



Universidade do Minho
Escola de Engenharia

UNIVERSIDADE DO MINHO
MESTRADO INTEGRADO EM ENGENHARIA DE TELECOMUNICAÇÕES E
INFORMÁTICA

PROJETO DE TELECOMUNICAÇÕES E INFORMÁTICA I
REDE OVERLAY P2P DE ANONIMIZAÇÃO
RELATÓRIO FINAL

2 de fevereiro de 2021

DOCENTES:

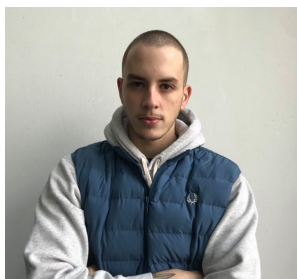
PROF. ANTÓNIO COSTA

PROF. HELENA RODRIGUES

Membros



Hugo Machado (a80362@alunos.uminho.pt)



Micael Vieira (a85659@alunos.uminho.pt)



Pedro Correia (a83666@alunos.uminho.pt)



Pedro Silva (a77854@alunos.uminho.pt)

Índice

1	Introdução	1
2	Descrição do projeto	2
2.1	Interpretação do problema	2
2.2	Requisitos	3
2.2.1	Requisitos funcionais	3
2.2.2	Requisitos não-funcionais	3
3	Fundamentos Teóricos	5
3.1	Redes P2P	5
3.2	Proxy	6
3.2.1	Socks	6
3.3	Criptografia	12
3.3.1	<i>Elliptic-curve Diffie-Hellman</i> (ECDH) [5]	12
3.3.2	<i>Cipher Block Chaining</i> (CBC) [6]	13
4	Desenvolvimento	15
4.1	Arquitetura de Rede	15
4.2	Protocolo de Comunicação	17
4.2.1	<i>BootStrapping</i>	17
4.2.2	Encaminhamento na Rede P2P	20
4.2.3	Manutenção da rede P2P	26
4.3	Arquitetura de Software do <i>Peer</i>	27
4.3.1	<i>ManagerThread</i>	29
4.3.2	<i>ProxyThread</i>	30
4.3.3	<i>ForwardingThreadPool</i>	31
4.3.4	<i>LocalThreadPool</i>	35
4.3.5	<i>ExitThreadPool</i>	35
4.4	Arquitetura de Software do Gestor	36
4.4.1	<i>MainThread</i>	37
4.4.2	<i>ClientThreadPool</i>	38

5	Testes, Resultados e Discussão	39
5.1	Requisitos para uso da rede	39
5.2	Testes e Resultados	41
5.3	Configuração	49
5.3.1	Configuração do Firefox Browser	49
5.3.2	Configuração do sistema para uso da rede	51
5.4	Discussão	52
5.4.1	Objetivos não cumpridos	52
5.4.2	Funcionalidades em falta	53
6	Conclusão	54

Índice de Imagens

1	First packet to server	6
2	Response packet from server	7
3	First packet to server	8
4	Client greeting	9
5	Server choice	9
6	Client connection request	10
7	SOCKS5 address	10
8	Response packet from server	11
9	Diagrama ECDH	13
10	Diagrama AES-CBC	14
11	Diagrama da Arquitetura da rede	16
12	Diagrama da troca de pacotes no Bootstrapping	17
13	Estrutura do pacote <i>Register</i>	18
14	Estrutura do pacote <i>Approval</i>	18
15	Estrutura do pacote <i>Connection to new</i>	19
16	Estrutura do pacote <i>Connection Request</i>	19
17	Estrutura do pacote <i>Connection Response</i>	19
18	Troca de pacotes para estabelecer uma conexão a um servidor através da rede	21
19	Exemplo de encaminhamento na rede P2P	22
20	Troca de pacotes para estabelecer uma conexão a um servidor através da rede	23
21	Estrutura dos pacotes <i>New, Response e Talk</i>	25
22	Estrutura do pacote <i>End</i>	26
23	Estrutura do pacote <i>Patch Up</i>	26
24	Arquitetura de <i>Software</i> de <i>Peer</i>	28
25	Fluxograma da <i>ProxyThread</i>	30
26	Fluxograma geral da <i>ForwardingThread</i>	32
27	Fluxograma de verificação do cabeçalho do pacote	33
28	Fluxograma do procedimento para o tipo <i>New</i>	33
29	Fluxograma do procedimento para o tipo <i>Response</i>	34
30	Fluxograma do procedimento para o tipo <i>Talk</i>	34
31	Fluxograma do procedimento para o tipo <i>End</i>	34

32	Diagrama da arquitetura do gestor	36
33	Erro caso não existam 4 clientes	39
34	Erro caso não existam 2 clientes	40
35	Topologia de rede utilizada	41
36	Sucesso ao aceder ao <i>e-learning</i>	42
37	Captura de tráfego Wireshark	43
38	Informações <i>exit node</i> / informações <i>client node</i>	44
39	Informações <i>exit node</i> / informações <i>client node</i>	45
40	Entrada dos nós na rede de anonimização	46
41	Saída de nós da rede	47
42	Reinserção de nós na rede	48
43	Configuração da <i>Proxy</i> do Firefox - Parte 1	49
44	Configuração da <i>Proxy</i> do Firefox - Parte 2	50
45	Configuração da <i>Proxy</i> do Firefox - Parte 3	50

Índice de Tabelas

1	Campos do pacote <i>First packet to server</i>	6
2	Campos do pacote <i>Response packet from server</i>	7
3	Campos do pacote <i>First packet to server (SOCKS4a)</i>	8
4	Campos do pacote <i>Client greeting</i>	9
5	Campos do pacote <i>Server choice</i>	9
6	Campos do pacote <i>Client connection request</i>	10
7	Campos da estrutura <i>SOCKS5 address</i>	10
8	Campos do pacote <i>Response packet from server</i>	11

1. Introdução

O presente relatório reflete o desenvolvimento de uma Rede *overlay* P2P (*peer-to-peer*) de Anonimização, realizado no âmbito da Unidade Curricular Projeto de Telecomunicações e Informática I.

Neste documento iremos descrever as decisões, implementações e conclusões em relação ao problema que é a anonimização dos utilizadores ao navegarem na Internet.

É do conhecimento geral que é uma das áreas mais negligenciadas em aplicações distribuídas e também das mais controversas. Isto porque ao mesmo tempo que se pretende proteger a privacidade dos utilizadores, também se procura proteger os serviços disponibilizados de usos maliciosos e o grupo de utilizadores que usufruem do mesmo serviço.

Os servidores que disponibilizam esses serviços possuem ficheiros de log que relatam todas as comunicações dos utilizadores, podendo muitas vezes utilizar essas informações para fins comerciais sem o consentimento dos utilizadores.

2. Descrição do projeto

2.1. Interpretação do problema

O problema originalmente proposto pelos docentes era a criação de uma rede overlay peer-to-peer de anonimização onde os nós dessa rede simplesmente encaminhariam pedidos HTTP de um cliente da rede e os efetuariam de facto para um certo servidor.

Porém, o grupo decidiu que em vez de existir clientes da rede de anonimização, a rede a ser construída iria ser composta por nós que fossem tanto clientes como encaminhadores da rede desenvolvida. Ou seja, cada nó da rede poderá efetuar pedidos para um determinado servidor e esses pedidos irão ser encaminhados por outros nós até a um nó final que irá efetuar a ligação ao servidor.

Em vez de existirem clientes da rede, todos os nós pertencentes à rede de anonimização poderão ser clientes da mesma, encaminhadores ou nós de saída.

A ideia é que quando um nó da rede aceda a um browser e comece a navegar pela Internet, todos os pedidos gerados sejam efetuados por um outro nó da rede.

Assim, em vez de somente estarmos a garantir anonimização para um certo serviço, pretendemos que os pedidos efetuados sejam todos realizados por outros nós de forma a que os servidores não tenham conhecimento de quem o originou, apenas de quem o efetuou.

2.2. Requisitos

2.2.1. Requisitos funcionais

- Cada nó da rede poderá ser cliente, encaminhador ou nó de saída (nó que efetua o pedido);
- Os nós só terão conhecimento dos vizinhos diretos, não possuindo qualquer conhecimento de outros nós da rede;
- Com os pedidos a serem realizados por um outro nó da rede, pretendemos obter a anonimização do nó que o originou, nomeadamente de não incluir o seu endereço IP no pedido;
- A estrutura da rede Overlay será edificada de forma a que o encaminhamento possa ser efetuado sem a inclusão de endereços IP nos cabeçalhos dos pacotes do nosso protocolo, evitando que um nó intermediário saiba quem o originou;
- Os nós/clientes da rede ao utilizarem um *WebBrowser* (por exemplo o Firefox), todo o seu tráfego irá ser intercetado por um *proxy local*. Para tal utilizaremos o protocolo *SOCKS* cujo funcionamento estará explicado posteriormente;
- Os pedidos HTTP/HTTPS originados irão ser encapsulados em pacotes do protocolo que o grupo implementou;
- Os pedidos serão encaminhados por três nós, ou seja, só darão três saltos antes de serem realizados por um nó qualquer da rede, de forma a que não seja perdida demasiada performance no acesso a um certo servidor;
- Um primeiro pedido não terá um caminho pré-definido, sendo este decidido em cada nó para onde seja enviado. Isto será possível devido à estrutura da rede e ao algoritmo de encaminhamento elaborado;
- Após o caminho estar estabelecido, pretendemos extrair a partir deste, um caminho inverso, para que a resposta seja encaminhada para o nó que originou o pedido.

2.2.2. Requisitos não-funcionais

- Fiabilidade na troca de dados entre um nó que pretende juntar-se à rede e o gestor, através do protocolo de comunicação desenvolvido, e que este é corretamente posicionado na rede de acordo com a arquitetura adotada;
- Fiabilidade na captura de tráfego entre o *WebClient* e o *WebBrowser*;

- Fiabilidade da conexão entre nós vizinhos e a correta troca de dados entre eles, através do protocolo de comunicação desenvolvido;
- Desempenho na interpretação dos pacotes recebidos e determinação da próxima direção de encaminhamento;
- Correta encriptação dos dados na troca de pacotes entre nós vizinhos, ou na troca de pacotes entre um nó e o gestor.

3. Fundamentos Teóricos

3.1. Redes P2P

Uma rede P2P [2] (*Peer-to-Peer*) trata-se de uma arquitetura de rede onde cada computador (denominado de nó) funciona simultaneamente como cliente e servidor, permitindo a partilha de dados sem a necessidade de um servidor central. Por não se basear em uma arquitetura cliente-servidor, onde apenas o servidor é responsável pela execução de todas as funções da rede, a rede P2P tem uma enorme vantagem por não depender de um servidor. Todos os nós estão interconectados permitindo o acesso a qualquer nó a partir de qualquer nó.

Na implementação de uma rede P2P podem ser seguidas três abordagens no que consta à estrutura da mesma:

- Rede P2P não-estruturada;

Este tipo de redes P2P não impõem uma determinada estrutura na rede *Overlay*, é sim formada por nós que formam aleatoriamente conexões entre si. Como não existe uma estrutura globalmente imposta, as redes não-estruturadas são fáceis de construir e são altamente robustas em situações em que um grande número de nós se conecta ou desconecta da rede frequentemente.

- Rede P2P estruturada;

Em redes P2P estruturadas, a rede *Overlay* é organizada numa topologia específica, e o protocolo implementado garante que qualquer nó possa procurar de forma eficiente um dado ficheiro/recurso que se encontre noutro nó. Porém para manter um encaminhamento de tráfego eficiente cada nó deve manter uma lista dos seus vizinhos.

- Rede P2P híbrida.

Em redes P2P híbridas são uma "mistura" do modelo P2P com o modelo cliente-servidor, e será esta a abordagem que iremos seguir na implementação da nossa rede *Overlay*. Um modelo comum de uma rede P2P híbrida é ter um servidor central que serve como ajuda para os nós se encontrarem uns aos outros. Atualmente este tipo de estrutura é a que apresenta um maior desempenho.

3.2. Proxy

Uma Proxy [3] é um servidor que age como intermediário para um cliente que pretende comunicar com algum servidor, por exemplo, para carregar uma página *Web*. Quando uma máquina se liga a um Proxy "obedece" às regras definidas por esta e todos os pedidos são feitos pelo próprio Proxy, que posteriormente os devolve ao cliente. Estando o cliente ligado à Internet através de um Proxy, o endereço IP deste não passa para além do Proxy.

3.2.1. Socks

O socks é um protocolo de internet que faz a troca de pacotes entre o cliente e o servidor através de um servidor proxy. Demos uso a este protocolo para que em cada cliente se possa intercetar todo o tráfego *web* localmente, visto que os browsers conseguem comunicar utilizando este protocolo. Desta forma conseguimos intercetar tanto pedidos de domínio (DNS), como pedidos ipv4 e ipv6. Assim os pedidos que este cliente iria realizar, irão ser reencaminhados para outros clientes da rede para que estes façam o pedido por ele.

Atualmente existem algumas versões do SOCKS [4], na realização do projeto procuramos a implementação das seguintes:

- SOCKS4:

1. Inicialmente é enviado um pacote do cliente para estabelecer uma conexão:

First packet to server

	VER	CMD	DSTPORT	DSTIP	ID
Byte Count	1	1	2	4	Variable

Figura 1: First packet to server

VER	Indica a versão socks, neste caso 0x04	
CMD	0x01	estabelecer uma conexão TCP/IP
DSTPORT	Porta (<i>network byte order</i>)	
DESTIP	Endereço IPV4 (<i>network byte order</i>)	
ID	(Não utilizado)	

Tabela 1: Campos do pacote *First packet to server*.

2. De seguida o servidor envia um pacote de resposta:

Response packet from server				
	VN	REP	DSTPORT	DSTIP
Byte Count	1	1	2	4

Figura 2: Response packet from server

VN	Byte nulo	
REP	0x5A	Pedido aceite
	0x5B	Pedido rejeitado ou sem sucesso
	0x5C	(Não utilizado)
	0x5D	(Não utilizado)
DSTPORT	Porta de destino	
DESTIP	Endereço IP de destino	

Tabela 2: Campos do pacote *Response packet from server*.

- SOCKS4a

A versão SOCKS4a estende a versão SOCKS4, permitindo ao cliente fazer pedidos DNS para além dos pedidos de *IPV4*.

1. Se o cliente desejar realizar um pedido DNS, irá utilizar o mesmo pacote que é utilizado na versão SOCKS4, mas desta vez, terá de indicar no campo "*DSTIP*" um endereço no formato "0.0.0.x", em que a última posição contém um *byte* diferente de 0. De seguida, após o "*ID*" que é terminado por um "*byte*" nulo deverá ser introduzido o domínio, terminado com um "*byte*" nulo.

First packet to server

	SOCKS4_C	DOMAIN
Byte Count	8+variable	variable

Figura 3: First packet to server

SOCKS4_C	Formato igual ao primeiro pacote da versão SOCKS4
DOMAIN	Domínio a que se pretende conectar, terminado com " <i>byte</i> " nulo

Tabela 3: Campos do pacote *First packet to server (SOCKS4a)*.

- SOCKS5

A versão do SOCKS5 representa uma extensão que não é compatível com a versão SOCKS4, oferece mais formas de autenticação e suporta IPv6, UDP e permite pedidos de DNS.

1. O cliente conecta-se e envia um pacote com uma lista de métodos de autenticação suportados:

Client greeting			
	VER	NAUTH	AUTH
Byte count	1	1	variable

Figura 4: Client greeting

VER	Indica a versão socks, neste caso 0x05	
NAUTH	Número de métodos de autenticação suportados	
AUTH	0x00 ...	Sem autenticação (Mais nenhum método foi utilizado)

Tabela 4: Campos do pacote *Client greeting*.

2. O servidor seleciona um método de autenticação do cliente:

Server choice		
	VER	CAUTH
Byte count	1	1

Figura 5: Server choice

VER	Indica a versão socks, neste caso 0x05
CAUTH	Método de autenticação aceite, ou 0xFF se nenhum método é aceite

Tabela 5: Campos do pacote *Server choice*.

3. Caso o servidor utilize algum tipo de autenticação, são trocados 2 pacotes, *Client authentication request* e *Server response*, na nossa abordagem, não utilizamos qualquer tipo de autenticação, pelo que não existe a troca destes pacotes.

4. De seguida o cliente envia um pacote com um pedido de conexão:

Client connection request					
	VER	CMD	RSV	DSTADDR	DSTPORT
Byte Count	1	1	1	Variable	2

Figura 6: Client connection request

VER	Indica a versão socks, neste caso 0x05	
CMD	0x01	Estabelecer uma conexão TCP/IP
	0x02	(Não utilizado)
	0x03	(Não utilizado)
RSV	Byte nulo	
DSTADDR	Endereço destino (numa estrutura explicada abaixo)	
DSTPORT	Porta (<i>network byte order</i>)	

Tabela 6: Campos do pacote *Client connection request*.

A estrutura de um endereço SOCKS5 é a seguinte:

SOCKS5 address		
	TYPE	ADDR
Byte Count	1	variable

Figura 7: SOCKS5 address

TYPE	0x01	Endereço IPv4
	0x03	Nome de domínio
	0x04	Endereço IPv6
ADDR	4 bytes se for IPv4	
	1 byte para o tamanho do domínio, seguido de 1 a 255 bytes de domínio	
	16 bytes se for IPv6	

Tabela 7: Campos da estrutura *SOCKS5 address*.

5. Por fim o servidor responde com o seguinte pacote:

Response packet from server					
	VER	STATUS	RSV	BNDADDR	BNDPORT
Byte Count	1	1	1	variable	2

Figura 8: Response packet from server

VER	Indica a versão socks, neste caso 0x05	
STATUS	0x00	Sucesso ao estabelecer conexão
	0x02	Falha ao estabelecer conexão
	...	(Mais nenhum código foi utilizado)
RSV	Byte nulo	
BNDADDR	Endereço SOCKS5 (tal como representado anteriormente)	
BNDPORT	Porta (<i>network byte order</i>)	

Tabela 8: Campos do pacote *Response packet from server*.

3.3. Criptografia

De uma maneira geral, criptografia é um conjunto de regras, princípios e protocolos que transformam a transmissão de dados numa comunicação segura na presença de terceiros, normalmente denominados de "adversários". No âmbito deste projeto a criptografia é implementada entre cada par de nós de forma a garantir a confidencialidade, integridade e a autenticidade dos dados recebidos por cada um destes nós.

3.3.1. *Elliptic-curve Diffie–Hellman (ECDH)* [5]

No uso de uma cifra simétrica a mesma chave é utilizada para a operação de encriptação/desencriptação, a dita chave de sessão. Uma das maiores vulnerabilidades desta cifra é a distribuição da chave de sessão de uma forma segura. Uma das formas de atingir isto é a utilização de técnicas de cifra assimétrica, ou seja, cifras onde cada entidade gera um par de chaves distintas, uma chave pública que pode ser vista por todos, e uma chave privada que deve ser mantida em segredo.

Para a distribuição da chave de sessão entre dois nós constituintes da rede utilizamos a técnica ECDH, uma variante do clássico protocolo Diffie-Hellman, que utiliza curvas elípticas. Suponhamos que existem dois nós que pretendem partilhar uma chave de sessão, denominados por Alice e Bob:

- Alice gera uma chave privada d_A e uma chave pública $Q_A = d_A G$ (onde G é o gerador para a curva elítica);
- De forma análoga, Bob também gera uma chave privada d_B e uma chave pública $Q_B = d_B G$;
- Se Bob envia a sua chave pública para a Alice, este consegue calcular $d_A Q_B = d_A d_B G$, tal como Bob com a chave pública de Alice consegue calcular $d_B Q_A = d_A d_B G$;
- A chave de sessão que ambos vão utilizar para encriptar/desencriptar mensagens trocadas entre ambos é a ordenada (x) do ponto calculado $d_A d_B G$.

Mesmo que um "adversário" intercete as chaves públicas será incapaz de calcular a chave de sessão.

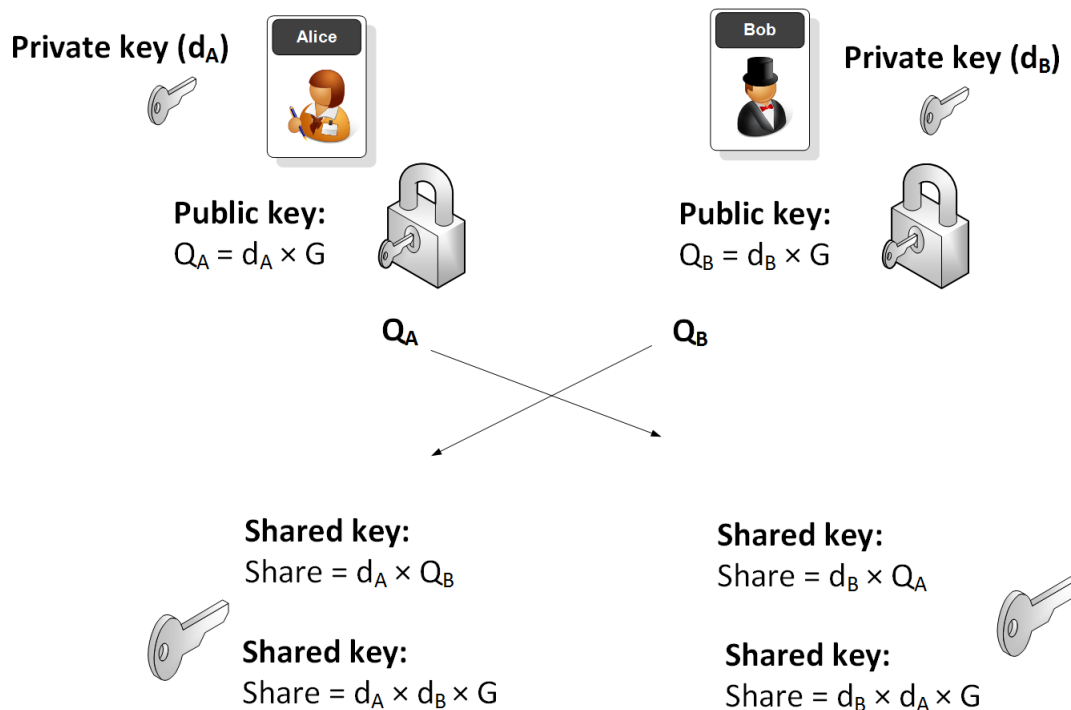


Figura 9: Diagrama ECDH

3.3.2. Cipher Block Chaining(CBC) [6]

Após obtida a chave de sessão através do protocolo ECDH esta é utilizada para a encriptação dos dados. Para tal utilizamos o modo de operação AES-CBC. AES trata-se de um algoritmo de encriptação que utiliza cifra em bloco, ou seja, uma cifra que opera sobre um número fixo de bytes denominados de "blocos". No nosso caso estes blocos têm o tamanho fixo de 16 bytes, logo, o tamanho dos pacotes encaminhados entre nós tem um tamanho múltiplo de 16.

Neste modo de operação cada bloco de texto limpo (*plaintext*) antes de ser encriptado é "XORed" com o bloco do criptograma anterior. Desta forma, cada bloco do criptograma depende de todos os blocos de texto limpo processados até ao momento. Para que esta cifra seja não determinística, ou seja, de modo a que cada bloco do criptograma seja único, mesmo com a mesma mensagem e a mesma chave de sessão, é introduzida uma variável aleatória (IV) que é "XORed" com o primeiro bloco de texto limpo.

Caso o bloco de texto limpo a ser encriptado não possua o comprimento (em *bytes*) correto, deve-se proceder ao uso de *padding*.

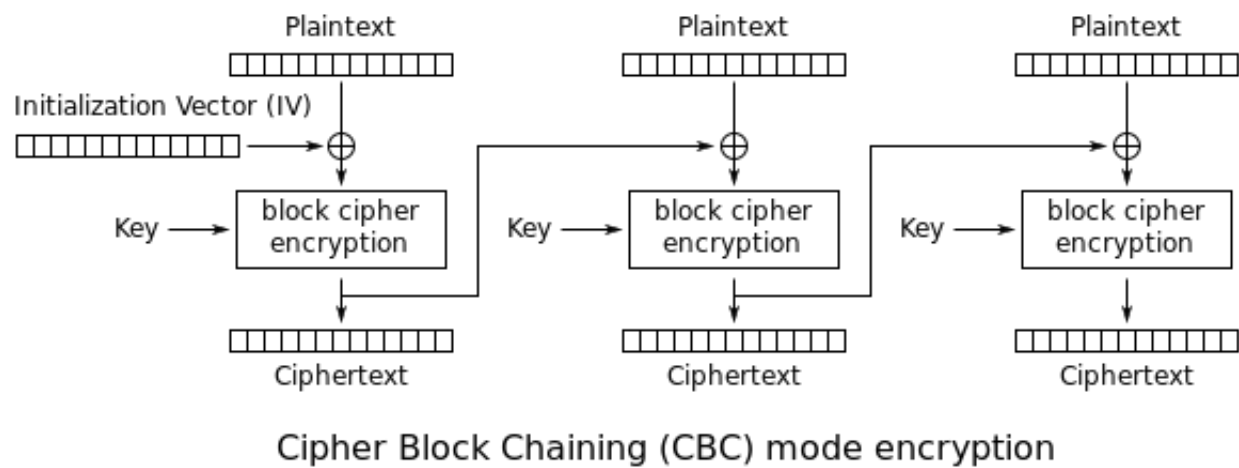


Figura 10: Diagrama AES-CBC

4. Desenvolvimento

4.1. Arquitectura de Rede

A rede que o grupo decidiu implementar possui uma estrutura híbrida, uma vez que irá ser constituída por nós da rede ("peers") e por um nó central cuja funcionalidade é gerir a estrutura da rede e as ligações existentes entre nós, ao qual o identificamos como *Manager* (gestor da rede). Todos os nós que se pretendam conectar à rede *Overlay* de anonimização irão ter que se conectar ao *Manager* para que este os coloque na rede e indique aos nós já pertencentes para se conectarem ao novo nó.

A estrutura de rede em que o *Manager* irá dispor os nós será uma rede em matriz. O facto do formato da rede ser uma matriz irá permitir que o encaminhamento de pacotes seja efetuado sem ser necessário a inclusão de uma identificação global dos nós por onde os pacotes são encaminhados. Em vez disso, só será necessário colocar as direções que um pacote toma em relação a cada nó por onde é encaminhado até ao nó final(*exit node*), evitando assim que o nó que efetua o pedido saiba quem o originou.

As direções a que nos referimos é a posição do nó vizinho em relação ao nó que irá efetuar o encaminhamento do pacote para esse vizinho. Estas direções podem ser esquerda, cima, direita ou baixo, uma vez que os nós irão formar uma matriz entre eles, e serão codificadas em binário da seguinte maneira para serem colocadas nos cabeçalhos dos pacotes:

- Cima \rightarrow 00 (0)
- Direita \rightarrow 01 (1)
- Baixo \rightarrow 10 (2)
- Esquerda \rightarrow 11 (3)

Para que esta infraestrutura de rede funcione corretamente, definimos três tipos de nós que irão possuir números diferentes de conexões com vizinhos :

1. O "*Vertex Node*", este é o primeiro nó da matriz posicionado no canto superior esquerdo, será único e terá apenas dois vizinhos (à direita e em baixo);
2. O "*Edge Node*", este é um nó de aresta da matriz, e será posicionado pela aresta esquerda e superior da matriz, e terá apenas três vizinhos (se for colocado na aresta superior terá vizinhos à esquerda, direita e em baixo; se colocado na aresta esquerda possuirá vizinhos em cima, à direita e em baixo);

3. O "*Middle Node*", este é um nó intermédio da matriz e possuirá quatro vizinhos (em todas as direções).

Na seguinte figura 11 está um diagrama que ilustra a nossa arquitetura de rede.

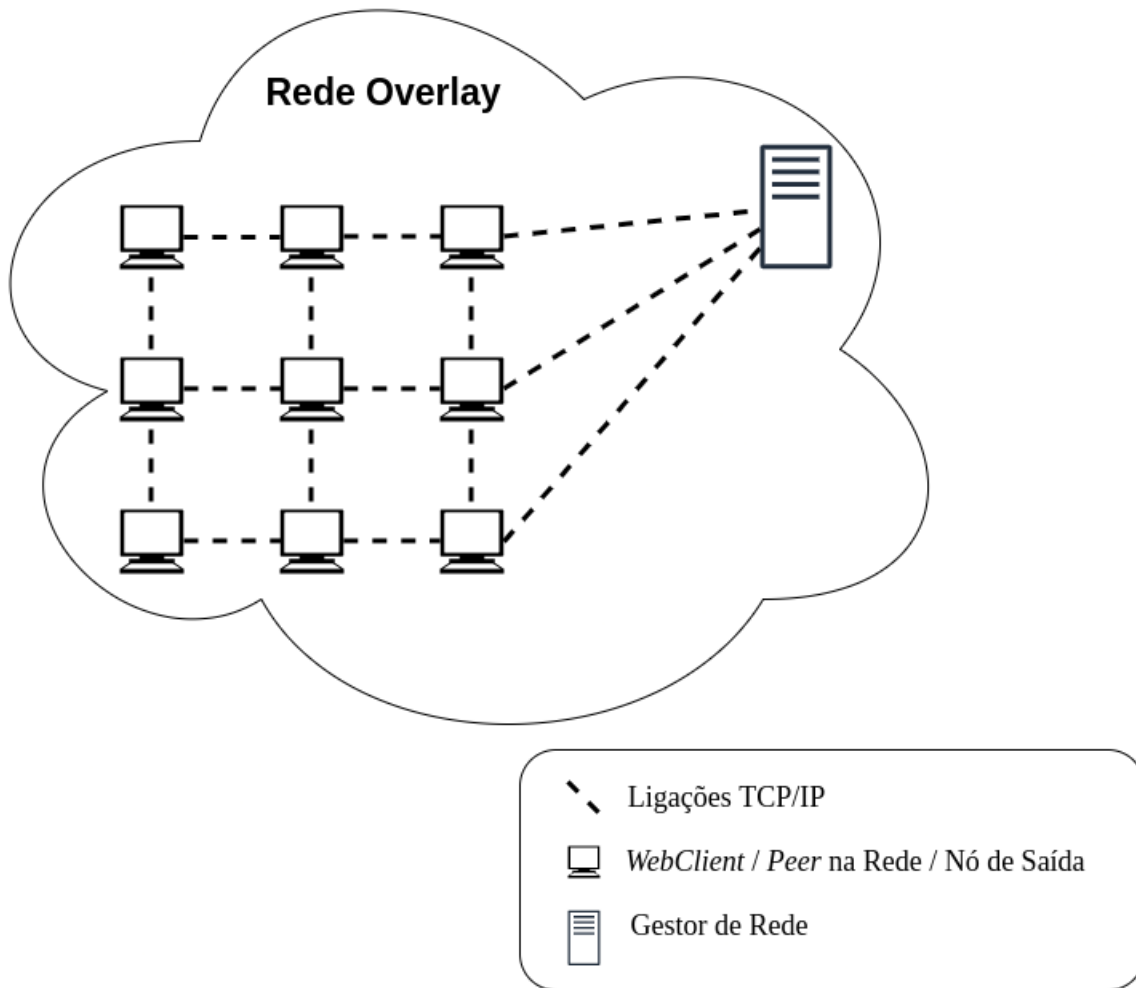


Figura 11: Diagrama da Arquitetura da rede

4.2. Protocolo de Comunicação

4.2.1. *BootStrapping*

Quando existe um novo cliente que se pretenda conectar à rede *Overlay*, este em primeiro lugar conecta-se ao gestor e envia um pacote do tipo *Register*.

O gestor após receber uma nova conexão, interpreta o pacote recebido e caso este seja um pacote de *Register* irá consultar a matriz existente para o colocar no lugar correto. De seguida irá responder com um pacote do tipo *Approval* informando o novo cliente do seu tipo e do número de ligações que deverá esperar de outros membros da rede.

Por fim, o gestor irá avisar os nós vizinhos já existentes para se conectarem ao novo nó, através de um pacote *Connect to New* que indica o endereço IP do novo nó e a direção destes em relação ao novo nó. Quando os vizinhos recebem esse pacote irão enviar um pacote *Connection Request* para o novo nó, que contém a direção deles em relação a este. O novo nó em resposta envia um pacote de *Connection Response*.

Na figura 12 demonstramos um exemplo da troca de pacotes entre os elementos da rede quando um novo nó pretende se conectar à rede.

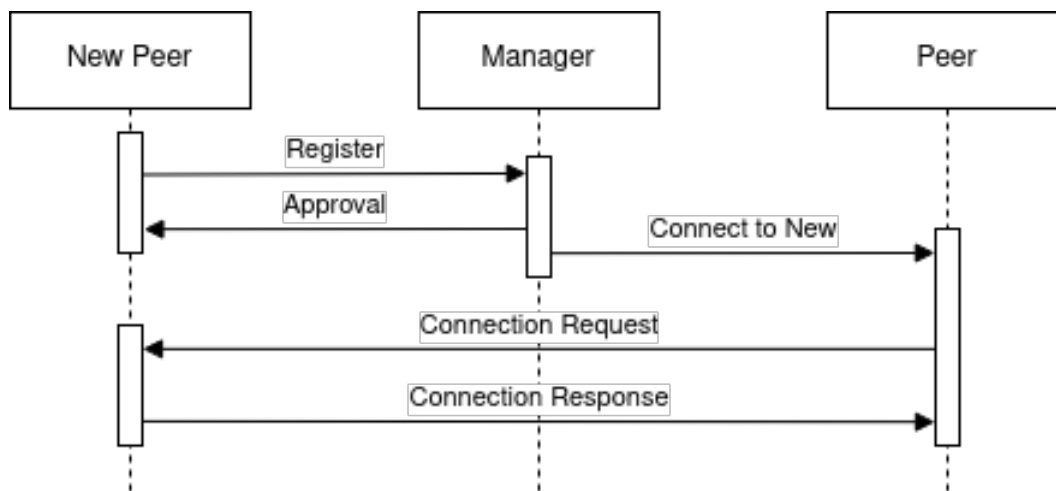


Figura 12: Diagrama da troca de pacotes no Bootstrapping

A constituição dos pacotes utilizados quando um nó se pretende conectar à rede é a seguinte:

1. *Register*:

- **Type** → tipo do pacote em questão, neste caso, será do tipo *Register*- valor 0;
- **Client ID** → identificador único do cliente, ou seja, do nó que se quer conectar à rede;

	TYPE	CLIENT ID
BYTE COUNT	1 byte	2 bytes

Figura 13: Estrutura do pacote *Register*

2. *Approval*:

- **Type** → tipo do pacote em questão, neste caso, será do tipo *Approval*- valor 1;
- **Node Type** → determina o número de vizinhos que o novo nó irá ter. Dependendo deste número o novo nó pode ser um *Vertex Node*(2), *Edge Node*(3) ou um *Middle Node*(4) (secção 4.1).

Caso o processo de registo não seja bem sucedido este campo irá assumir o valor 0.

- **Awaited Connections** → número de ligações que o novo nó espera receber, provenientes de nós já existentes na rede;

	TYPE	NODE TYPE	AWAITED CXN
BYTE COUNT	1 byte	1 byte	1 byte

Figura 14: Estrutura do pacote *Approval*

3. *Connect to New*:

- **Type** → tipo do pacote em questão, neste caso, será do tipo *Connect to New*- valor 2;
- **Direction** → a posição do nó já pertencente à rede em relação ao novo nó;
- **IP Addr** → endereço IP do novo nó.

	TYPE	DIRECTION	IP Addr
BYTE COUNT	1 byte	1 byte	4 bytes

Figura 15: Estrutura do pacote *Connection to new*4. *Connect Request*:

- **Direction** → a posição do nó já pertencente à rede em relação ao novo nó;
- **Public Key** → chave pública ECDH do nó já pertencente à rede.

	DIRECTION	PUBLIC KEY
BYTE COUNT	1 byte	91 bytes

Figura 16: Estrutura do pacote *Connection Request*5. *Connect Response*:

- **Public Key** → chave pública ECDH do novo nó.

	PUBLIC KEY
BYTE COUNT	91 bytes

Figura 17: Estrutura do pacote *Connection Response*

4.2.2. Encaminhamento na Rede P2P

O encaminhamento do tráfego *Web* na rede *Overlay* está dividido em três fases diferentes: o estabelecimento da ligação ao *WebServer*, a comunicação entre o *WebServer* e *WebClient* e a finalização da ligação.

A primeira fase, começa com o *browser* de um nó a efetuar uma conexão com o servidor proxy SOCKS local do mesmo nó.

Depois de efetuada a conexão, o *browser* faz um pedido SOCKS *Connection Request* ao servidor *proxy* SOCKS local, quando esse nó pretende aceder a um serviço da *Web*. Este pacote é encapsulado num pacote *New* do nosso protocolo.

O nó origem atribui uma identificação à respectiva *Stream* e escolhe de forma aleatória, dos vizinhos disponíveis, uma direção para o pacote ser encaminhado. Este regista-a no próprio pacote e encaminha-o, sendo o mesmo feito durante os dois nós seguintes (total de três saltos).

Quando chega ao ultimo nó, este por sua vez, adquire o endereço ou domínio DNS contido no pacote *Connection Request* que estava encapsulado no pacote *New* e tenta estabelecer uma conexão ao respetivo servidor. No caso de sucesso, este guarda o par caminho e identificação da *Stream* presente no pacote e responde ao nó origem, pelo caminho inverso, com um pacote de *Response* com a identificação da *Stream* e o bit que identifica o sucesso da conexão com o valor um. Caso contrário esse bit vai a zero.

Na chegada do pacote *Response* ao nó de origem, em caso de sucesso, o caminho e a identificação da *Stream* são armazenados e é enviado para o browser um pacote *Response packet from server* do protocolo SOCKS com o campo *status* com o valor 0x00 (caso seja utilizada a versão SOCKS5), ou 0x5A (no caso da versão SOCKS4), que indica que a conexão ao servidor pretendido foi bem sucedida.

No diagrama da figura 18 podemos verificar o exemplo da troca de pacotes para estabelecer uma conexão a um *Web Server* através da rede de anonimização.

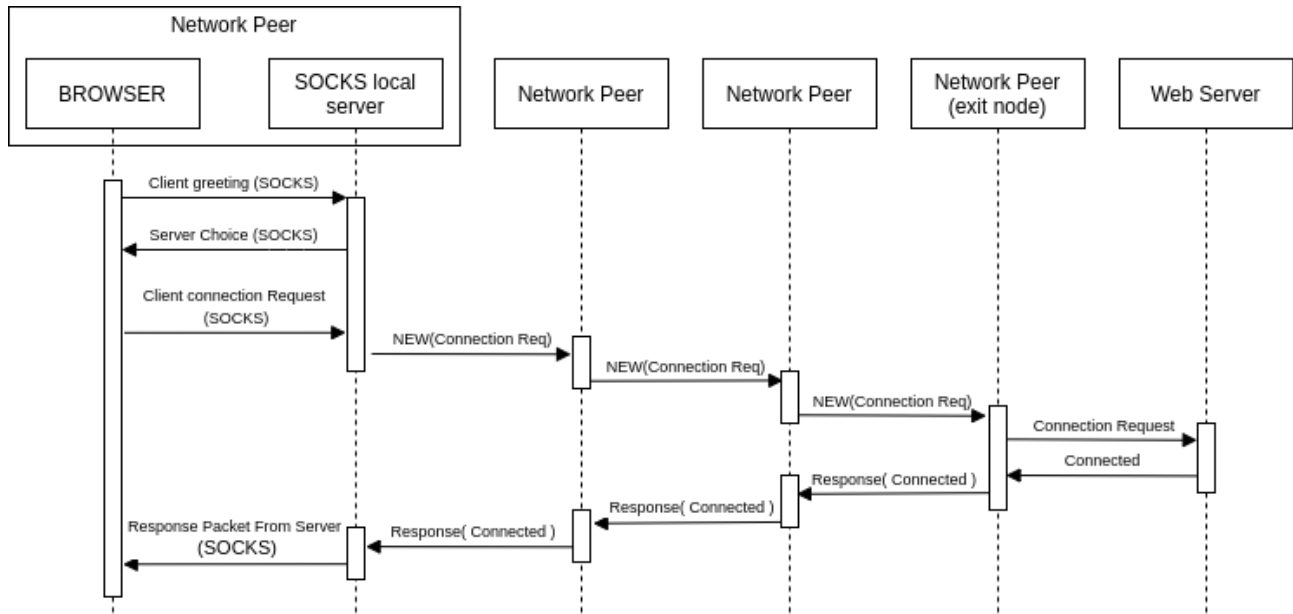


Figura 18: Troca de pacotes para estabelecer uma conexão a um servidor através da rede

Na segunda fase, após a conexão estar estabelecida, todo o tráfego *Web* entre o nó origem e o *WebServer* é encapsulado em pacotes *Talk*. Estes pacotes são encaminhados pela rede através do caminho calculado pelo pacote *New* e o destino deles é determinado através de um bit a zero ou a um, significando *Local* (nó origem) ou *Exit* (nó de saída), respetivamente. Cada um destes pacotes possui também uma identificação da *Stream* para identificar cada "conversa".

Na figura 19, está um exemplo de encaminhamento de tráfego *Web*, constituído por dois pedidos HTTP originados pelo mesmo nó mas com nós de saída diferentes.

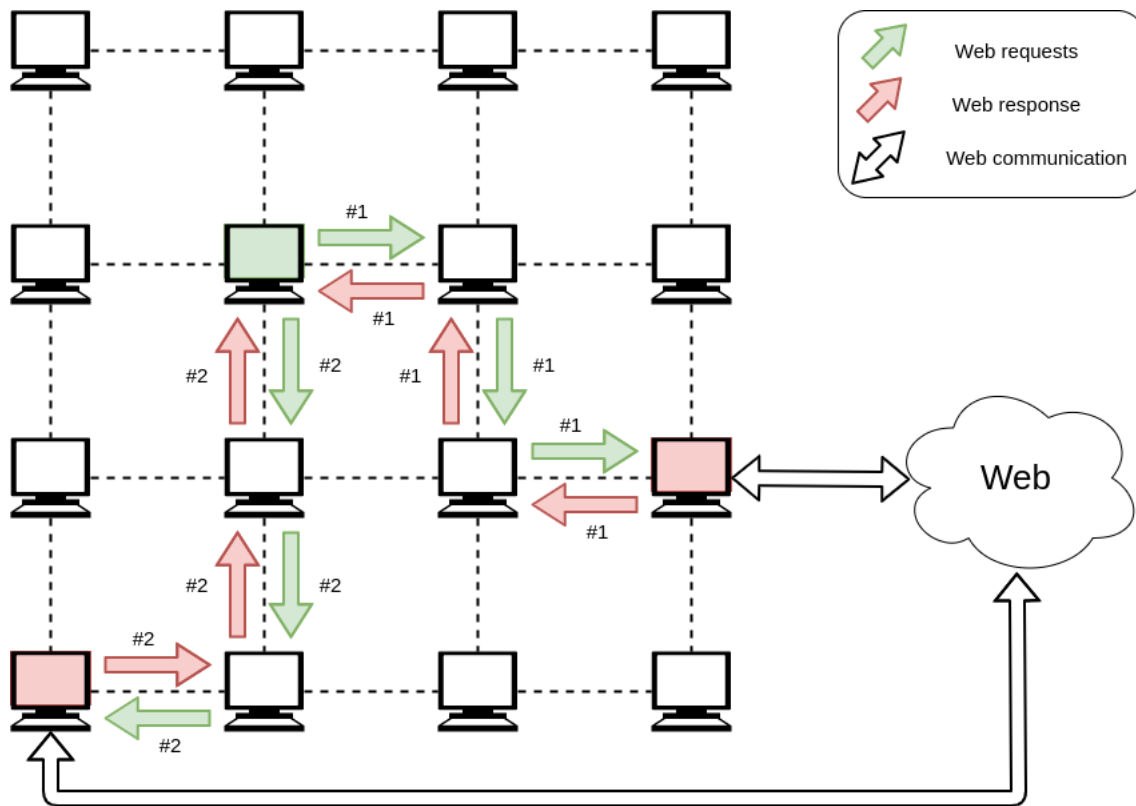


Figura 19: Exemplo de encaminhamento na rede P2P

Por fim, a terceira fase está associada à finalização da ligação. Esta pode ser evocada por duas razões diferentes. Pode ser terminada normalmente, quando o *WebClient* fecha a ligação no lado dele, sendo enviado um pacote *End* em direção ao nó de saída (através do caminho) com a identificação da respectiva *Stream*. Caso a ligação seja terminada de forma anormal, no nó origem ou no nó de saída, o procedimento é o mesmo. Nestes casos o par da identificação da *Stream* e caminho são eliminados do nó origem e nó destino.

Na figura 20, está um exemplo da troca de pacotes quando uma ligação termina, de forma normal ou anormal.

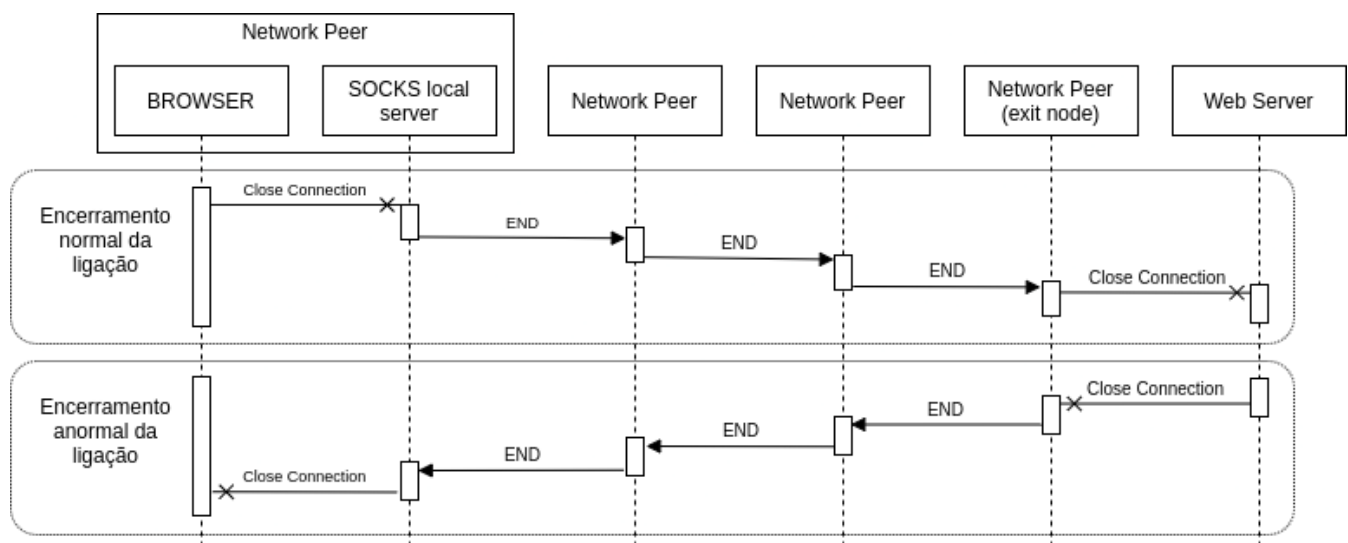


Figura 20: Troca de pacotes para estabelecer uma conexão a um servidor através da rede

Como se pode deduzir pelas informações que já fornecemos acerca do protocolo de encaminhamento, os pacotes são encaminhados por três nós ,ou seja, efetuam três saltos antes de chegarem a um nó de saída ou a um nó de origem.

Os pacotes utilizados no encaminhamento estão descritos de seguida:

1. *New*:

- **Circuit** → à medida que o pacote é encaminhado entre *peers* este campo é preenchido, dos bits mais para os menos significativos, com o caminho (direções) que o pacote toma;
- **Hops** → número de saltos (pré-definido) que o pacote deve dar antes de atingir o *exit node*. Este campo é decrementado a cada salto;
- **Type** → tipo do pacote em questão, neste caso, será do tipo *New* - valor 0;
- **Bit** → campo reservado para informação adicional, no caso do pacote *New* este não é utilizado, portanto segue com o valor **NULL**;
- **Stream ID** → Identificador único da *socket*;
- **Payload Length** → Tamanho do *payload* do pacote, tendo como valor máximo 1020 *bytes*;
- **Payload** → *Payload* que contém o pacote **Client connection request** (secção 3.2.1) encapsulado.

2. *Response*:

- **Circuit** → campo preenchido com o inverso do caminho que o pacote percorreu até ao *exit node*, de forma a que este pacote de resposta possa voltar ao nó que originou o pedido (*client node*);
- **Hops** → número de saltos (pré-definido) que o pacote deve dar antes de atingir o *client node*. Este campo é decrementado a cada salto;
- **Type** → tipo do pacote em questão, neste caso, será do tipo *Response*- valor 1;
- **Bit** → campo reservado para informação adicional, no caso do pacote *Response* este contém um valor lógico (0/1) que representa se o pedido ao *Web Server* foi bem efetuado, ou não;
- **Stream ID** → Identificador único da *socket*;
- **Payload Length** → Tamanho do *payload* do pacote, tendo como valor máximo 1020 *bytes*;

- **Payload** → *Payload* que contém o pacote **Response packet from server** (secção 3.2.1) encapsulado.

3. *Talk*:

- **Circuit** → campo preenchido com o caminho que o pacote irá percorrer (direções). Este caminho é definido no pacote **New** e será reutilizado.
- **Hops** → número de saltos (pré-definido) que o pacote deve dar antes de atingir o *client node*. Este campo é decrementado a cada salto;
- **Type** → tipo do pacote em questão, neste caso, será do tipo *Talk*- valor 2;
- **Bit** → campo reservado para informação adicional, no caso do pacote *Talk* este contém um valor lógico (0/1) que representa se o pacote tem como destino o *client node*, e se este for o caso o bit tem o valor 0, ou se tem como destino o *exit node* e tem o valor 1.
- **Stream ID** → Identificador único da *socket*;
- **Payload Length** → Tamanho do *payload* do pacote, tendo como valor máximo 1020 *bytes*;
- **Payload** → *Payload* do pacote que contém os pedidos HTTP ou a resposta aos mesmos, contendo os dados da página *web*.

	CIRCUIT	HOPS	TYPE	BIT	STREAM ID	PAYLOAD LENGTH	PAYLOAD
BIT COUNT	6 bits	2 bits	2 bits	1 bit	8 bits	13 bits	8160 bits

Figura 21: Estrutura dos pacotes *New*, *Response* e *Talk*

4. *End*:

- **Circuit** → Contém o circuito que a ligação utilizou entre o *client node* e o *exit node*, será utilizado o mesmo caminho para o envio deste pacote;
- **Hops** → número de saltos (pré-definido) que o pacote deve dar antes de atingir o *exit node* ou *WebClient*. Este campo é decrementado a cada salto;
- **Type** → tipo do pacote em questão, neste caso, será do tipo *End*- valor 3;
- **Bit** → este campo terá o valor 0, caso este pacote seja enviado do *exit node* para o *WebClient* ou 1, caso seja enviado do *WebClient* para o *exit node*.
- **Stream ID** → Identificador único da *socket*;

	CIRCUIT	HOPS	TYPE	BIT	STREAM ID
BIT COUNT	6 bits	2 bits	2 bits	1 bit	8 bits

Figura 22: Estrutura do pacote *End*

4.2.3. Manutenção da rede P2P

De forma a garantir a robustez da rede de anonimização é necessário que esta se ajuste a possíveis inconvenientes, como é o caso de um dos nós já estabelecidos na rede se desconectar da mesma. Com a saída de um nó, os seus vizinhos vão detetar este acontecimento e contactar o Gestor, via um pacote *Patch Up*, para que assim que possível este "buraco" na rede possa ser preenchido com um novo nó, não alterando assim a estrutura de matriz.

Descrição detalhada do pacote:

1. *Patch up*:

- **Type** → tipo do pacote em questão, neste caso, será do tipo *Patch Up* - valor 3;
- **Direction** → posição do nó vizinho que se desconectou, em relação ao nó que permanece na rede;
- **Client ID** → identificador único do cliente que perdeu um nó vizinho.

	TYPE	DIRECTION	CLIENT ID
BYTE COUNT	1 byte	1 byte	2 bytes

Figura 23: Estrutura do pacote *Patch Up*

4.3. Arquitetura de Software do *Peer*

A arquitetura de *software* do cliente é uma arquitetura *multi-threaded* e está dividida em cinco áreas diferentes, estas são:

1. A área de gestão da rede que comunica com o gestor e outros nós, no *Bootstrap* e na manutenção. Está representada por *ManagerThread*;
2. A área de comunicação com o *Browser* que recebe os pedidos de ligações iniciais e que faz comunicação com o mesmo utilizando o protocolo *SOCKS*. Está representada por *ProxyThread*;
3. A área de encaminhamento, que além do encaminhamento dos pacotes também colabora com todas as restantes áreas. Está representada por *ForwardingThreadPool*;
4. A área de gestão da ligação com o *browser*, que encaminha o tráfego do *browser* e gere a ligação com este. Está representada por *LocalThreadPool*;
5. A área de gestão da ligação com um *WebServer*, que encaminha o tráfego proveniente de um *WebServer* e gere a ligação com este. Está representada por *ExitThreadPool*;

As áreas que possuem na sua nomenclatura *ThreadPool* são áreas onde irá existir mais que uma *Thread* com essa funcionalidade.

De forma a obter o melhor desempenho desta arquitetura, utilizamos *thread pools* para evitar perdas de tempo na criação e destruição de *threads*. Visto que o acesso a um *website* pode, por exemplo, fazer 15 pedidos diferentes, sendo que cada um se irá traduzir numa ligação diferente, se tivéssemos de criar e destruir 15 *threads* por *website* tornava-se numa grande despesa de tempo.

Outra consideração, para um melhor desempenho, foi a utilização de operações binárias, sempre que possível, mas principalmente no protocolo de encaminhamento, porque é neste onde todo o tráfego passa.

De modo a fazer um boa gestão de todas as ligações que são criadas, tanto para o *browser* como para os *WebServers*, cada uma destas tem uma identificação própria, ao que nós designamos de *StreamID*, e estão contidas numa lista. Para não utilizar as mesmas *StreamIDs* na origem e no destino, foram usados mapas para associar um par, caminho e *StreamID*, na respectiva *StreamID* de saída.

De seguida na figura 24, está representado um pequeno diagrama de como estas áreas se interligam, e de forma muito resumida as funcionalidades integrais de cada uma.

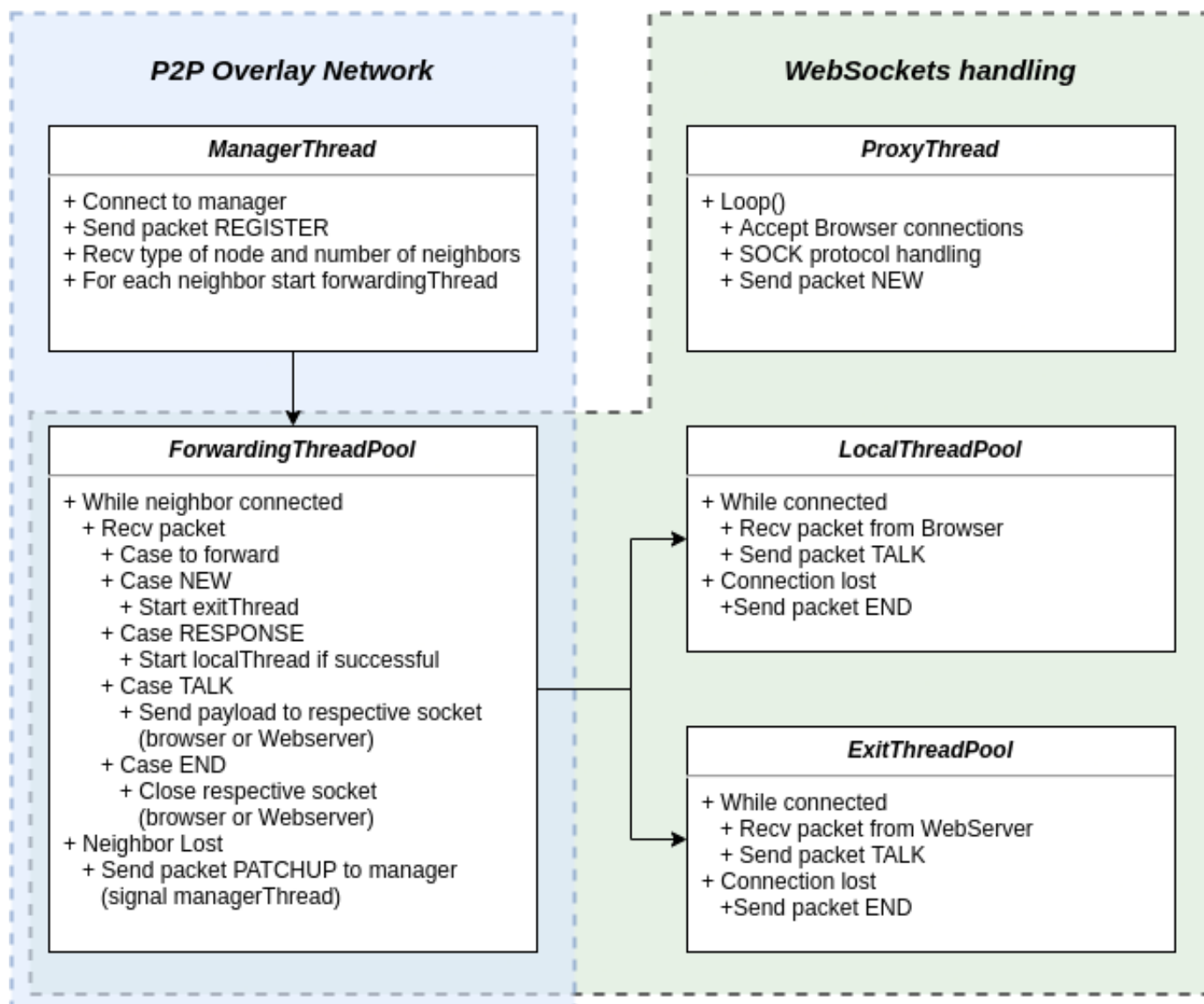


Figura 24: Arquitetura de *Software de Peer*

4.3.1. *ManagerThread*

A *ManagerThread* é responsável por estabelecer a conexão com o gestor e também por estabelecer as conexões com os vizinhos.

Primeiramente, é efetuada uma conexão TCP com o gestor. Após a conexão ser sucedida, é enviado um pacote *Register* para que o nó seja inserido na rede. Depois é aguardada uma resposta por parte do gestor, nomeadamente o pacote *Approval*.

Como já foi referido anteriormente, este pacote irá dizer o tipo de nó que o nó irá ser, que lhe indica o número de vizinhos que irá possuir, e o número de conexões que este irá ter que aguardar que os seus vizinhos efetuem com ele. Caso esse número de conexões que tenha que aguardar seja superior a 0, é chamada uma função para que o nó espere pelo número de ligações indicado no pacote. Senão, simplesmente continua a aguardar por pacotes *Connect To New* (que contêm o endereço IP de um vizinho e a direção dele relativa a esse vizinho) provenientes do gestor para que ele se conecte a um vizinho.

Quando aguarda conexões por parte dos vizinhos, irá esperar por pacotes *Connection Request* que indicam a direção que se encontra o novo vizinho e a chave ECDH pública do mesmo. A isto irá responder com um pacote *Connection Response* com a sua própria chave pública e depois é gerado o segredo partilhado através da curva elíptica de *Diffie-Hellman* e lança uma *thread* da *ForwardingThreadPool* para se responsabilizar pelo tráfego proveniente desse vizinho, indicando a direção a que esse vizinho se encontra.

Após todas as conexões aguardadas estarem efetuadas e caso ainda não possua conexões com todos os vizinhos de acordo com o seu tipo de nó, irá aguardar por pacotes *Connect to New* provenientes do gestor. Ao receber um pacote desse tipo, é efetuado o processo inverso de quando se aguarda por conexões. Ou seja, efetua uma conexão TCP com o endereço IP contido no pacote *Connect to New* e envia um pacote *Connection Request* para esse vizinho e aguarda por um pacote *Connection Response* vindo do mesmo, para que seja gerada o segredo partilhado, nomeadamente a chave de sessão que irá permitir comunicações encriptadas entre os nós. Depois é também lançada uma *thread* da *ForwardingThreadPool* para tratar das comunicações com esse vizinho.

Todos os identificadores das *sockets* que fazem ligação com vizinhos irão ser armazenados num *array* de acesso global, em que a posição de armazenamento será a direção em que o vizinho se encontra.

4.3.2. *ProxyThread*

A *ProxyThread* é responsável pela gestão dos pedidos de conexão do *browser* sobre o protocolo *SOCKS*.

Quando este recebe uma nova ligação, estabelece uma troca de pacotes do protocolo *SOCKS* até receber o pacote com o endereço/domínio do servidor destino. Depois procura-se por uma identificação de *Stream* disponível para associar à *socket* desta ligação e este ultimo pacote *SOCKS* é encapsulado num pacote *New* e é enviado pela rede *Overlay*.

Todo este processo é repetido para cada pedido de conexão que o *browser* tente efetuar.

Na figura 25 apresentamos um fluxograma que representa as operações efetuadas nesta *thread*:

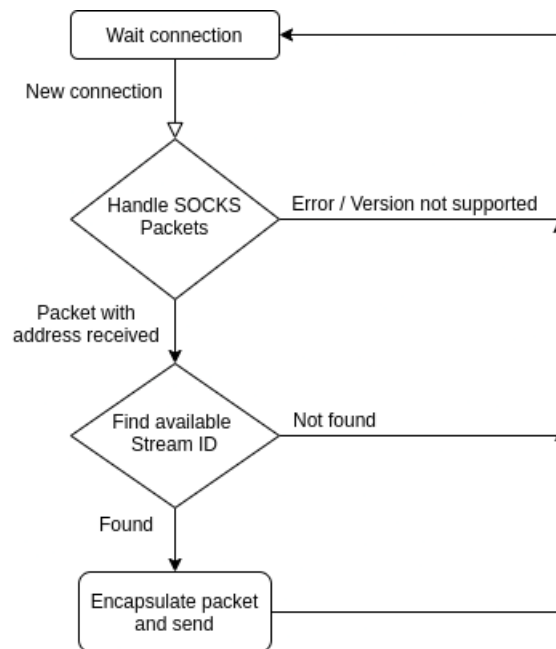


Figura 25: Fluxograma da *ProxyThread*

4.3.3. *ForwardingThreadPool*

Uma *thread* da *ForwardingThreadPool* é responsável principalmente por uma conexão a um vizinho. Apesar do nome que possui, não é apenas efetuado o encaminhamento dos pacotes provenientes do vizinho correspondente para outros vizinhos. O funcionamento desta está descrito de seguida:

1. Quando é recebido um pacote do vizinho é verificado o número de saltos. Caso este seja zero, verifica-se o tipo do pacote e conforme o tipo de pacote existe um procedimento a ser efetuado. Caso o numero de saltos seja diferente de zero, verifica-se o seguinte:
 - Se o pacote é *New*, é selecionada uma nova direção dos vizinhos existentes e é adicionada ao cabeçalho. O número de saltos é decrementado e o pacote é encaminhado para o vizinho da direção selecionada;
 - Caso seja diferente de *New*, lê-se a próxima direção correspondente ao número de salto que se encontra, decrementa-se o número de saltos e encaminha-se o pacote para o vizinho na direção obtida.
2. Caso o numero de saltos seja zero, existem os seguinte procedimentos:
 - Se o pacote é do tipo *New*, tentamos estabelecer ligação com o servidor através do endereço recebido. Caso a conexão seja bem sucedida, procura-se uma nova identificação de *Stream* de saída associada a este nó e se existir, associamos num mapa o par caminho e *StreamID* do pacote a este novo *StreamID* de saída. Depois é enviado um pacote de *Response* com o bit de sucesso a um, e é lançada uma *thread* da *ExitThreadPool*. Caso não seja possível estabelecer ligação ao servidor, ou não haja *StreamID* de saída disponível, é enviado um pacote de *Response* com o bit de sucesso a zero;
 - Quando o pacote é do tipo *Response*, verificamos se houve sucesso na ligação e se sim, responde-se ao *browser* com uma mensagem a indicar que a conexão ao servidor foi bem estabelecida e lança-se uma *thread* da *LocalThreadPool*. O caminho contido no pacote é também armazenado para ser utilizado nos pedidos seguintes associados à *StreamID* correspondente. Caso contrário, a respectiva ligação ao *browser* é terminada e a *StreamID* é desassociada dessa ligação ;
 - Sendo o pacote recebido do tipo *Talk*, verifica-se se tem como destino um nó de origem (*Local*) ou um nó de saída (*Exit*). Se for um nó de origem, o conteúdo do *payload* será uma resposta a um pedido efetuado anteriormente, e através do *StreamID* contido no

pacote obtemos a socket que representa esta ligação ao *browser* e envia-se o conteúdo do *payload*. Se o destino é *Exit*, através do par caminho e *StreamID* que vêm no cabeçalho do pacote obtemos a *StreamID* de saída associada a esse par no nó de saída. Através da *StreamID* de saída obtemos a *socket* que faz a ligação ao servidor e envia-se o conteúdo do *payload*, que é um pedido HTTP/HTTPS;

- Caso o pacote seja *End*, conforme o destino, *Local* ou *Exit*, e através da *StreamID* respectiva, é terminada a determinada ligação e é limpo o par caminho e *StreamID*.

3. Caso o vizinho se desconecte, é terminada a conexão com ele, elimina-se o registo dele e é enviado um pacote de *Patch Up* para o gestor a avisar a perda do vizinho;

De forma a exemplificar a descrição anterior, foram feitos um conjunto de fluxogramas, presentes nas seguintes figuras:

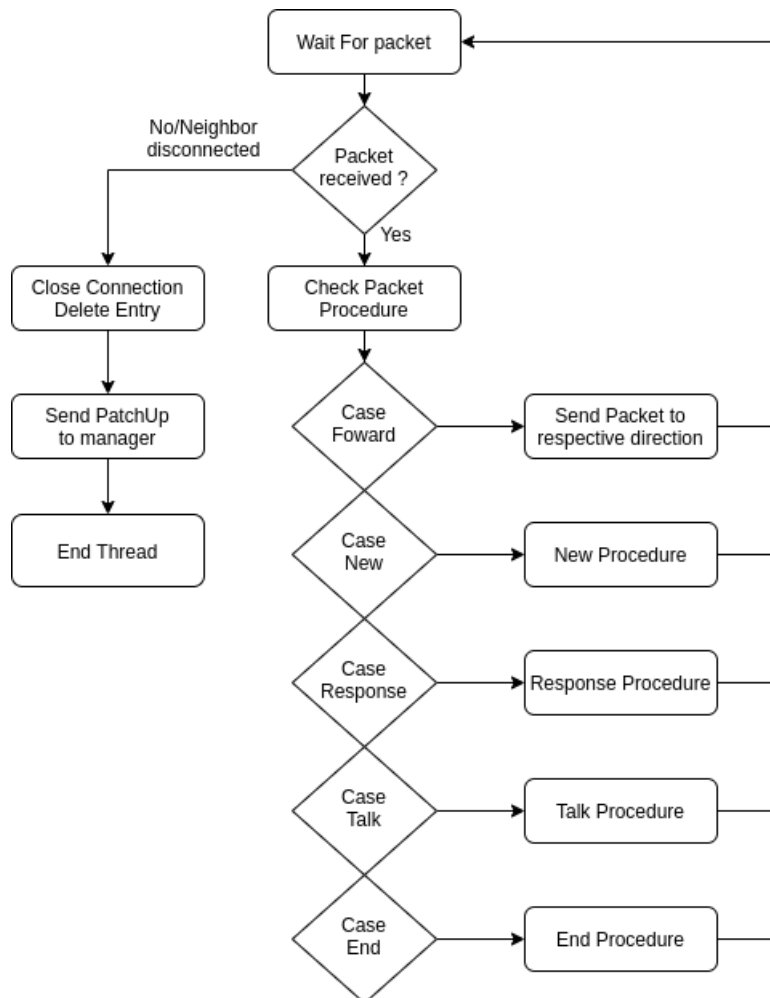


Figura 26: Fluxograma geral da *ForwardingThread*

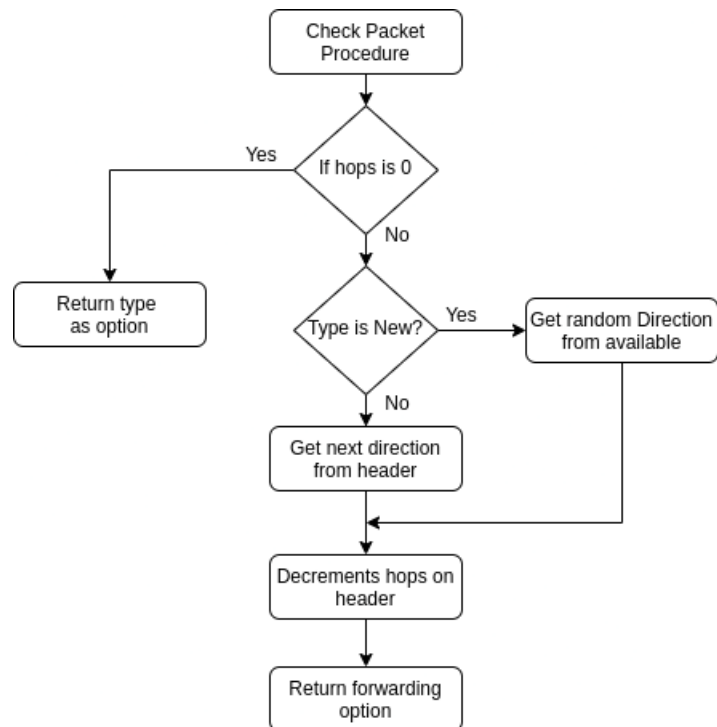
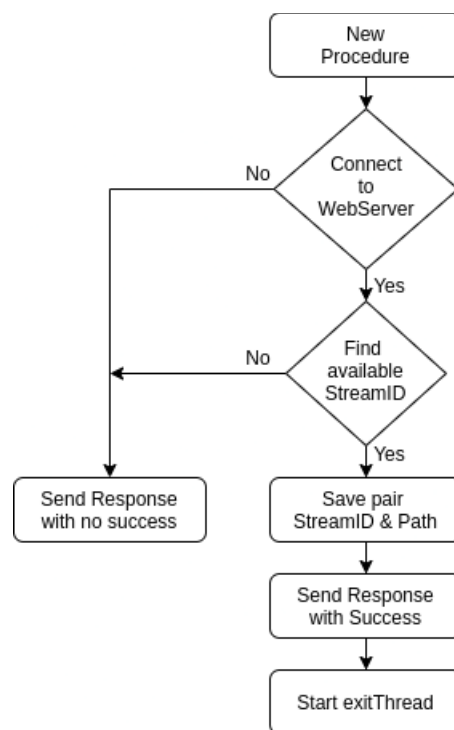
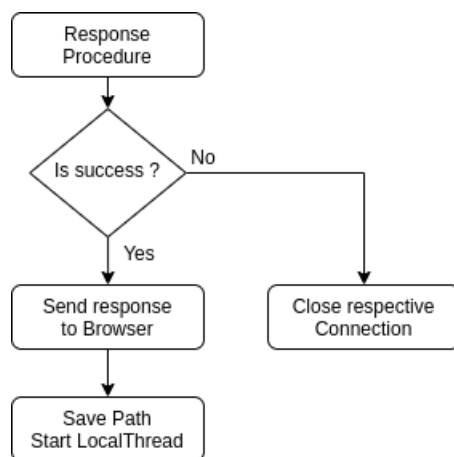
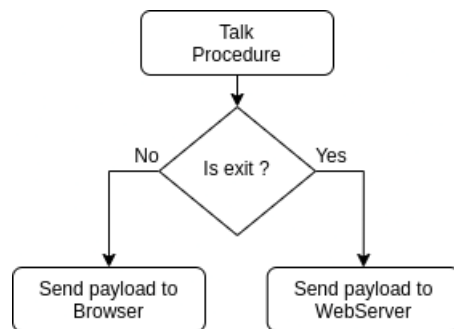
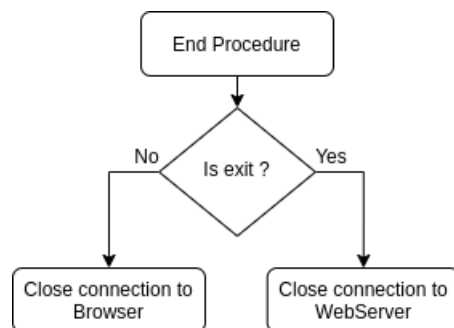


Figura 27: Fluxograma de verificação do cabeçalho do pacote

Figura 28: Fluxograma do procedimento para o tipo *New*

Figura 29: Fluxograma do procedimento para o tipo *Response*Figura 30: Fluxograma do procedimento para o tipo *Talk*Figura 31: Fluxograma do procedimento para o tipo *End*

4.3.4. *LocalThreadPool*

As *threads* deste tipo são responsáveis por receber os pedidos provenientes de uma ligação com o *browser* e encapsulá-los em pacotes do tipo *Talk* com destino ao nó de saída.

Ao serem lançadas recebem como argumento a identificação da *Stream* que identifica a respetiva ligação ao *browser*. Através desta é obtido o caminho a ser colocado nos cabeçalhos dos pacotes *Talk* gerados por esta ligação.

Quando o *browser* fecha a conexão, é gerado um pacote *End* para ser encaminhado até ao nó de saída para que este feche a ligação com o servidor e esta *thread* volta para a *LocalThreadPool*

4.3.5. *ExitThreadPool*

As *threads* deste tipo são responsáveis por receber as respostas provenientes de uma ligação ao servidor e encapsulá-los em pacotes do do tipo *Talk* com destino ao nó de origem.

Estas quando são lançadas recebem por forma de argumento a identificação da *Stream* de saída de que é responsável. Através desta consegue obter o par caminho e *StreamID* de origem, que vai usar para a construção dos pacotes *Talk*.

Quando a ligação termina, a *thread* em questão volta para a *ExitThreadPool*.

4.4. Arquitetura de *Software* do Gestor

A arquitetura de *software* do gestor, tal como a do cliente, é uma arquitetura *multi-threaded* e está dividida em duas áreas, que são as seguintes:

1. A área de gestão da rede em matriz. Está representada por *MainThread*;
2. A área de gestão da ligação com o cliente. Está representada por *ClientThreadPool*;

A figura 32 demonstra um diagrama que representa a arquitetura do gestor, com um resumo das funcionalidades de cada área.

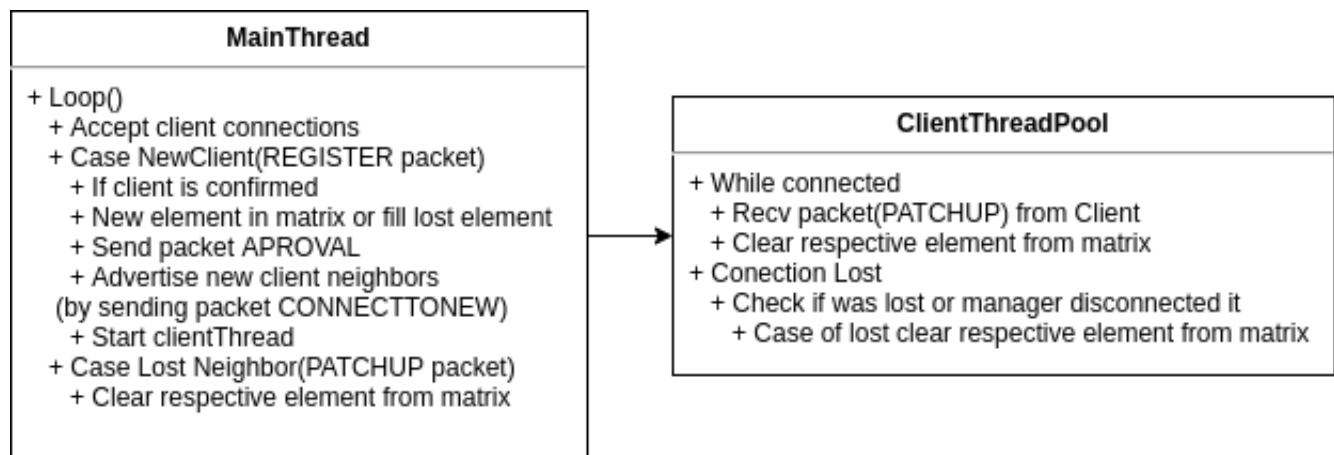


Figura 32: Diagrama da arquitetura do gestor

4.4.1. *MainThread*

A *MainThread* é a responsável pela ordenação de todos os nós pela matriz e da gestão da mesma.

A gestão da matriz é feita através de um mapa onde as coordenadas são a chave e a identificação do nó o valor. Desta forma não nos limitamos a uma estrutura de dados fixa. Para as coordenadas usamos uma estrutura com métodos para obter as coordenadas em torno de uma dada coordenada.

Depois para que esta matriz cresça da forma que determinamos, desenvolvemos o seguinte algoritmo:

```
\\valores iniciais
lado = 1;
coordenada_atual = (0,0);

ProximaCoordenada(){
    if (numero_elementos == lado*lado) {
        lado++;
        coordenada_atual.x = lado - 1;
        coordenada_atual.y = 0;
    } else {
        if (coordenada_atual.x == lado-1) {
            if (coordenada_atual.y == lado-2)
                coordenada_atual.x = 0;
            coordenada_atual.y++;
        } else {
            coordenada_atual.x++;
        }
    }
}
```

Conforme o dado algoritmo, conseguimos fazer com que a matriz cresça sempre com uma coluna à direita, quando esta enche é iniciada uma linha abaixo e por fim aumenta o lado e começa de novo uma coluna à direita.

De modo a que a matriz esteja sem "buracos", desenvolvemos uma *PriorityQueue* que coloca os nós perdidos numa *Queue* ordenada de acordo com o número de vizinhos que os nós possuíam. Se existir mais do que um nó a desconectar-se da rede, a posição do nó que possuía mais vizinhos é mais prioritária, pois é a que garante mais conectividade à rede.

Quando o gestor recebe uma ligação TCP e um pacote do tipo *Register*, é inicializado o procedimento de adicionar um novo elemento à matriz e é lançada uma *thread* da *ClientThreadPool*. Este começa por

verificar se a *PriorityQueue* possui alguma entrada e se sim o novo elemento toma a posição dessa entrada. Senão o novo elemento toma uma nova posição na matriz.

Neste procedimento de tomar uma posição da matriz, o primeiro passo foi verificar a existência de vizinhos da determinada posição, depois obter o tipo de nó a partir da posição dele e enviar um pacote do tipo *Approval*. Por fim, são enviados pacotes do tipo *Connect to New* para os vizinhos desse novo elemento. Se o gestor verificar que um elemento já possui conexões com todos os possíveis vizinhos, este termina a ligação com esse elemento.

Quando o gestor recebe um pacote do tipo *Patch Up*, este através da identificação do nó e direção e obtém a identificação do elemento que saiu, depois verifica se este já foi removido. Se sim significa que já foi reportado, se não vai ser adicionado à *PriorityQueue*.

4.4.2. *ClientThreadPool*

A *ClientThreadPool* tem duas funcionalidades: receber pacotes de manutenção dos nós que ainda estão com ligações estabelecidas com o gestor e verificar se um nó se desconectou da rede de forma anormal. Por exemplo, se um nó estiver sem vizinhos e este se desconecte, não haveria reportagem da sua saída e assim, existindo uma *thread* que monitorize essa conexão, o gestor consegue obter essa informação.

5. Testes, Resultados e Discussão

5.1. Requisitos para uso da rede

Para executar um gestor de rede, ou um *WebClient* é necessário utilizar um sistema operativo que utilize *kernel* linux.

Um requisito necessário para ser cliente da rede, é ter instalada a biblioteca *libssl*. Esta é necessária para o uso das funções criptográficas implementadas [3.3]. A instalação desta deverá ser simples utilizando um *package manager* do sistema (ex: apt, pacman, yum, ...).

Para evitar que o pacote que atravessa a rede seja encaminhado mais do que uma vez por um cliente, nunca permitimos que seja enviado para o nó de onde veio. Esta simples implementação provoca duas limitações:

1. Para um cliente dar uso à rede, tem de haver no mínimo quatro clientes.

Como podemos verificar na figura abaixo, numa situação em que existam três nós na rede, se houver um pedido vindo do cliente inferior, como o protocolo está definido para haver três saltos na rede, este irá ser encaminhado para o cliente superior esquerdo, que irá encaminhar para o cliente superior direito, como ainda não foram completados os três saltos, e como não permitimos que um pacote seja enviado para o local de onde veio, não vai ser possível encaminhar o pacote, e esta tentativa de conexão à web não vai ter sucesso.

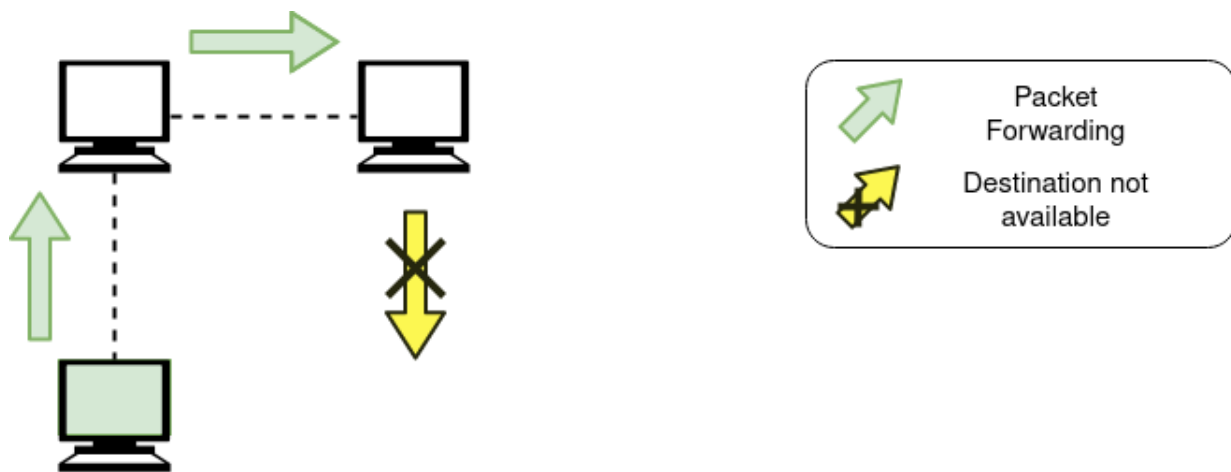


Figura 33: Erro caso não existam 4 clientes

2. Para um cliente encaminhar tráfego, tem de estar conectado a dois vizinhos.

Se um cliente não possuir pelo menos duas ligações a nós da rede, embora este possa estar possibilitado a iniciar uma ligação com o seu vizinho e a encaminhar pedidos que possam ser bem sucedidos, este pode não ser capaz de responder a pedidos que lhe cheguem de outro cliente. Por exemplo, na imagem a seguir, o nó da direita que apenas está conectado a um vizinho, não consegue encaminhar um pacote que seja proveniente do seu único vizinho, e que ainda tenha de dar algum salto na rede, pois não é permitido que este reencaminhe o pacote para o local de onde ele veio.

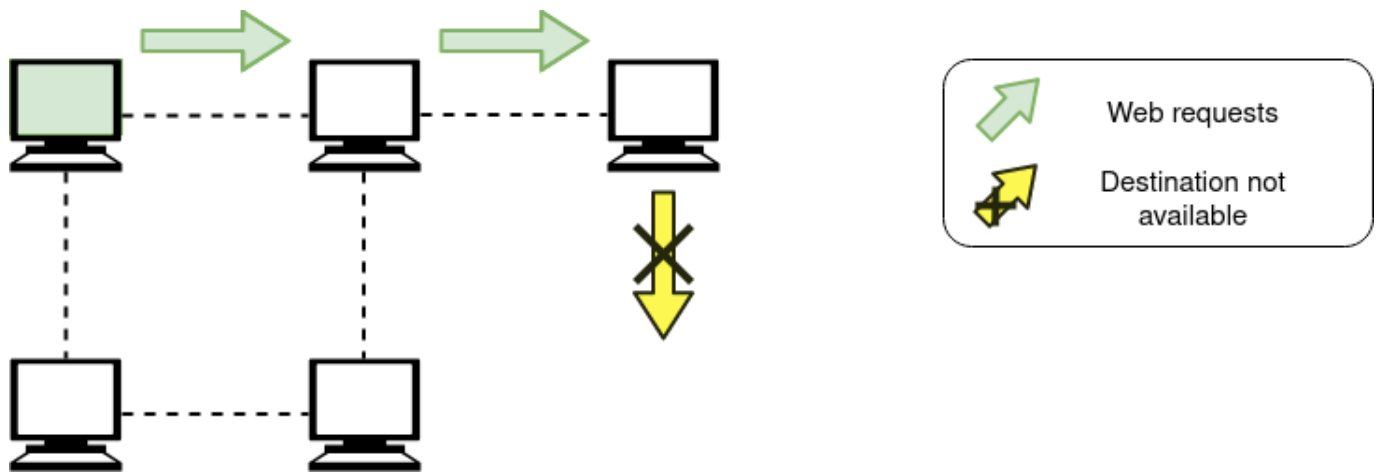


Figura 34: Erro caso não existam 2 clientes

5.2. Testes e Resultados

De forma a testarmos todo o sistema desenvolvido, decidimos utilizar uma ferramenta de simulação de rede, nomeadamente o GNS3.

A topologia de rede que nós criámos é constituída por 6 computadores com o sistema Ubuntu, sendo 4 utilizados como nós de rede, um como o gestor da rede e um outro que irá funcionar como um servidor *Web*.

Para interligar todos estes componentes, é utilizado um *switch*, que por sua vez está ligado a um componente NAT que permite a conexão à Internet.

A figura 35 apresenta a topologia que acabamos de descrever.

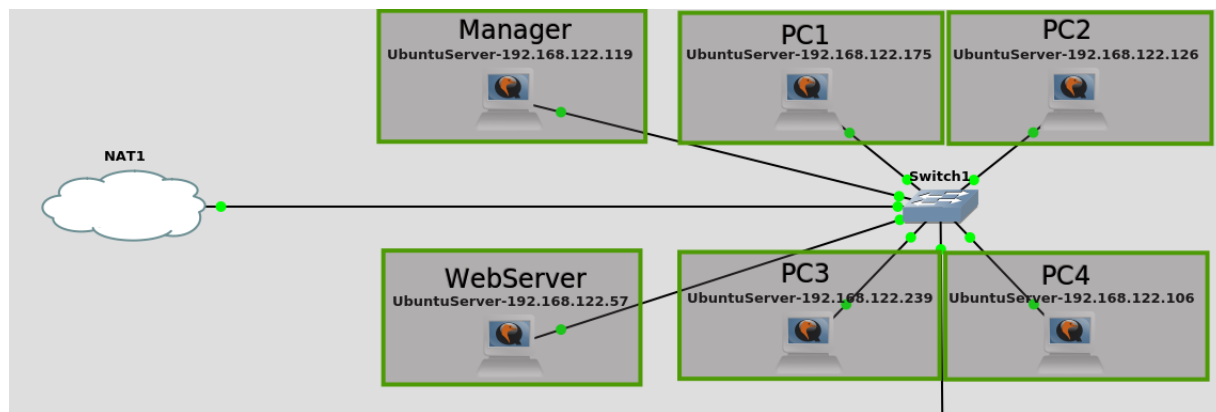


Figura 35: Topologia de rede utilizada

Após todos os elementos estarem devidamente conectados à rede de anonimização, através do computador com o endereço IP 192.168.122.175 acedemos, com sucesso, ao *website* *www.elearning.uminho.pt*, como podemos observar na figura 36.

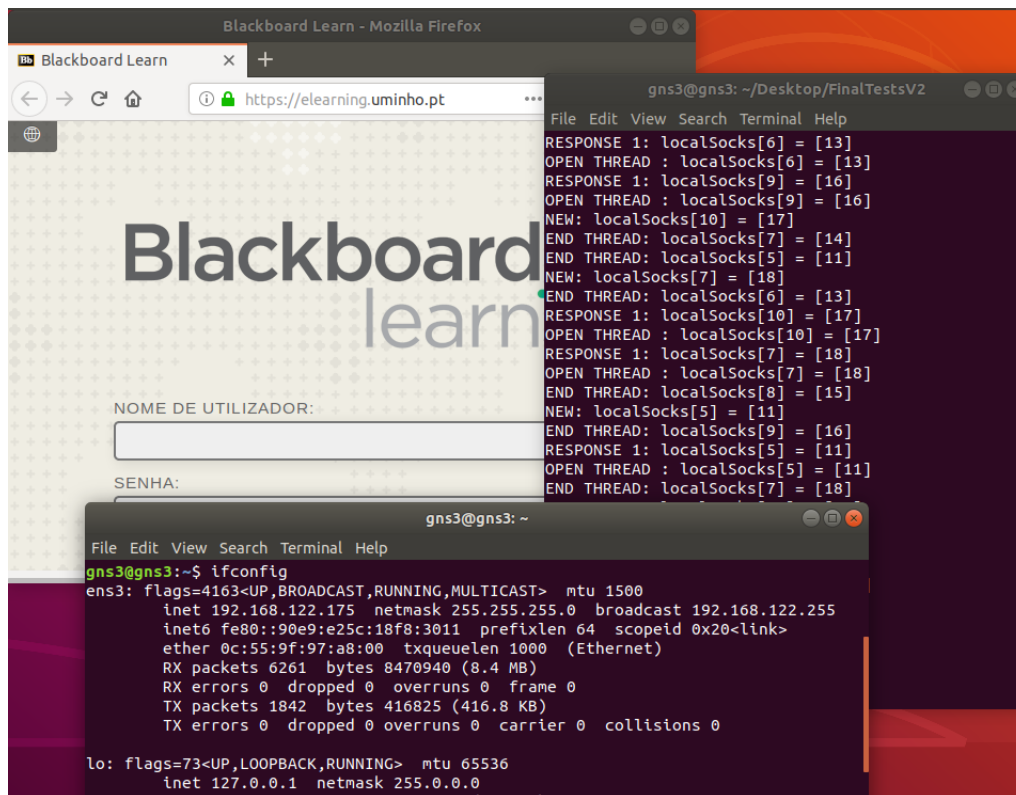


Figura 36: Sucesso ao aceder ao *e-learning*

Ao mesmo tempo que estávamos a efetuar o acesso ao *e-learning*, iniciamos uma captura de tráfego na ligação com o NAT.

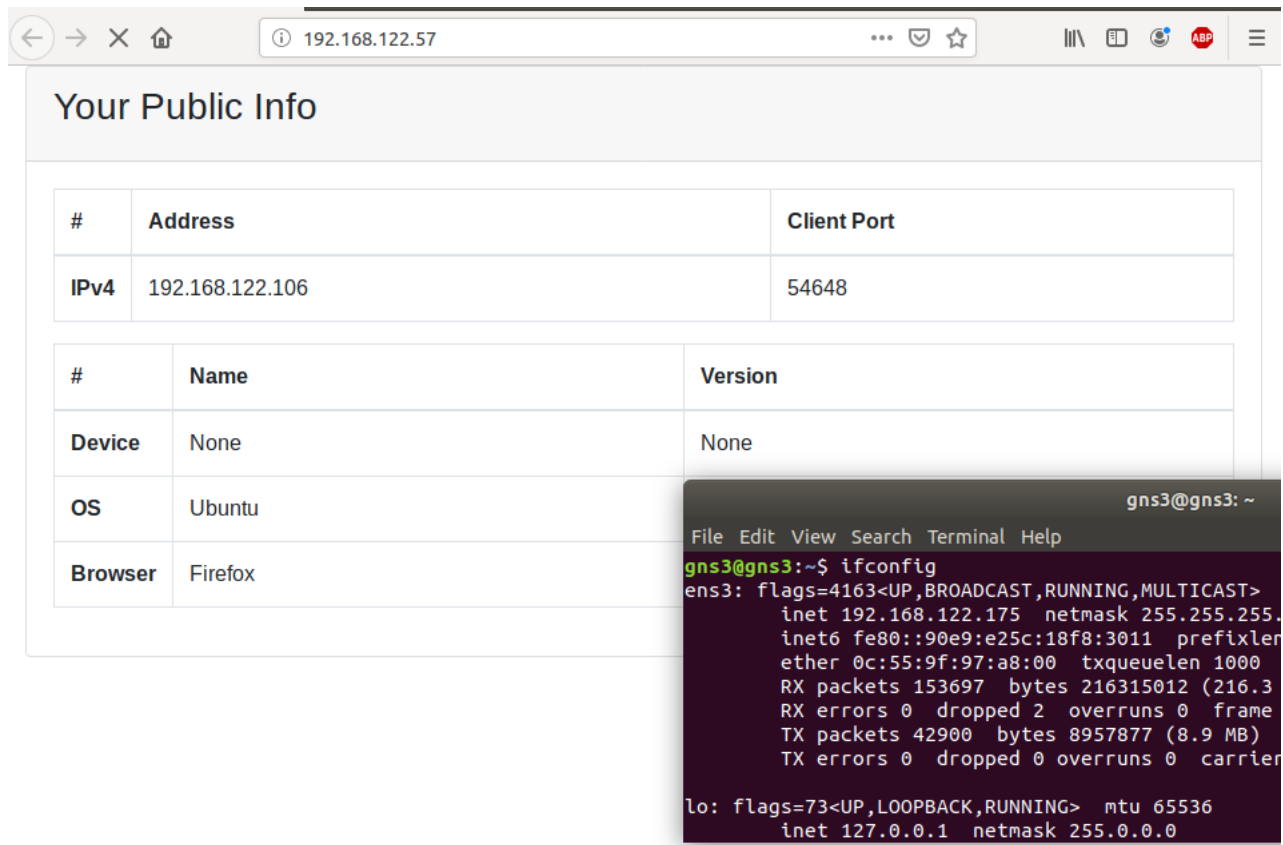
Como se pode averiguar na captura presente na figura 37, o endereço IP que acede ao *e-learning* não corresponde aos endereços que efetuam os pedidos (192.168.122.126 e 192.168.122.106), demonstrando assim que os pedidos estão a ser efetuados por outros elementos da rede.

895	39.269133	192.168.122.106	193.137.9.150	TCP	66 60706 → 443 [ACK]	Seq=1192 Ack=4486 W
896	39.269174	193.137.9.150	192.168.122.106	TCP	1514 443 → 60706 [ACK]	Seq=4486 Ack=1192 W
897	39.269323	192.168.122.106	193.137.9.150	TCP	66 60706 → 443 [ACK]	Seq=1192 Ack=5934 W
898	39.273018	193.137.9.150	192.168.122.106	TCP	1514 443 → 60706 [ACK]	Seq=5934 Ack=1192 W
899	39.273057	193.137.9.150	192.168.122.106	TCP	1514 443 → 60706 [ACK]	Seq=7382 Ack=1192 W
900	39.273064	193.137.9.150	192.168.122.106	TCP	1514 443 → 60706 [ACK]	Seq=8830 Ack=1192 W
901	39.273071	193.137.9.150	192.168.122.106	TCP	1514 443 → 60706 [ACK]	Seq=10278 Ack=1192 W
902	39.273323	192.168.122.106	193.137.9.150	TCP	66 60706 → 443 [ACK]	Seq=1192 Ack=11726 W
903	39.277184	193.137.9.150	192.168.122.106	TCP	1514 443 → 60706 [ACK]	Seq=11726 Ack=1192 W
904	39.277249	193.137.9.150	192.168.122.106	TCP	1514 443 → 60706 [ACK]	Seq=13174 Ack=1192 W
905	39.277505	192.168.122.106	193.137.9.150	TCP	66 60706 → 443 [ACK]	Seq=1192 Ack=13174 W
906	39.277558	192.168.122.106	193.137.9.150	TCP	66 60706 → 443 [ACK]	Seq=1192 Ack=14622 W
907	39.290835	193.137.9.150	192.168.122.106	TCP	1514 443 → 60706 [ACK]	Seq=14622 Ack=1192 W
908	39.290875	193.137.9.150	192.168.122.106	TLSv1.2	1514 Application Data	Application Data
909	39.290883	193.137.9.150	192.168.122.106	TLSv1.2	238 Application Data	
910	39.291075	192.168.122.106	193.137.9.150	TCP	66 60706 → 443 [ACK]	Seq=1192 Ack=17691 W
911	39.312473	192.168.122.106	193.137.9.150	TLSv1.2	97 Encrypted Alert	
912	39.324848	192.168.122.126	192.168.122.1	DNS	90 Standard query 0xa3cc AAAA elearning.	
913	39.328909	193.137.9.150	192.168.122.106	TCP	54 443 → 60706 [RST, ACK]	Seq=17691 Ack=17691 W
914	39.336693	192.168.122.1	192.168.122.126	DNS	90 Standard query response 0xa3cc AAAA elearning.	
915	39.337209	192.168.122.126	193.137.9.150	TCP	74 49256 → 443 [SYN]	Seq=0 Win=29200 Len=0 W
916	39.353546	193.137.9.150	192.168.122.126	TCP	74 443 → 49256 [SYN, ACK]	Seq=0 Ack=1 Win=0 W
917	39.353816	192.168.122.126	193.137.9.150	TCP	66 49256 → 443 [ACK]	Seq=1 Ack=1 Win=293 W
918	39.399315	192.168.122.126	193.137.9.150	TLSv1.2	583 Client Hello	
919	39.427191	193.137.9.150	192.168.122.126	TLSv1.2	156 Server Hello	
920	39.427230	193.137.9.150	192.168.122.126	TLSv1.2	72 Change Cipher Spec	
921	39.427639	192.168.122.126	193.137.9.150	TCP	66 49256 → 443 [ACK]	Seq=518 Ack=91 Win=0 W
922	39.427677	192.168.122.126	193.137.9.150	TCP	66 49256 → 443 [ACK]	Seq=518 Ack=97 Win=0 W
923	39.427703	193.137.9.150	192.168.122.126	TLSv1.2	111 Encrypted Handshake Message	
924	39.427927	192.168.122.126	193.137.9.150	TCP	66 49256 → 443 [ACK]	Seq=518 Ack=142 Win=0 W
925	39.474162	192.168.122.126	193.137.9.150	TLSv1.2	626 Change Cipher Spec, Encrypted Handshake Message	
926	39.494500	193.137.9.150	192.168.122.126	TCP	1514 443 → 49256 [ACK]	Seq=142 Ack=1078 Win=0 W
927	39.494539	193.137.9.150	192.168.122.126	TCP	1514 443 → 49256 [ACK]	Seq=1590 Ack=1078 Win=0 W
928	39.494892	192.168.122.126	193.137.9.150	TCP	66 49256 → 443 [ACK]	Seq=1078 Ack=3038 W
929	39.499687	193.137.9.150	192.168.122.126	TCP	1514 443 → 49256 [ACK]	Seq=3038 Ack=1078 W
930	39.499729	193.137.9.150	192.168.122.126	TCP	1514 443 → 49256 [ACK]	Seq=4486 Ack=1078 W
931	39.499746	193.137.9.150	192.168.122.126	TCP	1514 443 → 49256 [ACK]	Seq=5934 Ack=1078 W
932	39.499751	193.137.9.150	192.168.122.126	TCP	1514 443 → 49256 [ACK]	Seq=7382 Ack=1078 W
933	39.499756	193.137.9.150	192.168.122.126	TLSv1.2	140 Application Data	
934	39.499763	193.137.9.150	192.168.122.126	TCP	66 443 → 49256 [FIN, ACK]	Seq=8904 Ack=1078 W
935	39.499938	192.168.122.126	193.137.9.150	TCP	66 49256 → 443 [ACK]	Seq=1078 Ack=8904 W
936	39.541277	192.168.122.126	193.137.9.150	TLSv1.2	97 Encrypted Alert	

Figura 37: Captura de tráfego Wireshark

Para demonstrar que o endereço que efetua os pedidos *web* não é o cliente, desenvolvemos um simples servidor *web* em *Python* que retorna informações sobre o nó a que está conectado, nomeadamente, o seu endereço IP e porta, o nome do dispositivo, o sistema operativo e o *browser* que foi utilizado.

Nas seguintes imagens 38 e 39 comparamos as informações do nó que efetuou o pedido *web* (*exit node*) e as informações do cliente (*client node*). A cada "refresh" do *webserver* o *exit node* poderá ser diferente, demonstrando assim a aleatoriedade dos caminhos de encaminhamento.



The image shows a web browser window with the address bar displaying '192.168.122.57'. The page title is 'Your Public Info'. Below the title, there are two tables. The first table shows public information, and the second table shows system information. A terminal window is overlaid on the bottom right, showing the output of the 'ifconfig' command.

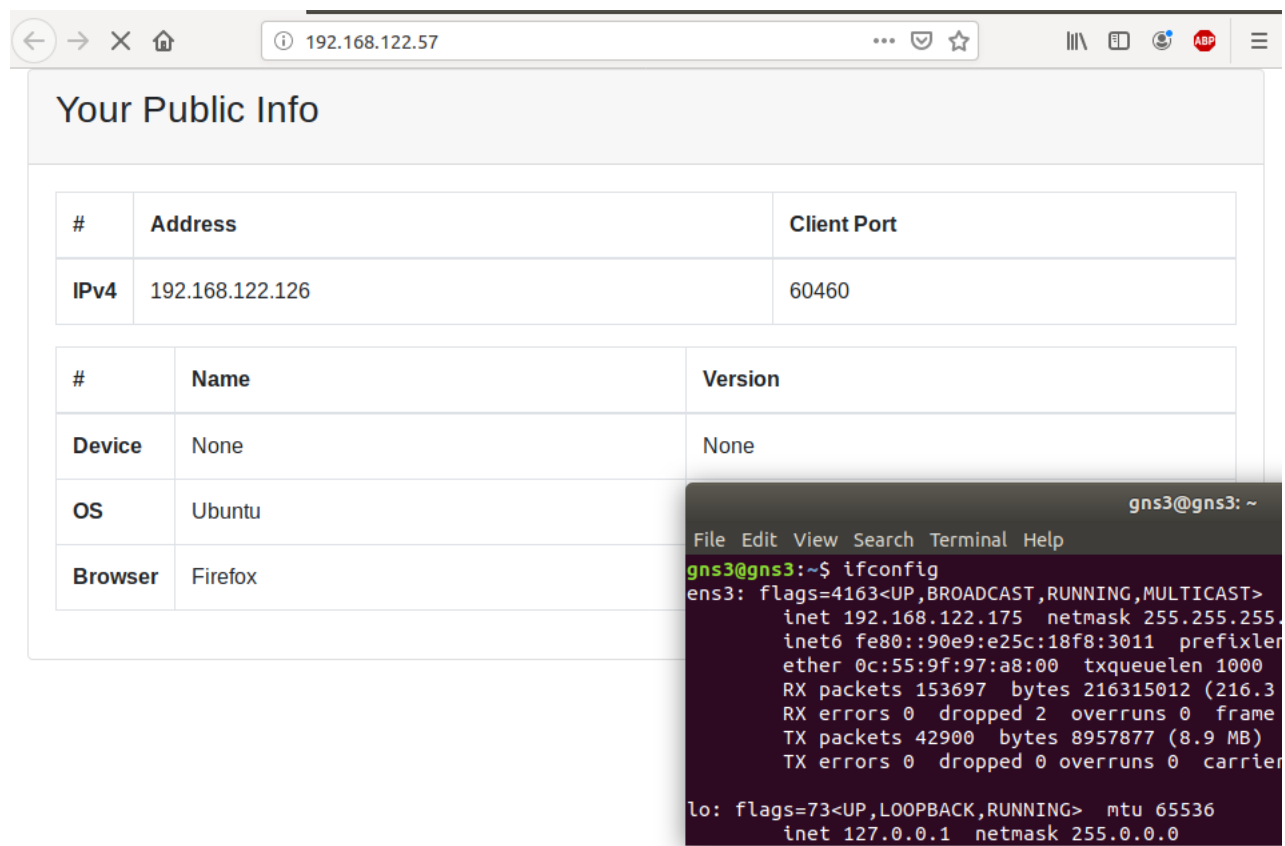
#	Address	Client Port
IPv4	192.168.122.106	54648

#	Name	Version
Device	None	None
OS	Ubuntu	
Browser	Firefox	


```

gns3@gns3: ~
File Edit View Search Terminal Help
gns3@gns3:~$ ifconfig
ens3: flags=4163<UP,BROADCAST,RUNNING,MULTICAST>
    inet 192.168.122.175 netmask 255.255.255.0
    inet6 fe80::90e9:e25c:18f8:3011 prefixlen 64
    ether 0c:55:9f:97:a8:00 txqueuelen 1000
    RX packets 153697 bytes 216315012 (216.3 MB)
    RX errors 0 dropped 2 overruns 0 frame
    TX packets 42900 bytes 8957877 (8.9 MB)
    TX errors 0 dropped 0 overruns 0 carrier
    device
    lo: flags=73<UP,LOOPBACK,RUNNING> mtu 65536
    inet 127.0.0.1 netmask 255.0.0.0
  
```

Figura 38: Informações *exit node* / informações *client node*



The image shows a web browser window with the address bar set to 192.168.122.57. The page title is "Your Public Info". Below the title, there are two tables. The first table has columns for "#", "Address", and "Client Port". The second table has columns for "#", "Name", and "Version".

#	Address	Client Port
IPv4	192.168.122.126	60460

#	Name	Version
Device	None	None
OS	Ubuntu	
Browser	Firefox	

Overlaid on the bottom right of the browser window is a terminal window titled "gns3@gns3: ~". The terminal shows the output of the "ifconfig" command:

```
gns3@gns3:~$ ifconfig
ens3: flags=4163<UP,BROADCAST,RUNNING,MULTICAST>
    inet 192.168.122.175 netmask 255.255.255.0
    inet6 fe80::90e9:e25c:18f8:3011 prefixlen 64
    ether 0c:55:9f:97:a8:00 txqueuelen 1000
    RX packets 153697 bytes 216315012 (216.3 MB)
    RX errors 0 dropped 2 overruns 0 frame
    TX packets 42900 bytes 8957877 (8.9 MB)
    TX errors 0 dropped 0 overruns 0 carrier
    device not promiscuous
lo: flags=73<UP,LOOPBACK,RUNNING> mtu 65536
    inet 127.0.0.1 netmask 255.0.0.0
```

Figura 39: Informações *exit node* / informações *client node*

Para demonstrar o *Bootstrap* nesta topologia, inicializamos o gestor e quatro *peers*.

Podemos verificar na figura 40, no terminal de baixo, que corresponde ao gestor, cada par de linhas representa um novo nó na rede (identificação do *peer* e número de vizinhos).

Nos restantes terminais, correspondentes aos *peers*, podemos verificar a cada ligação a um vizinho:

1. `neighbor[0] = 0` → não possui vizinho acima;
2. `neighbor[1] = 1` → possui vizinho à direita;
3. `neighbor[2] = 0` → não possui vizinho abaixo;
4. `neighbor[3] = 0` → não possui vizinho à esquerda;
5. `sharedKey` → chave partilhada com o respectivo vizinho.

```

sr-d1r3171nh0@pop-os: ~/Documents/4ºAno/PTI1/Overlay-Network...
nt
f5bbb4672b0fa61273348f10faa358545aa2e2fb95694c13b562335ac8823d3c
neighbor[0] = 0
neighbor[1] = 1
neighbor[2] = 0
neighbor[3] = 0
b2211ee2ebc4a0e97b687487c7fe3ef2cbcf6466a0ee22f8af6695534809e9c9
neighbor[0] = 0
neighbor[1] = 1
neighbor[2] = 1
neighbor[3] = 0

UbuntuServer-192.168.122.126
gns3@gns3:~/Desktop/FinalTestsV2$ ./client 2
neighbor[0] = 0
neighbor[1] = 0
neighbor[2] = 0
neighbor[3] = 1
f5bbb4672b0fa61273348f10faa358545aa2e2fb95694c13b562335ac8823d3c
1d811b47af10fd7427291faa5b3e15ed6fa106959baf5523b3ad5bc1c655156d
neighbor[0] = 0
neighbor[1] = 0
neighbor[2] = 1
neighbor[3] = 1

UbuntuServer-192.168.122.106
gns3@gns3:~/Desktop/FinalTestsV2$ ./client 3
neighbor[0] = 1
neighbor[1] = 0
neighbor[2] = 0
neighbor[3] = 0
b2211ee2ebc4a0e97b687487c7fe3ef2cbcf6466a0ee22f8af6695534809e9c9
4e90cc19d11051aac5c90ca3fb283deae08f33db3d0c6134f9bb99b46dee7a27
neighbor[0] = 1
neighbor[1] = 1
neighbor[2] = 0
neighbor[3] = 0

UbuntuServer-192.168.122.239
gns3@gns3:~/Desktop/FinalTestsV2$ ./client 4
neighbor[0] = 1
neighbor[1] = 0
neighbor[2] = 0
neighbor[3] = 0
1d811b47af10fd7427291faa5b3e15ed6fa106959baf5523b3ad5bc1c655156d
neighbor[0] = 1
neighbor[1] = 0
neighbor[2] = 0
neighbor[3] = 1
4e90cc19d11051aac5c90ca3fb283deae08f33db3d0c6134f9bb99b46dee7a27

UbuntuServer-192.168.122.119
gns3@gns3:~/Desktop/FinalTestsV2$ ./manager
New place 32900
Numero de vizinhos 0
New place 14027
Numero de vizinhos 1
New place 31434
Numero de vizinhos 1
New place 47176
Numero de vizinhos 2

```

Figura 40: Entrada dos nós na rede de anonimização

Para demonstrar os avisos da saída de cada nó (pacotes de *Patch Up*), forçamos a saída de dois nós da rede anterior.

Na figura 41, no terminal de baixo correspondente ao gestor, podemos verificar as seguinte linhas:

1. `Lost neighbor 2` → um nó avisou que perdeu o vizinho de baixo;
2. `Reporting node 31434` → o gestor conclui que o *peer* que saiu foi o 31434;
3. `Node 31434 lost` → o registo foi apagado;
4. `Lost neighbor 3` → outro nó avisou que perdeu o vizinho da esquerda;
5. `Reporting node 0` → o gestor conclui que o *peer* que saiu já foi reportado;

The figure displays three terminal windows from the UbuntuServer-192.168.122.126 environment, showing the execution of a network simulation. The top-left window shows the output of the 'client 2' command, displaying neighbor status and error messages. The top-right window shows the output of the 'client 4' command, also displaying neighbor status and error messages. The bottom window shows the output of the 'manager' command, displaying the current state of the network, including the number of neighbors and the status of various nodes.

```

sr-d1r3171nh0@pop-os: ~/Documents/4ºAno/PTI1/Overlay-Network/FinalTestsV2$ ./client 2
nt
f5bbb4672b0fa61273348f10faa358545aa2e2fb95694c13b562335ac8823d3c
neighbor[0] = 0
neighbor[1] = 1
neighbor[2] = 0
neighbor[3] = 0
b2211ee2ebc4a0e97b687487c7fe3ef2cbcf6466a0ee22f8af6695534809e9c9
neighbor[0] = 0
neighbor[1] = 1
neighbor[2] = 1
neighbor[3] = 0
-----ATENCAO: n = 0
ERRO: neighbor[2] = [7]
-----ATENCAO: n = 0
ERRO: neighbor[1] = [6]
[]

UbuntuServer-192.168.122.106
gns3@gns3:~/Desktop/FinalTestsV2$ ./client 3
neighbor[0] = 1
neighbor[1] = 0
neighbor[2] = 0
neighbor[3] = 0
b2211ee2ebc4a0e97b687487c7fe3ef2cbcf6466a0ee22f8af6695534809e9c9
4e90cc19d11051aac5c90ca3fb283deae08f33db3d0c6134f9bb99b46dee7a27
neighbor[0] = 1
neighbor[1] = 1
neighbor[2] = 0
neighbor[3] = 0
^C
gns3@gns3:~/Desktop/FinalTestsV2$ []

UbuntuServer-192.168.122.239
gns3@gns3:~/Desktop/FinalTestsV2$ ./client 4
neighbor[0] = 1
neighbor[1] = 0
neighbor[2] = 0
neighbor[3] = 0
1d811b47af10fd7427291faa5b3e15ed6fa106959baf5523b3ad5bc1c655156d
neighbor[0] = 1
neighbor[1] = 0
neighbor[2] = 0
neighbor[3] = 1
4e90cc19d11051aac5c90ca3fb283deae08f33db3d0c6134f9bb99b46dee7a27
-----ATENCAO: n = 0
ERRO: neighbor[3] = [10]
-----ATENCAO: n = 0
ERRO: neighbor[0] = [5]
[]

UbuntuServer-192.168.122.119
gns3@gns3:~/Desktop/FinalTestsV2$ ./manager
New place 32900
Numero de vizinhos 0
New place 14027
Numero de vizinhos 1
New place 31434
Numero de vizinhos 1
New place 47176
Numero de vizinhos 2
Lost neighbor 2
Reporting node: 31434
Node 31434 lost
Lost neighbor 3
Reporting node: 0
Lost neighbor 1
Reporting node: 14027
Node 14027 lost
Lost neighbor 0
Reporting node: 0

```

Figura 41: Saída de nós da rede

Para demonstrar a reinserção de *peers* na rede voltamos a colocar os mesmos nós, o processo descrito no *bootstrap* repete-se. Se verificarmos com atenção a figura 42, podemos verificar que os que saíram mudaram de posição ($US: 192.168.122.106 \leftrightarrow US: 192.168.122.126$)

```

sr-d1r3171nh0@pop-os: ~/Documents/4ºAno/PTI1/Overlay-Networ...
nt
f5bbb4672b0fa61273348f10faa358545aa2e2fb95694c13b562335ac8823d3c
neighbor[0] = 0
neighbor[1] = 1
neighbor[2] = 0
neighbor[3] = 0
b2211ee2ebc4a0e97b687487c7fe3ef2cbcf6466a0ee22f8af6695534809e9c9
neighbor[0] = 0
neighbor[1] = 1
neighbor[2] = 1
neighbor[3] = 0
-----ATENCAO: n = 0
ERROR: neighbor[2] = [7]
-----ATENCAO: n = 0
ERROR: neighbor[1] = [6]
7fbff8a9b9125ddf9eda4068bd8eedeb01ac3ade7d76eefc374eb1876e10d14
neighbor[0] = 0
neighbor[1] = 1
neighbor[2] = 0
neighbor[3] = 0
31cc79fe6bb6b21e175f5a3f5aeb8d59f2912d2a89883e17e22fd49ed63feb91
neighbor[0] = 0
neighbor[1] = 1
neighbor[2] = 1
neighbor[3] = 0
[]

UbuntuServer-192.168.122.106
gns3@gns3:~/Desktop/FinalTestsV2$ ./client 3
neighbor[0] = 1
neighbor[1] = 0
neighbor[2] = 0
neighbor[3] = 0
b2211ee2ebc4a0e97b687487c7fe3ef2cbcf6466a0ee22f8af6695534809e9c9
4e90cc19d11051aac5c90ca3fb283daee08f33db3d0c6134f9bb99b46dee7a27
neighbor[0] = 1
neighbor[1] = 1
neighbor[2] = 0
neighbor[3] = 0
^C
gns3@gns3:~/Desktop/FinalTestsV2$ ./client 3
neighbor[0] = 0
neighbor[1] = 0
neighbor[2] = 0
neighbor[3] = 1
7fbff8a9b9125ddf9eda4068bd8eedeb01ac3ade7d76eefc374eb1876e10d14
neighbor[0] = 0
neighbor[1] = 0
neighbor[2] = 1
neighbor[3] = 1
fa7c41c01c64343651f5be3cf525287255b771903cb915ff72b41a4fac8b89ee
[]

UbuntuServer-192.168.122.126
gns3@gns3:~/Desktop/FinalTestsV2$ ./client 2
neighbor[0] = 0
neighbor[1] = 0
neighbor[2] = 0
neighbor[3] = 1
f5bbb4672b0fa61273348f10faa358545aa2e2fb95694c13b562335ac8823d3c
1d811b47af10fd7427291faa5b3e15ed6fa106959baf5523b3ad5bc1c655156d
neighbor[0] = 0
neighbor[1] = 0
neighbor[2] = 1
neighbor[3] = 1
^C
gns3@gns3:~/Desktop/FinalTestsV2$ ./client 2
neighbor[0] = 1
neighbor[1] = 0
neighbor[2] = 0
neighbor[3] = 0
31cc79fe6bb6b21e175f5a3f5aeb8d59f2912d2a89883e17e22fd49ed63feb91
neighbor[0] = 1
neighbor[1] = 1
neighbor[2] = 0
neighbor[3] = 0
6fddf7064f937925acb98339c5baf54cda5b40b53b60b72896a8d8e44b74b92
[]

UbuntuServer-192.168.122.239
gns3@gns3:~/Desktop/FinalTestsV2$ ./client 4
neighbor[0] = 1
neighbor[1] = 0
neighbor[2] = 0
neighbor[3] = 0
1d811b47af10fd7427291faa5b3e15ed6fa106959baf5523b3ad5bc1c655156d
neighbor[0] = 1
neighbor[1] = 0
neighbor[2] = 0
neighbor[3] = 1
4e90cc19d11051aac5c90ca3fb283daee08f33db3d0c6134f9bb99b46dee7a27
-----ATENCAO: n = 0
ERROR: neighbor[3] = [10]
-----ATENCAO: n = 0
ERROR: neighbor[0] = [5]
fa7c41c01c64343651f5be3cf525287255b771903cb915ff72b41a4fac8b89ee
neighbor[0] = 1
neighbor[1] = 0
neighbor[2] = 0
neighbor[3] = 0
6fddf7064f937925acb98339c5baf54cda5b40b53b60b72896a8d8e44b74b92
neighbor[0] = 1
neighbor[1] = 0
neighbor[2] = 0
neighbor[3] = 1
[]

UbuntuServer-192.168.122.119
gns3@gns3:~/Desktop/FinalTestsV2$ ./manager
New place 32900
Numero de vizinhos 0
New place 14027
Numero de vizinhos 1
New place 31434
Numero de vizinhos 1
New place 47176
Numero de vizinhos 2
Lost neighbor 2
Reporting node: 31434
Node 31434 lost
Lost neighbor 3
Reporting node: 0
Lost neighbor 1
Reporting node: 14027
Node 14027 lost
Lost neighbor 0
Reporting node: 0
Take over 39226
Numero de vizinhos 2
Take over 5877
Numero de vizinhos 2
[]

```

Figura 42: Reinserção de nós na rede

5.3. Configuração

Cada cliente da rede poderá apenas reencaminhar o tráfego web de um *browser*, ou de todo o seu sistema, abaixo será explicado como configurar ambos os métodos.

Quase todos os *browsers* modernos permitem ser configurados para que seja utilizado o protocolo *SOCKS*.

5.3.1. Configuração do Firefox Browser

Para testarmos a rede, utilizamos o *Firefox Browser*, que facilmente se configura.

1. Abrir as Preferências.

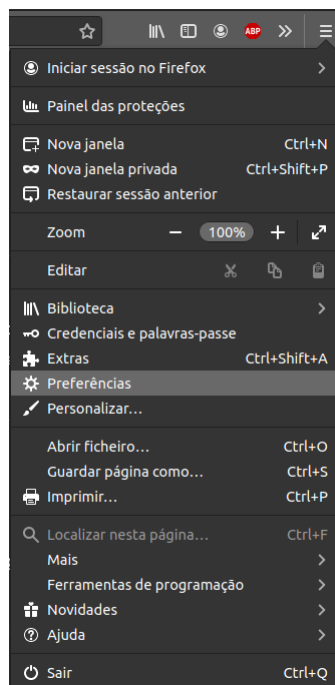


Figura 43: Configuração da *Proxy* do Firefox - Parte 1

2. No painel Geral, abrir as definições de rede no final da página.

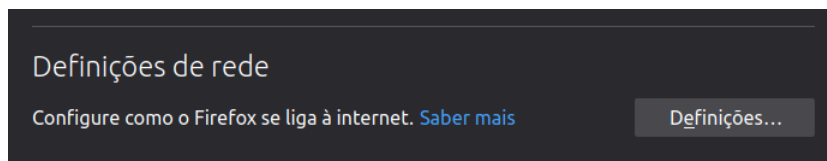


Figura 44: Configuração da *Proxy* do Firefox - Parte 2

3. Configurar como na imagem abaixo:

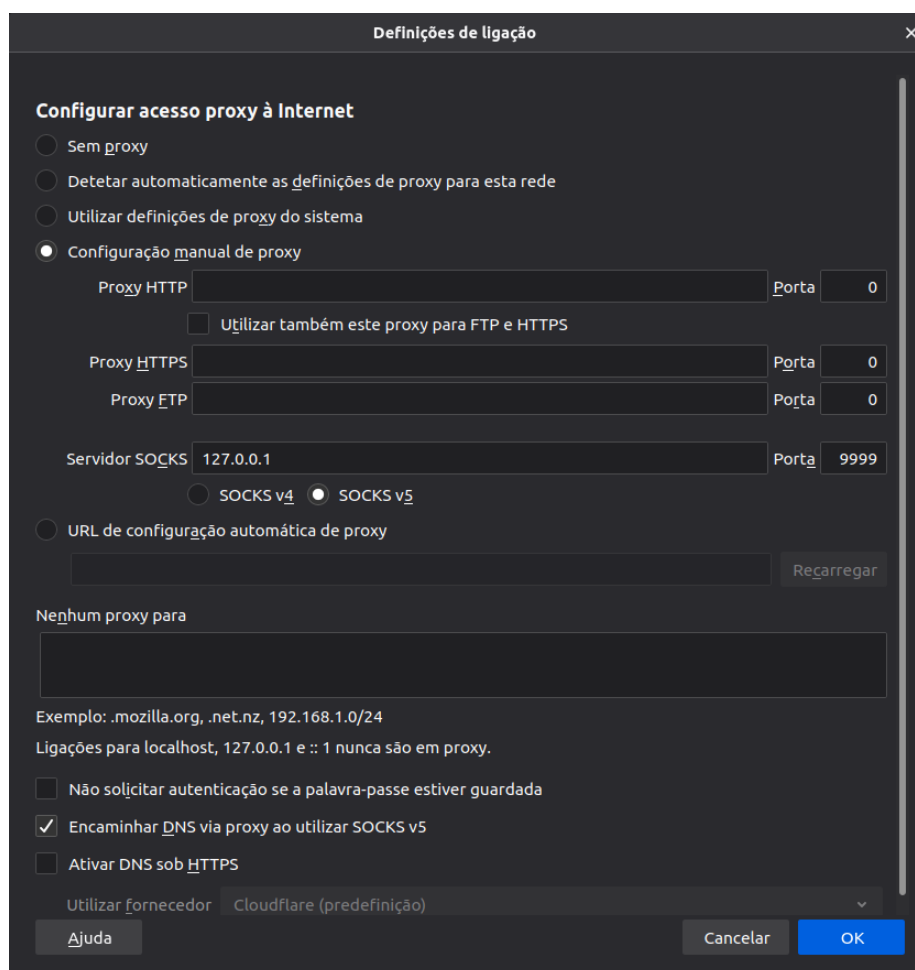


Figura 45: Configuração da *Proxy* do Firefox - Parte 3

Servidor SOCKS será *localhost*, pois a *proxy* que captura o tráfego é local. O nosso programa permite a utilização do *SOCKS v4* ou do *SOCKS v5*.

Se o cliente utilizar a versão 5, recomendamos selecionar a opção "Encaminhar DNS via *proxy* ao utilizar *SOCKS v5*", desta forma também estes pedidos serão anonimizados.

5.3.2. Configuração do sistema para uso da rede

Para encaminhar todo o tráfego web do sistema pela nossa rede, facilmente se configura o cliente com poucas linhas através do terminal:

```
gsettings set org.gnome.system.proxy mode 'manual'
gsettings set org.gnome.system.proxy.socks port 9999
gsettings set org.gnome.system.proxy.socks host 'localhost'
```

Para desativar a *proxy*, basta executar o seguinte comando:

```
gsettings set org.gnome.system.proxy mode 'none'
```

5.4. Discussão

5.4.1. Objetivos não cumpridos

- Durante a idealização da solução, o grupo tinha como objetivo outra abordagem em relação aos caminhos que os pacotes iriam adotar, decidimos que cada pacote, individualmente, iria tomar um caminho aleatório.

Com esta escolha, verificamos que os pacotes não chegavam ao destino pela ordem correta, o que iria ser necessário alterar o protocolo, para que os pacotes fossem numerados e no fim organizados. Esta solução à partida não iria apresentar qualquer problema no desempenho das ligações.

Como já tínhamos o nosso protocolo bem desenvolvido e implementado, e visto que o tempo para adotar estas alterações poderia comprometer a realização dos restantes objetivos, decidimos que cada ligação utilizaria o mesmo percurso na rede. Esta solução continua a satisfazer o nosso objetivo, pois o facto de cada ligação utilizar um percurso, continua a garantir bastante aleatoriedade em cada ligação.

- Outra idealização que tivemos foi a de aplicar certificados nas conexões entre cada cliente para garantir a autenticidade das chaves públicas por eles trocadas. A razão pela qual não nos foi possível a sua implementação, foi a escassez de documentação relativa à biblioteca *OPENSSL*, para a linguagem de programação em que o projeto foi desenvolvido (C++).
- Um dos objetivos traçados pelo grupo, era conseguir visualizar um vídeo na plataforma *Youtube* com a qualidade 4K, assumimos isto como um dos objetivos não cumpridos porque embora o tenhamos conseguido, não foi de forma estável, pois ocasionalmente a ligação era perdida, acreditamos vivamente que esta falha ocorra pelo facto que a rede P2P ter sido testada num ambiente simulado.
- Uma falha detetada aquando dos testes, foi a implementação da versão do *SOCKS4*, que embora nos tenhamos esforçado a tentar solucionar, não oferece um correto funcionamento. Contudo, é de notar, que é possível configurar o *browser* para dar uso à versão *SOCKS5*, que funciona corretamente, e a nosso ver é mais vantajosa em relação às versões anteriores.

5.4.2. Funcionalidades em falta

Com a solução praticamente terminada, foi-nos possível encontrar funcionalidades não implementadas que permitiriam ultrapassar algumas limitações.

- A implementação de um protocolo de estado de ligações iria permitir que os clientes da rede trocassem entre si informações úteis sobre o seu estado, como por exemplo, a sua possibilidade ou não de encaminhar pacotes.

Com este protocolo, poderia-se tentar ultrapassar as limitações indicadas anteriormente na secção 5.1, pois se um cliente pretendesse iniciar uma ligação, poderia saber em ante-mão se um vizinho reúne as condições necessárias para satisfazer o seu pedido, e se assim não o fosse, procurar outro vizinho. Sem esta solução, este irá tentar realizar uma conexão sem garantias que a mesma se virá realizar.

- A criptografia, como mencionado anteriormente utilizando a troca de chaves ECDH e a encriptação AES-CBC, foi apenas implementada na troca de informações entre nós vizinhos. A troca de informação entre os nós e o gestor da rede, quer na fase de *bootstrapping*, quer aquando um nó abandona a rede e requer o envio de pacotes *Patch Up* não é encriptada.

6. Conclusão

Este projeto teve um caráter desafiante, uma vez que a construção de uma rede *overlay peer-to-peer* é algo complexo. Implicou que o grupo aplicasse diversas tecnologias, com diversos protocolos de comunicação, alguns deles já existentes, e outros que tiveram de ser desenvolvidas por nós, nomeadamente a tecnologia que permitiu aos nós conhecerem-se. Também foi necessário desenvolver um protocolo para que os nós da rede encaminhassem tráfego entre eles.

Inicialmente procuramos discutir as possíveis abordagens para o desenvolvimento da solução, tendo realizado a especificação da mesma.

Naturalmente, ao longo da implementação da solução que tínhamos idealizado, foram surgindo obstáculos que nos levaram a alterar algumas das abordagens a que nos tínhamos proposto.

Finalizando, ficamos convencidos que embora nem todos os objetivos que o grupo tinha estabelecido foram concluídos, obtivemos uma boa solução uma vez que esta procura solucionar o objetivo principal, que era anonimizar os pedidos *web* de um cliente.

7. Referências

- [1] **Enunciado PTI1 - Prof. António Costa e Prof. Helena Rodrigues** [acedido em 13.10.2020)].
- [2] **Peer to Peer Overlay Networks: Structure, Routing and Maintenance - Wojciech Galuba e Sarunas Girdzijauskas.** [online] Disponível em: https://link.springer.com/referenceworkentry/10.1007/978-0-387-39940-9_1215 [acedido em 20.10.2020].
- [3] **What is a Proxy Server and How Does it Work? - Jeff Petters** [online] Disponível em: <https://www.varonis.com/blog/what-is-a-proxy-server/> [acedido em 15.10.2020].
- [4] **SOCKS** [online] Disponível em: <https://en.wikipedia.org/wiki/SOCKSSoftware> [acedido em 20.10.2020].
- [5] **Elliptic Curve Diffie-Hellman** [online] Disponível em: https://en.wikipedia.org/wiki/Elliptic-curve_Diffie-Hellman [acedido em 15.01.2021].
- [6] **AES-CBC operation mode.** [online] Disponível em: [https://en.wikipedia.org/wiki/Block_cipher_mode_of_operation#Cipher_block_chaining_\(CBC\)](https://en.wikipedia.org/wiki/Block_cipher_mode_of_operation#Cipher_block_chaining_(CBC)) [acedido em 15.01.2021].