nesta fase, o processo vai ser idêntico à comunicação udp anterior. O RequestHandler cria pacotes PDU capazes de armazenar o conteúdo da reposta e envia-os, sequencialmente, para o peer de onde veio o Request original, utilizado a socket udp, internal socket.

No primeiro nodo, a thread UDPListener está continuamente a correr e recebe os pacotes. Quando todos os pacotes chegarem, volta a realizar a conversão para Request e a armazenar numa queue, tal como explicado previamente.

2.6 Fim da Ligação

Agora que já temos o Request com a resposta do servidor na queue do primeiro nodo, o programa vai dar uso à thread TCPReplier que já tinhamos inicializado no início da ligação.

Esta thread, começa por verificar se existem Requests na queue. Se existir, pega no primeiro e atualiza do seu estado para **to**, a ser transmitido à origem. Depois, utiliza um BufferedWriter com um socket TCP para enviar para o cliente a resposta, existente no Request, vinda do servidor.

Por fim, como a resposta já chegou ao cliente, voltamos a atualizar o estado do Request, desta vez para **so**, servido na origem, e removemos da queue. Terminando assim o processo de pedido e resposta entre o cliente e o servidor.

3 Especificação do Protocolo UDP

3.1 Formato das mensagens protocolares (PDU)

Começamos por definir o tamanho máximo dos nossos pacotes (PDU), que será, por defeito, 20256 Bytes. Tem um cabeçalho com um tamanho variável até aos 256 bytes e ainda uma secção para os dados com um tamanho de 20 KBytes.

Definimos um conjunto de variáveis que achamos necessárias à resolução do problema.

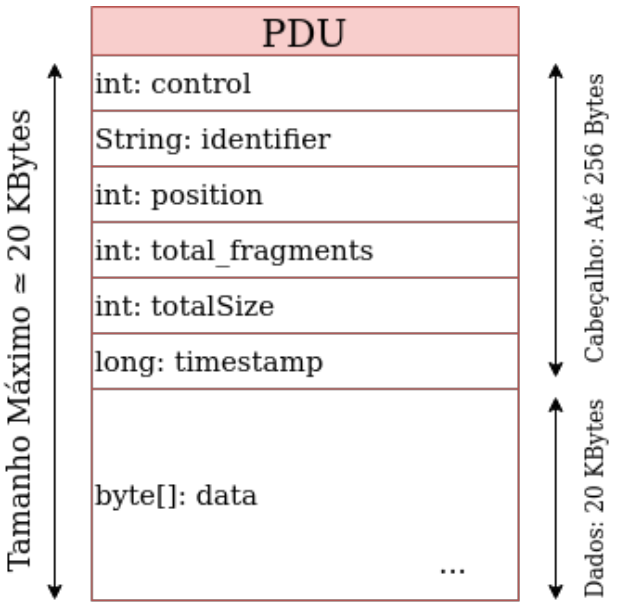

Figure 7: Pacote PDU

Temos a variável **control** porque decidimos fazer com que o nosso pacote seja genérico, no sentido de que vai ter a capacidade de realizar dois tipos de trabalho. Pode servir como **pacote de dados**, cuja função é encapsular mensagens de dados. Ou como **pacote de controlo** que se destina a encapsular mensagens de controlo. Se esta variável tiver o valor 0, faz o papel de pacote de dados, caso tenha o valor 1, serve como pacote de controlo.

A string **identifier** serve para identificar todos os pacotes criados a partir do mesmo Request. O seu valor é composto pela concatenação do endereço do nodo onde o pacote foi criado com o número global do Request, que é um valor que vamos incrementar sempre que um novo Request passa pelo nodo.

O tamanho do Request vai ser armazenado na variável **totalSize**. Tendo este valor, conseguimos calcular o número de pacotes que vamos necessitar e guardar em **total fragments**. Para além disso, vamos precisar de identificar qual a posição do pacote na sequência em **position**.

O momento da criação do pacote vai nos ser útil para realizar verificação e análise por isso vamos guardar essa informação em **timestamp**.

Por fim, temos a variável **data** que serve para armazenar a resposta do servidor.

3.2 Interações

Os pacotes que implementamos, vão trabalhar sobre vias de transmissão UDP.

Como temos dois tipos de pacote, controlo e dados, decidimos criar duas vias. Isto permite, de forma mais fácil, distinguir os diferentes pacotes que chegam a um determinado nodo. Ambas são implementadas sobre a forma de sockets mas têm portas diferentes.

No caso dos pacotes de dados, criamos uma **internal socket**, através da função atagram-Socket(protected port). O valor do **protected port** definimos como 6666. No caso dos pacotes de controlo, criamos uma **control socket** da mesma forma que criamos o socket anterior, mas neste caso com uma **control port** com valor 4646.

Antes de enviarmos os pacotes, utilizamos a função serialize para os converter em buffers de bytes. Depois, criamos um DatagramPackets onde introduzimos os buffers que acabamos de criar, o endereço do nodo destino e também a porta da socket por onde os vamos enviar. São estes DatagramPackets que vão ser enviados pelas vias UDP.

4 Implementação

4.1 Multiplexação

Um sistema deste tipo, para ser eficiente e privado, deve permitir que vários clientes o possam utilizar em simultâneo. Isto significa, que podem tentar comunicar com o servidor simultaneamente o que exige que a existência de multiplexação.

Para tal ser possível, um só mecanismo não é suficiente. Várias soluções, tais como queues tiveram que ser implementadas para que nenhum problema quebrasse o funcionamento livre de vários utilizadores. O mecanismo principal é a utilização de threads que estão constantemente ativas através de ciclos. Temos por exemplo a thread TCPListener que corre constantemente enquanto o nodo estiver ativo no sistema e que pode receber mensagens vindas de múltiplos clientes. Depois, converte essas mensagens em Requests que prosseguem o seu caminho ao longo do sistema.

```
public void startTCPListener() throws IOException {
    external_socket_in = new ServerSocket(this.outside_port);

    Thread listener = run() → {
            while (true) {
                Socket socket = null;
                try {
                    socket = external_socket_in.accept();
                    served++;
                    NodeTCPListener nl = new NodeTCPListener(socket, requests,replies,target_address,
                            my_address, internal_socket, peers,protected_port,served,control_port,control_socket);
                    new Thread(nl).start();
                } catch (Exception e) {
                    e.printStackTrace();
                }
            }
    };

    listener.start();
}
```

Figure 8: Thread TCPListener

4.2 Encriptação da Comunicação

Para garantir a segurança da informação contida nos nossos pacotes, tivemos que utilizar um método de encriptação. Assim, mesmo que a comunicação seja interceptada ao longo dos túneis UDP, que são simples e de baixo nível de segurança, é impossível para os "captores" obterem informação "legível".

O método de encriptação que utilizamos é um algoritmo AES com uma chave interna ao sistema. Todo o conteúdo do Request é encriptado antes da criação dos pacotes PDU e antes do seu envio. Para além disso, decidimos encriptar também o identificador dos pacotes PDU, assim torna-se impossível identificar o nodo de onde origina a mensagem.

```
r.setMessage(data,secretKey);
r.setContactNodeAddress(this.node_address,secretKey);
```

Figure 9: Exemplo de Encriptação de um Request

```
public synchronized void setMessage(String message, String secretKey) {
    this.message = AES.encrypt(message, secretKey);
}
```

Figure 10: Exemplo de Função com Encriptação

4.3 Controlo de Perdas

Quando um cliente recorre ao nosso sistema para efetuar alguma comunicação com um servidor de destino, espera que venha a obter uma resposta. Ao utilizarmos canais UDP para enviar os nossos PDU's entre nodos estamos sujeitos a perdas e/ou atrasos substanciais. Para isso implementámos um mecanismo que pede a retransmissão de fragmentos perdidos. O **NodeUDPListener** ao iniciar lança uma thread que irá correr um mecanismo **anti-stall**.Este mecanismo, consiste em efetuar verificações de 500 em 500 milisegundos. Estas verificações por sua vez consistem em comparar o **timestamp** de um PDU com o **timestamp** atual. Se a diferença for superior a **3500 milisegundos** este é marcado como **stalled**. É também verificado que o identificador do PDU não esteja já na lista de suspeitos ou na lista de PDU's/Requests já atendidos. Caso estas condições se verifiquem, o identificador é adicionado á lista de suspeitos. Aí é lançado um **PDUChecker** para resolver a falha. Este PDUChecker consiste numa thread com um ciclo que verifica sempre se os fragmentos já se encontram todos na linha de montagem. Caso não estejam todos ele levanta o número dos fragmentos em falta e procede para pedir a sua retransmissão. A retransmissão consiste apenas em enviar um **PDU** com o controlo assinalado a 1 e com o número do fragmento no campo de dados. Cada fragmento em falta resulta num PDU de controlo. Estes PDU's são enviado via um socket UDP de controlo por uma porta de controlo específica. O **RequestHandler** por sua vez após ter enviado os fragmentos não cessa a execução. Enquanto não receber um PDU de controlo a informar que do outro lado está tudo bem, fica À espera desse PDU. Ao receber os PDU's de controlo ele verifica o conteudo do payload. Sendo que o conteúdo é o número do fragmento em falta, ele consulta a lista dos fragmentos e reenvia esse fragmento. Esta operação é efetuada para todos os PDU's de controlo com um número no payload. Após os fragmentos estarem todos com sucesso no destino é finalmente enviado o PDU de controlo a informar o RequestHandler que tudo correu bem e que pode parar a execução.

4.4 Autenticação da Origem

As vias de conversação UDP não têm um elevado grau de seguranda, por isso, é fazível que um grau alheio ao sistema tente enviar um pacote cuja função seja maliciosa. Para evitar que este problema descarrile o nosso sistema tivemos que implementar um mecanismo de autenticação da

origem.

Este mecanismo, resume-se a termos uma função **validOrigin** que verifica se um novo pacote que chegou ao nodo tem origem válida. Um pacote tem origem válida, se o endereço que traz no seu identificador pertence à lista de endereços dos nossos peers.

```
private boolean validOrigin(String s){
    String[] ip = s.split( regex: "\\s+");
    if (peers.contains(ip[0])) {
        return true;
    } else {
        return false;
    }
}
```

Figure 11: Função ValidOrigin

5 Testes e Resultados

Para efeitos de teste consideramos alguns casos base. Para melhor visualizarmos e testarmos um maior número de fragmentos consideramos que para efeitos de teste, o tamanho máximo do payload de cada fragmento é de 20 bytes. Para induzir a falhar de fragmentos temos também uma versão de teste do RequestHandler chamada de **TestingRequestHandler**.

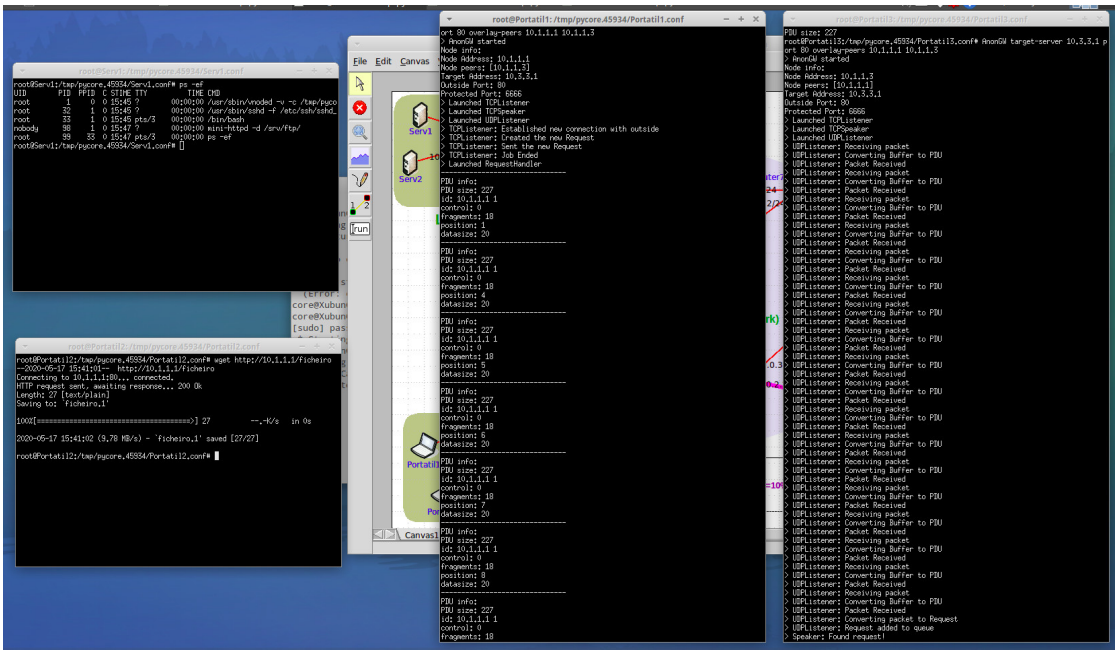

Figure 12: Teste Funcional Parte 1

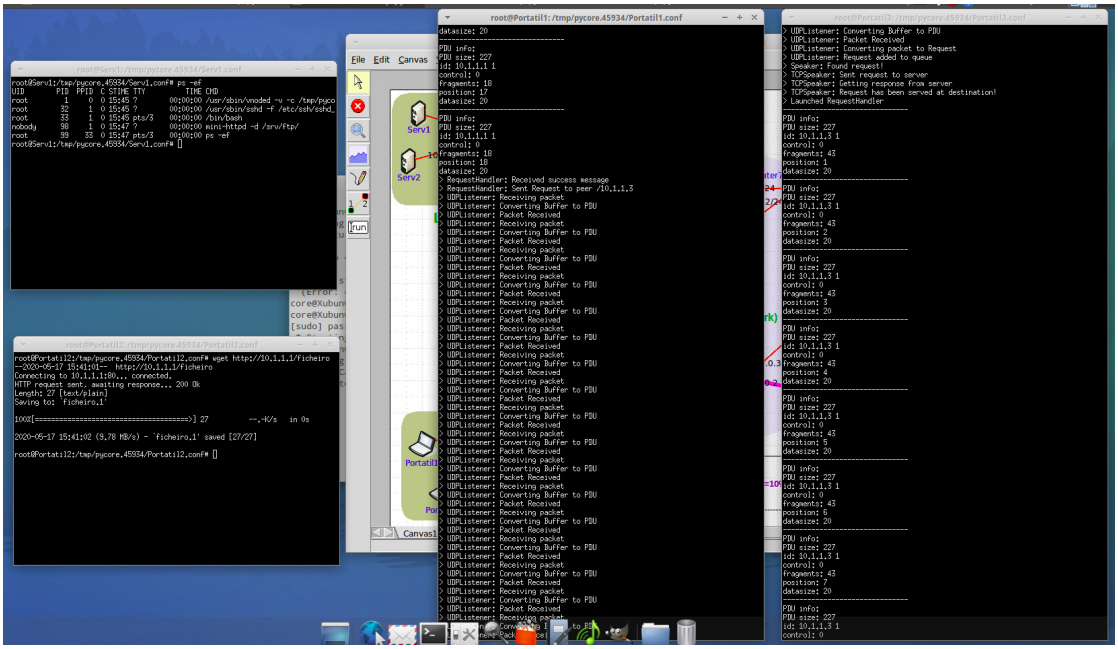

Figure 13: Teste Funcional Parte 2

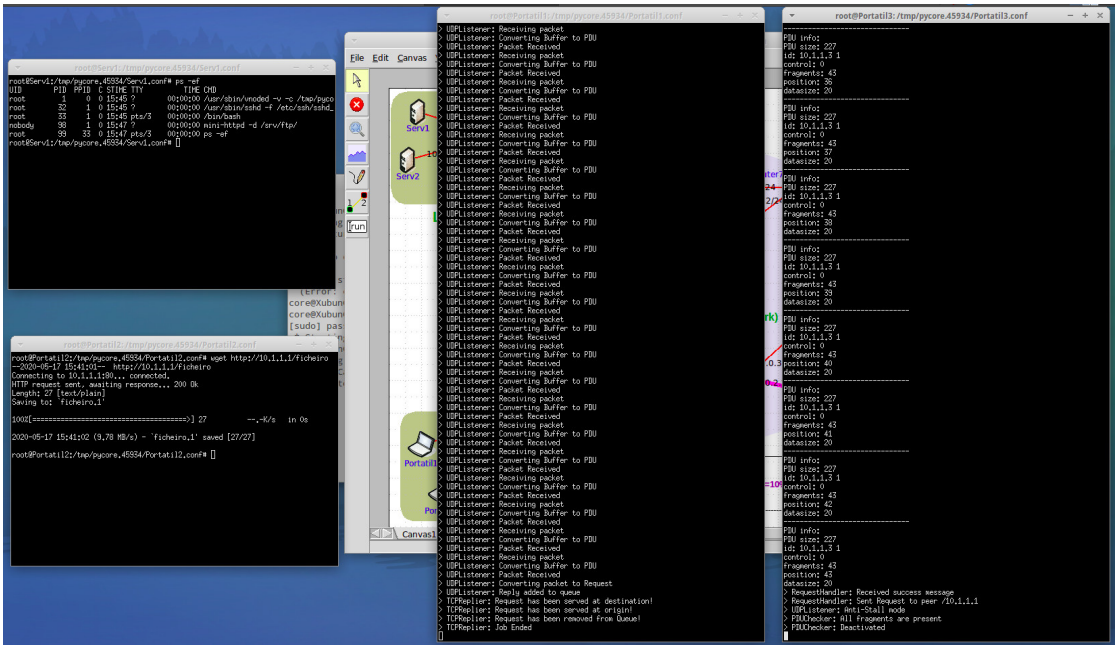

Figure 14: Teste Funcional Parte 3

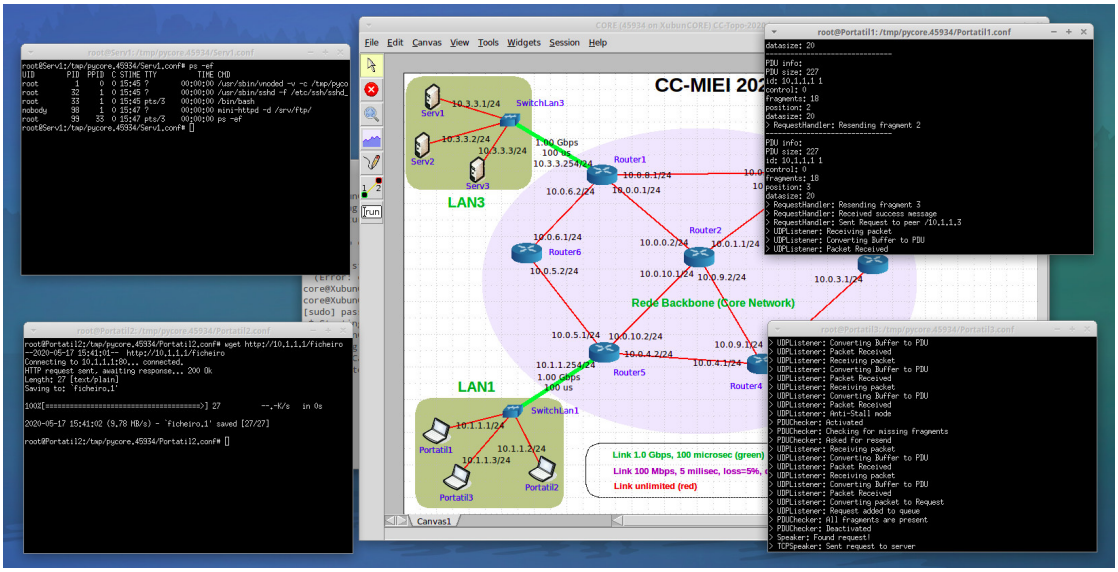

Figure 15: Teste do Controlo de Perdas

6 Conclusões e Trabalho Futuro

Através do uso de diferentes mecanismos, achamos que com sucesso conseguimos de definir uma tecnologia suficientemente especifica para combater todos as características que devem ser enfrentadas quando estamos a trabalhar com pacotes UDP.

Com a tecnologia desenvolvida conseguimos com sucesso aplicar anonimização de conexão ao longo de uma rede anonimizadora. Permitindo, desta forma, obter um nível moderado de segurança dos pacotes no seu percurso de origem-destino.

Como conclusão, temos que o sistema explicitado é capaz de estabelecer todas as necessários comportamentos, e extras, requisitados pelos docentes, fornecendo assim à sociedade um mecanismo capaz de lidar com uma complexidade moderada.

Como trabalho futuro, entendemos que as empreitadas mais promissoras no longo prazo seriam, notoriamente, as seguintes, com uma grande ênfase na escalabilidade do sistema:

- Tratamento de controlo de congestão, temática que até ao presente, propositadamente, foi ignorada.
- Extensão desta tecnologia para N nós com ligações reentrantes.
- Aplicação de um mecanismo de segurança mais sofisticado.