



Universidade do Minho

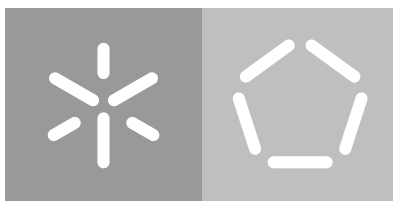
Escola de Engenharia

Departamento de Informática

João Pedro de Moura Pereira e Ferreira Carvalho

Framework de Rede Tolerante a Falhas

Setembro 2021



Universidade do Minho

Escola de Engenharia

Departamento de Informática

João Pedro de Moura Pereira e Ferreira Carvalho

Framework de Rede Tolerante a Falhas

Dissertação de Mestrado

Mestrado Integrado em Engenharia Informática

Dissertação orientada por

Prof. Doutor António Luís Duarte Costa

Setembro 2021

DIREITOS DE AUTOR E CONDIÇÕES DE UTILIZAÇÃO DO TRABALHO POR TERCEIROS

Este é um trabalho académico que pode ser utilizado por terceiros desde que respeitadas as regras e boas práticas internacionalmente aceites, no que concerne aos direitos de autor e direitos conexos.

Assim, o presente trabalho pode ser utilizado nos termos previstos na licença abaixo indicada.

Caso o utilizador necessite de permissão para poder fazer um uso do trabalho em condições não previstas no licenciamento indicado, deverá contactar o autor, através do RepositóriUM da Universidade do Minho.

Licença concedida aos utilizadores deste trabalho



Creative Commons Atribuição-NãoComercial-CompartilhaIgual 4.0 Internacional

CC BY-NC-SA 4.0

<https://creativecommons.org/licenses/by-nc-sa/4.0/deed.pt>

DECLARAÇÃO DE INTEGRIDADE

Declaro ter atuado com integridade na elaboração do presente trabalho académico e confirmo que não recorri à prática de plágio nem a qualquer forma de utilização indevida ou falsificação de informações ou resultados em nenhuma das etapas conducente à sua elaboração.

Mais declaro que conheço e que respeitei o Código de Conduta Ética da Universidade do Minho.

ABSTRACT

A *Fault Tolerant Network* is a Network structure that aims to guarantee Node communication in certain Network environments that are prone to *Datagram* Drops and consequent loss of Information. These Drops can be present due to a multitude of reasons but this Project aims to analyze three specific cases where these can occur, Node Mobility, Intermittent Network Connection and Sporadic Network Connection.

All these cases present the same challenge to Node Communication, there's a possibility of a high volume of *Datagram* Drops that can happen unexpectedly, limits the amount of Data that can be exchanged between Nodes and lowers the Quality of Service of said exchanges. The main objective of this Project is to address these limitations that are innate in these specific cases as well as other topics that are derivative of said limitations such as extensive interruptions in Data Transfers and subsequent resume of these.

Since all the cases described previously present the same challenges, or they can be treated like they are similar, it was decided to develop a Data Transfer Protocol fitted for these limitations. This Protocol is between the Transport and Application Layers of the Network Stack and it's purpose is to improve Data Transfer and Reception between Nodes.

In this Project, a Data Transfer Protocol was designed and implemented that enables Data transmission between Nodes in the cases previously referred, showing resilience against *Datagram* Drops, Transmission Delays and Node Mobility.

In addition to this Protocol, a simple *Ad Hoc* Network Architecture was designed using said Protocol where each Node has a view of the Network centralized onto itself and the Neighboring Nodes organized in Neighbor Levels depending on their distance to the central Node. The objective of designing a simple version of a *DTN* was also established bringing new challenges that strongly influenced the design of the Data Transfer Protocol due to its strict requirements.

Lastly, some tests were performed in real world scenarios in such ways that all the features designed and developed in this Project could be shown to be working properly and that all objectives delined were met.

Keywords: AdHoc; Data Transfer; DTN; Fault Tolerance; Node Mobility;

RESUMO

Uma *Fault Tolerant Network* é uma estrutura de Redes que tem como objetivo garantir a comunicação entre Nodos de uma Rede mesmo que esta seja propícia à perda de *Datagramas* e à consequente perda de Informação. Estas perdas podem acontecer por vários motivos, mas este projeto tem como alvo analisar três casos, o da mobilidade de Nodos na Rede, o da conexão intermitente e o da conexão esporádica.

Todos estes ambientes onde se pretende assegurar a troca de Informação entre Nodos apresentam uma característica em comum, um possível volume elevado de perda de *Datagramas* a qualquer instante que limita a quantidade de Dados que podem ser trocados bem como a Qualidade de Serviço destas mesmas trocas de Dados. Esta é a principal característica que se pretende atenuar com o desenvolvimento deste Projeto, porém existem outras relacionadas como a interrupção prolongada de uma Transmissão e a sua retoma que também foram analisadas.

Como todos os ambientes referidos anteriormente apresentam adversidades semelhantes ou que podem ser tratadas como tal, optou-se pelo desenvolvimento de um Protocolo de Transferência de Dados adaptado a tais adversidades. Este encontra-se entre a Camada de Transporte e a Camada de Aplicações da *Network Stack* e pode ser utilizado como base para o Desenvolvimento de Arquiteturas que possibilitem a Troca organizada de Informação entre Nodos.

Neste Projeto foi Desenhado e Implementado um Protocolo de Transferência de Dados que possibilita a troca de informação nos ambientes anteriormente referidos, apresentando resiliência a *Drops* de *Datagramas*, grandes *Delays* na transmissão destes e movimentações de Nodos na Rede.

Para além deste Protocolo, foi Desenhada uma simples Arquitetura de Redes baseada em Redes *Ad Hoc* onde cada Nodo tem uma visualização da Rede centralizada nele próprio e os restantes Nodos visíveis encontram-se organizados em Níveis de Vizinhança consoante a sua distância ao Nodo *central*. Foi também definido o objetivo de desenho de uma Arquitetura de Redes baseada em Redes *DTN*, que fortemente influenciou o Protocolo de Transferência de dados devido às suas grandes exigências.

Por fim foram realizados testes em determinados cenários reais pertinentes ao Protótipo Implementado de forma a provar que os objetivos delineados inicialmente foram atingidos.

Palavras Chave: AdHoc; DTN; Mobilidade de Nodos; Tolerância a Falhas; Transferência de Dados;

CONTEÚDO

1	INTRODUÇÃO	1
1.1	Enquadramento	1
1.2	Objectivos	1
1.3	Metodologia	2
1.4	Estrutura do Documento	3
2	ESTADO DA ARTE	4
2.1	Delay or Disruption Tolerant Network (DTN)	4
2.1.1	Assunções da Arquitetura DTN	5
2.1.2	Estruturas de Dados	5
2.1.3	Bundles	5
2.1.4	Tipos de contacto	8
2.1.5	Fragmentação e Reconstrução de Bundles	9
2.1.6	EndPoint ID (EID)	10
2.1.7	Registo	10
2.1.8	Custódia	11
2.1.9	Segurança	11
2.1.10	Routing	12
2.2	Protocolos da Camada de Transporte	12
2.2.1	Comunicação Orientada à Conexão	12
2.2.2	Comunicação não Orientada à Conexão	16
2.2.3	Novos desafios e avanços recentes	18
3	TRANSFERÊNCIA DE DADOS TOLERANTE A FALHAS	20
3.1	Arquitetura do Protocolo de Transferência de Dados	20
3.1.1	Processo de Transmissão	21
3.1.2	Troca de Meta Informação (<i>Handshake</i>)	22
3.1.3	Transmissão de Dados (<i>Transfer</i>)	24
3.1.4	Finalização da Transferência (<i>Over</i>)	25
3.1.5	Falhas no Protocolo de Transferência de Dados	26
3.2	Endereços	54
3.3	Estatísticas	55
3.3.1	Datagramas por Segundo	57
3.3.2	<i>Timeout</i>	59
3.4	Segurança	60

3.4.1	Segurança da informação	61
3.4.2	Métodos Criptográficos	61
3.5	Deteção e Correção de Erros	66
4	CONCEÇÃO DA FRAMEWORK	68
4.1	Controlador	68
4.1.1	Contolo de NICs	69
4.1.2	Listener	71
4.1.3	Sender	75
4.1.4	<i>Knock</i> Manager	78
4.2	Sistema de Armazenamento	81
4.2.1	Estrutura	81
4.2.2	Compressão de Dados	83
4.2.3	Métodos	86
5	DESENHO DAS ARQUITETURAS DE REDE	87
5.1	Arquitetura de Redes <i>Ad Hoc</i>	87
5.1.1	Estrutura	87
5.1.2	Adição de Vizinhos	89
5.1.3	Routing	96
5.2	Arquitetura das Redes DTN	96
6	DESENVOLVIMENTO	99
6.1	Decisões	99
6.2	Implementação	100
6.2.1	<i>Messages Package</i>	100
6.2.2	<i>Data Package</i>	103
6.2.3	Network	108
6.2.4	KnockManager	115
6.2.5	Problemas	115
6.2.6	Main	118
7	TESTES E RESULTADOS	120
7.1	Resultados	120
7.1.1	Conexão Directa entre Dispositivos	121
7.1.2	3 <i>Routers</i>	125
7.1.3	1 <i>Router</i>	129
7.1.4	2 <i>Routers</i>	131
7.2	Discussão	134
8	CONCLUSÃO E PROSPEÇÃO DE TRABALHOS FUTUROS	135
	Bibliografia	138

LISTA DE FIGURAS

Figura 1	Estrutura geral do Processo de Transferência de Dados entre um <i>Transmissor</i> e um <i>Recetor</i>	22
Figura 2	Especificação da primeira Iteração do Bloco <i>Handshake</i> do Processo de Transferência	24
Figura 3	Especificação da primeira Iteração do Bloco <i>Transfer</i> do Processo de Transferência	25
Figura 4	Especificação da primeira Iteração do Bloco <i>Over</i> do Processo de Transferência	26
Figura 5	Especificação da segunda Iteração do Bloco <i>Handshake</i> do Processo de Transferência	28
Figura 6	Especificação da segunda Iteração do Bloco <i>Transfer</i> do Processo de Transferência	29
Figura 7	Exemplo prático da Estrutura usada para a compressão dos Identificadores {1285, 1805, 2115, 2116}	30
Figura 8	Exemplo da Estrutura usada para a compressão dos Identificadores que segue a abordagem de <i>Grupos de Identificadores</i>	32
Figura 9	Gráfico de comparação entre % de <i>Chunks</i> recebidos e % de Identificadores a enviar ao <i>Transmissor</i> por ciclo com Confirmação Positiva e sem Mensagens de confirmação de chegada	34
Figura 10	Gráfico de comparação entre % de <i>Chunks</i> recebidos e % de Identificadores a enviar ao <i>Transmissor</i> por ciclo com Confirmação Positiva e com Mensagens de confirmação de chegada	34
Figura 11	Gráfico de comparação entre % de <i>Chunks</i> recebidos e % de Identificadores a enviar ao <i>Transmissor</i> por ciclo com Confirmação Negativa e sem Mensagens de confirmação de chegada	35
Figura 12	Gráfico de comparação entre % de Identificadores enviados ao <i>Transmissor</i> por ciclo em cada uma das 3 estratégias	35
Figura 13	Terceira Iteração do Bloco <i>Handshake</i>	38
Figura 14	Terceira Iteração do Bloco <i>Transfer</i>	39
Figura 15	Segunda Iteração do Datagrama <i>Over</i>	41
Figura 16	Mobilidade de um Cliente e mudança de Endereço	42
Figura 17	Mobilidade do Transmissor durante uma transferência	43

Figura 18	Demonstração do pedido de retransmissão de <i>Datagramas CM</i> devido com a mobilidade do Transmissor	45
Figura 19	Mobilidade do Recetor durante uma transferência	46
Figura 20	Demonstração da perda de <i>Datagramas CM</i> devido a movimentação do Recetor	47
Figura 21	Demonstração de movimento simultâneo dos 2 Clientes	48
Figura 22	Demonstração de um movimento que não é possível estabelecer conexão entre os Clientes	49
Figura 23	Comunicação entre um Cliente C e um Servidor S através de uma Rede NAT	50
Figura 24	Procedimento genérico de <i>NAT Hole Punching</i>	51
Figura 25	Demonstração do problema existente com o <i>Scope</i> dos endereços	53
Figura 26	<i>Hole Punching</i> usando <i>Datagramas Multicast</i>	54
Figura 27	Exemplo de <i>Jitter</i> quando uma Rede se encontra sobre stress	58
Figura 28	Fluxo de Mensagens e ordem de Operações num Sistema de PGP	63
Figura 29	Possibilidades de ordem de realização do Algoritmo de MAC e Cifrar Dados	64
Figura 30	Exemplo de um Ataque Man in the Middle	65
Figura 31	<i>Multipath</i> e <i>MultiTransmitter</i>	70
Figura 32	Exemplos de Distribuição da Banda Larga disponível	70
Figura 33	Utilização de Portas conforme é especificado no Protocolo, sem qualquer otimização	72
Figura 34	Utilização de Portas conforme é especificado no Protocolo, mas é usada uma única Porta de Gestão para todas as Transferências a serem realizadas	73
Figura 35	Agrupamento das portas de receção de <i>Datagramas</i> , tornando-se independente do número de Transferências e reduzindo a carga no CPU do Dispositivo	74
Figura 36	Representação dos estados que as Transferências se podem encontrar e como é que este se altera conforme os acontecimentos do Protocolo	77
Figura 37	Emissão de <i>Datagrama Knock Multicast</i> para descoberta de Nodos na Rede	79
Figura 38	Diferentes comportamentos perante o uso de diferentes mecanismos de Resposta a uma Mensagem Multicast	79
Figura 39	Fluxo de <i>Datagramas Knock Multicast</i> e <i>Unicast</i>	80
Figura 40	Estrutura dos Ficheiros do Sistema de Armazenamento descrito	83

Figura 41	Processo de compressão após Divisão dos Dados em <i>Chunks</i> originando x <i>Chunks</i> com tamanhos bastantes menores que o tamanho máximo do <i>payload</i> do <i>Datagrama</i>	84
Figura 42	Processo de compressão antes da Divisão dos Dados em <i>Chunks</i> originando y <i>Chunks</i> com tamanhos iguais ao tamanho máximo do <i>payload</i> do <i>Datagrama</i>	85
Figura 43	Processo de compressão e Divisão simultânea dos Dados em <i>Chunks</i> originando z <i>Chunks</i> com tamanhos variáveis, mas que maioritariamente ficam perto do tamanho máximo do <i>payload</i> do <i>Datagrama</i>	85
Figura 44	Visualização da Rede centrada num <i>Nodo</i> e dividida em níveis de <i>Vizinhos</i> conforme a Distância entre esses <i>Nodos</i> e o <i>Nodo</i> Central	89
Figura 45	Alcance da Interface do <i>Nodo</i> N em que a comunicação com outros <i>Nodos</i> é possível	90
Figura 46	Processo de Adição de um <i>Nodo</i> a <i>Vizinho</i> de primeiro Nível	90
Figura 47	Número Máximo de <i>Nodos</i> distintos que um <i>Nodo</i> N pode conhecer e manter informação atualizada se for utilizado um limite máximo de <i>Vizinhos</i> de primeiro Nível	91
Figura 48	Adição de um <i>Nodo</i> como <i>Vizinho</i> quando um deles não possui qualquer <i>Vizinho</i> e o outro não tem qualquer vaga	92
Figura 49	Calculo da Distância entre 2 <i>Nodos</i> é feito conforme a interceção dos <i>Níveis</i> de <i>Vizinhos</i> . Este exemplo representa uma distância imprecisa.	93
Figura 50	Junção de 2 Redes Distintas	94
Figura 51	Actualização da informação relativa aos <i>Vizinhos</i> utilizando o Controlador de <i>Knocks</i>	95
Figura 52	Fecho de uma Ligação entre <i>Nodos</i> por estes já não se encontrarem dentro do alcance de comunicação	95
Figura 53	Fecho de uma Ligação que o <i>Nodo</i> N possuía com um outro <i>Nodo</i> utilizando um <i>Datagrama</i>	96
Figura 54	Diagrama de Classes UML simplificado do <i>Messages Package</i>	103
Figura 55	Diagrama de Classes UML simplificado do <i>Data Package</i>	108
Figura 56	Diagrama de Classes UML simplificado do <i>Network Package</i>	115
Figura 57	Diagrama de Classes UML simplificado do Protótipo Desenvolvido	119
Figura 58	Conexão direta entre os 2 Dispositivos	121
Figura 59	Transferência de um Ficheiro de 512 <i>MBytes</i> numa ligação de 1000 <i>Mbps</i> estável	122
Figura 60	Início de uma Transferência e a sua interrupção.	123
Figura 61	Retoma e conclusão de uma Transferência interrompida	124

Figura 62	Estrutura da Rede utilizada para os testes que usam 3 <i>Routers</i>	125
Figura 63	Transferência através de um <i>NIC Wireless</i> num ambiente com interferência	126
Figura 64	Transferência onde o Transmissor se move na Rede	127
Figura 65	Transferência onde o Recetor se move na Rede	128
Figura 66	Estrutura da Rede Utilizada para o Teste de detecção de Nodos na Rede	129
Figura 67	Detecção de Nodos na mesma Rede utilizando o Mecanismo de troca de <i>Datagramas Knock</i>	130
Figura 68	Estrutura da Rede para o Teste de <i>contorno</i> de Redes <i>NAT</i>	131
Figura 69	Detecção de Nodos na Rede utilizando <i>Datagramas Knock</i> através de uma Rede <i>NAT</i>	132
Figura 70	Visualização da saída da <i>Datagramas TMI</i> usando <i>Wireshark</i>	133

ACRÓNIMOS

ACK Acknowledge. 1, 13–16, 27, 32, 40

ADU Application Data Unit. 1, 5–7

AIMD Additive Increase / Multiplicative Decrease. 1, 14, 57

CM Chunk Message. 1, 24, 33, 35–37, 44, 46, 56, 57, 59, 71, 72, 76, 77, 84, 109, 111–114, 123, 126

CPU Central Processing Unit. viii, 1, 73, 74, 111, 120

DMZ Demilitarized Zone. 1, 125

DPS Datagramas por Segundo. 1, 56–59, 110, 113

DTN Delay Tolerant Network. 1–12, 55, 66, 81, 97, 105, 108, 114, 135, 137

ECC Error Correction Code. 1, 66

EID Endpoint Identifier. 1, 5–8, 10

FIN Finish. 1, 14

IANA Internet Assigned Numbers Authority. 1, 10, 17, 18

ID Identifier. 1, 23

IDE integrated development environment. 1, 100

IETF Internet Engineering Task Force. 1, 19

IP Internet Protocol. 1, 15–17

IPS Intrusion Prevention Services. 1, 61

IPv4 Internet Protocol version 4. 1, 17, 49, 55

IPv6 Internet Protocol version 6. 1, 17, 55, 80, 88, 116, 117

JAR Java ARchive. 1, 100

JVM Java Virtual Machine. 1, 99

MAC Message Authentication Code. viii, 1, 63, 64

MAC Media Access Control. 1, 55, 88

MCID Missing Chunk ID. 1, 36, 37, 40, 43, 44, 46, 59, 71, 77, 102, 110, 111, 113, 114

MSS Maximum Segment Size. 1, 13, 14

MTU Maximum Transfer Unit. 1, 13, 15, 17, 21, 27, 31, 53

NAT Network Address Translation. viii, x, 1, 49–54, 101, 102, 117, 125, 131–133, 135, 136

NIC Network Interface Card. x, 1, 41, 55, 69–71, 73–77, 114–117, 120, 125, 126, 133–135

NSU Network Status Update. 1, 42–46, 48, 71, 101, 102, 110, 113, 117

NVME Nonvolatile Memory Express. 1, 120

P2P Peer-to-Peer. 1, 47, 48, 88, 96

PGP Pretty-Good-Privacy. 1, 62, 65

RAM Random Access Memory. 1, 86, 105, 112, 113, 120, 134

RTT Round Trip Time. 1, 56, 59, 109, 111, 115

SATA Serial Advanced Technology Attachment. 1, 120

SDNV Self-Delimiting Numeric Values. 1, 7

SSD Solid State Drive. 1, 120

SYN Synchronize. 1, 13

TCP Transmission Control Protocol. 1, 12–16, 19, 27, 32, 57

TMI Transfer Meta Info. x, 1, 23, 24, 41, 76, 77, 102, 114, 122, 133

TMRI Transfer Multi Receiver Info. 1, 38, 41, 71, 76, 77, 102, 110, 111, 113, 114

TRI Transfer Receiver Info. 1, 23, 24, 27, 37, 38

UDP User Datagram Protocol. 1, 12, 16–19, 53, 78, 101, 109

URI Universal Resource Identifier. 1, 10

INTRODUÇÃO

1.1 ENQUADRAMENTO

Atualmente muitas aplicações dão como garantido que os seus utilizadores terão uma boa conexão à Internet e que esta seja estável. Embora isto seja verdade para muitos casos, existem locais onde este acesso estável e de boas condições ainda é inexistente e, por isso, seja impossível o uso de tais aplicações que requerem uma conexão a servidores centrais durante a sua utilização. Muitas vezes estas aplicações poderiam possuir um modo *Offline* que permita a utilização parcial do produto por parte do cliente e, sempre que possível, comunicaria os dados necessários aos servidores centrais [1]. Estes problemas podem estar presentes devido à mobilidade dos utilizadores, o que não possibilita uma conexão estável com Pontos de Acesso que podem estar presentes e dentro do alcance.

Com a evolução dos dispositivos móveis que uma pessoa possui e transporta no seu dia a dia também apareceu a oportunidade de explorar a comunicação entre estes. A esta ligação de dispositivos sem a necessidade de qualquer Ponto de Acesso dá-se o nome de Rede *Ad Hoc* [2]. Estas Redes apresentam um grande desafio na transmissão de informação entre Nodos porque estes tratam-se de dispositivos móveis, podem estar em constante movimento e a sua permanência na Rede não é assegurada. Com estas características é normal a existência de grandes perdas de *Datagramas* e, por vezes, a impossibilidade de qualquer transmissão ou receção de dados.

Este projeto tem como objetivo a conceção de uma *Network Framework* que permita a Transmissão de Dados nestes ambientes anteriormente descritos. Também é desejável que esta *Network Framework* seja capaz de operar de forma competente em ambientes estáveis e propícios à Transferência de Dados entre Nodos da mesma Rede.

1.2 OBJECTIVOS

O principal objetivo deste Trabalho é o Desenho e Implementação de uma *Network Framework* que permita a Transferência de Dados entre dois ou mais Nodos da mesma Rede onde possam existir momentos em que a troca de *Datagramas* entre eles não seja realizada

devidamente. Estes momentos podem ocorrer devido à falta de conexão à Rede, momentânea ou extensa, por parte dos Nodos, à conexão ser intermitente ou instável, ou até à inexistência de uma Rede. Podemos dar o nome de *Fault Tolerant Networks* a este tipo de Redes que conseguem manter a comunicação entre Nodos mesmo que esta seja comprometida pelo *Drop* constante de *Datagramas* [3] [4]. Também é desejável que esta *Framework* permita a comunicação em Redes com um funcionamento normal e que seja capaz de transferir informação de forma ordinária.

Inicialmente serão analisadas Tecnologias já existentes que poderão influenciar o desenvolvimento desta *Network Framework* ou até serem utilizadas neste. Após esta análise serão iniciados os seguintes processos:

- Desenhar um Protocolo de Transferência de Dados que permita a troca de informação entre Nodos mesmo que haja intermitência na comunicação ou até momentos de desconexão. Este Protocolo também tem que ser capaz de lidar com a possibilidade de existência de Nodos Móveis na Rede e, por isso, ser capaz de manter o seu bom funcionamento em situações de saída da Rede de tais Nodos e situações de elevadas perdas de *Datagramas* (superiores a 10%) [5][6]. Este Protocolo também deverá de ser apto a ser usado numa Rede "convencional" (estável, sem Nodos Móveis, e perda de *Datagramas* inferior a 8% [5][6]);
- Desenhar uma simples Arquitetura de Redes *Ad Hoc* que tire partido do Protocolo de Transferência de Dados referido anteriormente;
- Desenhar uma simples Arquitetura de Redes *Delay Tolerant Network (DTN)* [7] que tire partido do Protocolo referido no primeiro ponto.

Por fim, pretende-se a Implementação de Protótipos do Protocolo de Transferência de Dados e Arquiteturas de Redes *Ad Hoc* e *DTN*, sendo que estes serão testados em ambientes reais e não em simuladores.

1.3 METODOLOGIA

A metodologia seguida baseia-se no desenho, implementação e teste de um Protótipo funcional em ambiente real. Inicialmente foi feita uma fase de estudo do estado da arte bem como das Tecnologias a utilizar e as suas limitações. Após este estudo seguiu-se uma fase de estabelecimento de requisitos e desenho das funcionalidades da *Framework* a implementar. De seguida foram realizadas 2 tarefas quase em simultâneo, a implementação da *Framework* e a realização de testes. Após a Implementação dos Protótipos, foram realizados os testes finais e a consequente análise dos resultados obtidos. Por fim foi feita uma análise do trabalho desenvolvido ao longo da dissertação e se os objectivos delineados foram cumpridos.

1.4 ESTRUTURA DO DOCUMENTO

Este Documento está estruturado em 8 Capítulos. Neste Primeiro foram apresentados os objetivos deste Projeto como o Desenvolvimento de um Protocolo de Transferência de Dados e a Implementação de Protótipos que tirem partido desse mesmo Protocolo e, finalmente, foram detalhados os métodos de trabalho que serão seguidos no desenvolvimento deste.

No Segundo Capítulo (2) são analisadas Tecnologias que influenciam a lógica e o desenho do Protocolo de Transferência de Dados que por sua vez influencia o funcionamento das Arquiteturas de Rede *AdHoc* e *DTN*.

No Terceiro Capítulo (3) é feito o Desenho do Protocolo de Transferência de Dados. Devido a grandes exigências, o desenho deste segue uma filosofia de iterações, ou seja, inicialmente é apresentada uma versão muito simplificada que delinea as funções básicas deste mas que apresenta problemas e não satisfaz todas as características que são pretendidas. Desta forma são apresentadas múltiplas iterações que apresentam melhorias relativamente às versões anteriores, mas que poderão ter os seus próprios problemas que serão analisados e discutidos nas versões seguintes.

No Quarto Capítulo (4) são feitas divisões do Protocolo de Transferência de Dados Tolerante a Falhas em Categorias conforme as responsabilidades de cada uma delas. Juntamente com estas Categorias que auxiliam a Implementação da *Framework*, são apresentadas otimizações que tencionam reduzir o consumo de recursos usados durante a execução deste Protocolo. Finalmente são introduzidos conceitos úteis como o *Multipath* e *Multitransmitter*.

No Quinto Capítulo (5) são desenhadas as Arquiteturas de Redes *AdHoc* e *DTN* tirando partido de todas as funcionalidades do Protocolo de Transferência de Dados Tolerante a Falhas anteriormente desenhadas.

O Sexto Capítulo (6) deste Documento tem o objetivo de apresentar os processos de Implementação dos Protótipos a realizar, expondo decisões tomadas, estruturação do código desenvolvido e problemas encontrados neste processo.

No Sétimo Capítulo (7) são apresentados todos os Testes a que os Protótipos foram submetidos, discutindo-se os resultados obtidos, possíveis otimizações e que problemas podem ser observados.

Finalmente, no Oitavo Capítulo (8), é feita uma análise final do Trabalho realizado neste Projeto, que conclusões se pode tirar com os resultados obtidos e apresentada uma lista de Trabalhos Futuros que melhorariam o Trabalho desenvolvido neste Projeto e certas variantes deste Projeto que poderiam ser realizadas.

ESTADO DA ARTE

Neste capítulo serão analisadas as Tecnologias existentes que poderão vir a auxiliar o desenvolvimento deste projeto. Como Tecnologia principal, será analisado o funcionamento, requerimentos e limitações da Arquitetura de Redes *DTN*, sendo posteriormente analisados os 2 Paradigmas de Protocolos da Camada de Transporte, indicando as vantagens e desvantagens que cada um apresenta.

2.1 DELAY OR DISRUPTION TOLERANT NETWORK (DTN)

Ao contrário de outras Arquiteturas de Redes mais convencionais, uma Arquitetura *DTN* [7][8][9] adota o conceito de uma Rede em que os seus Nodos estão ocasionalmente conectados e que pode sofrer partições de forma frequente. Nestes ambientes é de esperar que haja um *delay* intrínseco que torna difícil o uso de Protocolos de troca de informação tradicionais.

Esta Arquitetura é definida por um *Overlay* orientado à troca de Mensagens (e não à troca de *Datagramas*) que se localiza entre a Camada de Transporte e a Camada das Aplicações. A esta nova camada dá-se o nome de *Bundle Layer* [10] e todos os Nodos que a implementem são designados de *Nodos DTN*.

A *Bundle Layer* tem como objetivo servir de intermediário entre Aplicações que pretendem comunicar numa rede intermitente e os Protocolos da Camada de Transporte, garantindo que as Mensagens a transmitir vão ser propagadas sempre que existir uma ligação entre *Nodos DTN*. Visto que é pretendida a troca de informação numa rede intermitente, com *delays* potencialmente elevados, e que sofre partições frequentemente, é necessário adotar uma filosofia de *Store-and-Forward*. Nesta filosofia sempre que um *Nodo* recebe uma Mensagem tem como prioridade principal guarda-la em memória persistente e apenas depois de o fazer é que procura o próximo *Nodo* a quem deve passar esta Mensagem. Desta forma é possível que as Mensagens sobrevivam a *System Restarts* e que cada *Nodo DTN* tenha a capacidade de armazenar Mensagens para transmissão futura. No caso dos Nodos não conseguirem transmitir a informação imediatamente e necessitarem de se movimentar para obterem uma ligação para o fazer, esta filosofia passa a ser designada por *Store-Carry-and-Forward*.

2.1.1 Assunções da Arquitetura DTN

Devido à natureza desta Arquitetura, existem certas assunções que devem de ser tomadas em conta antes de considerar o uso de uma Rede DTN.

- Por princípio a Arquitetura assume que todos os Nodos presentes na Rede terão Memória Física suficiente para guardar os **Bundles** que lhes são transmitidos;
- A memória Física que os Nodos possuem é suficientemente persistente e robusta;
- Este modelo de *Store-and-Forward* é o melhor modelo para a situação em que vai ser usada (não será o melhor modelo para uma rede estável e com delays reduzidos).

Estas assunções fazem com que este novo recurso necessário em todos os Nodos DTN, a memória persistente, tenha que ser protegido e gerido para o bom funcionamento da Rede DTN.

2.1.2 Estruturas de Dados

Na Arquitetura DTN existe uma estrutura de dados organizada que tenta generalizar a informação que é transmitida entre Nodos DTN. Toda a informação que uma aplicação pretende que seja transferida a partir da Rede DTN é chama de *Application Data Unit (ADU)* que pode ser de tamanho variável. É esta estrutura de dados que é usada para fazer a divisão da informação noutra estrutura de dados (*Bundles*) para ser transferida entre Nodos.

2.1.3 Bundles

Um *Bundle* tem como objetivo dividir a informação de um determinado ADU em estruturas de dados mais simples e com informação que ajude a identificar a sua origem, destino e ordem. Para isto, a estrutura *Bundle* é constituída por 2 *sub-estruturas*:

- Primary Block
- Payload Block

2.1.3.1 Primary Block

Neste Bloco pode ser encontrada toda a *Meta-Informação* relacionada com o próprio *Bundle*.

- **Endpoint Identifier** de Origem
- **Endpoint Identifier** de Destino

- **Endpoint Identifier** de custódia
- **Endpoint Identifier Report-To**
- Processing Control Flags
- Número de Sequência
- Timestamp de Criação
- Tempo de vida útil
- Tamanho

Primeiramente são encontrados os **Endpoint Identifier** que identificam os *Nodos Delay Tolerant Network* Origem e Destino de um determinado *Bundle*. De seguida encontra-se o **Endpoint Identifier** do *Nodo Delay Tolerant Network* que possui a custódia do *Bundle* em questão. Este *Nodo* terá a responsabilidade de guardar e retransmitir o **Bundle** até o Tempo de vida útil do mesmo acabar ou até passar esta responsabilidade a outro *Nodo Delay Tolerant Network*. Desta forma é possível encurtar o tempo de espera de retransmissão de um *Bundle* em caso de falha e reduzir o número total de pacotes transmitidos entre *Nodos* na mesma situação, visto que a informação se encontra mais perto do Destino. O último **Endpoint Identifier** que aparece é referente a um *Nodo Delay Tolerant Network* que pode ser diferente que o da Origem e que serve para serem enviadas Mensagens de Diagnostico.

Também podem ser encontrados neste *Primary Block* o *Timestamp* de Criação e o Tempo de vida Útil que servem para indicar quando é que este *Bundle* pode ser descartado. Para além disto, o *Timestamp* de Criação, juntamente com o **Endpoint Identifier** de Origem, o Número de Sequência e o Tamanho do *Bundle* é usado para identificar o *Bundle* unicamente.

2.1.3.2 *Payload Block*

Este bloco tem o objetivo de encapsular toda a informação do *ADU* de forma simplificada. Isto faz com que apenas tenha 3 componentes:

- Tamanho do *Payload*
- Processing Control Flags
- *Payload*

2.1.3.3 *Processing Control Flags*

Este bloco é um componente dos 2 últimos blocos, mas tem funções e tamanhos diferentes dependendo do bloco em que se encontra. Enquanto no *Primary Block* ocupa 21 bits (3 bytes

em *Self-Delimiting Numeric Values (SDNV)* [11] e o seu *Scope* é o *Bundle*, no *Payload Block* apenas ocupa 7 bits (1 byte em *SDNV*) e o *Scope* é apenas o próprio bloco.

Estas são as Flags presentes no *Primary Block* pela ordem que são lidas:

- Bundle é um Fragmento;
- *ADU* é um administrative record;
- Bundle não pode ser fragmentado;
- Pedido de transferência de Custodia;
- *Endpoint Identifier (EID)* de Destino é um *Singleton*;
- Pedido de *Acknowledgement* à aplicação;
- Reservado para uso Futuro.

De seguida é usado 1 byte *SDNV* para indicar a *Class of Service* do *Bundle*. Este atributo define a prioridade de transmissão deste *Bundle* sobre outros *Bundles* com o mesmo *Endpoint Identifier (EID)* de Origem presentes no mesmo *Nodo DTN*.

No último byte *SDNV* estão presentes as Flags:

- Solicitar indicação de receção do *Bundle*;
- Solicitar indicação de aceitação da Custodia;
- Solicitar indicação de encaminhamento do *Bundle*;
- Solicitar indicação de entrega do *Bundle*;
- Solicitar indicação de Eliminação do *Bundle*;
- Reservado para uso Futuro;
- Reservado para uso Futuro.

Visto que o *Scope* das Flags presentes no *Payload Block* é mais pequeno, não são necessárias tantas. Estas são as 7 Flags presentes no *Payload Block*:

- Este bloco deve de ser replicado em todos os fragmentos.
- Enviar relatório de estado se o bloco não conseguir ser processado.
- Apagar *Bundle* se o bloco não conseguir ser processado.
- É o último bloco.

- Apagar bloco se não conseguir ser processado.
- O bloco foi encaminhado sem ser processado.
- O bloco contém um campo de referência a um [Endpoint Identifier \(EID\)](#).

2.1.4 Tipos de contacto

Esta Arquitetura tem como objetivo permitir a transmissão de informação em situações de intermitência ou de completa rutura da Rede. Esta característica força a que a troca de dados entre 2 *Nodos DTN* tenha que ser analisada de outra forma, visto que este acontecimento pode ser raro ou com bastantes momentos em que não existem Links de conexão, é daqui que surge o conceito de *Contactos*. Um *Contacto* é um momento em que a troca de informação entre 2 *Nodos DTN* a um salto (*Hop*) de distância acontece. Um contacto entre 2 *Nodos DTN* tem 6 características fundamentais que influenciam a transmissão de dados:

- Capacidade do Link
- *Delay* do Link
- Direcção do Link
- Duração do Contacto
- Volume do Contacto
- Periodicidade do Contacto

A Capacidade do Link é a medida que permite saber que quantidade de informação (em bits) que é possível ser transmitida através de um Link num determinado momento do contacto. Em semelhança, o *Delay* do Link é o atraso da chegada da informação ao Nodo Destino num determinado momento do Contacto. A Direcção do Link indica se o Link é Bidirecional ou Unidirecional, sendo que este último indicaria qual a direcção em que os dados fluem. De forma geral, é assumido que estas características são constantes na duração integral do contacto.

Com esta informação seria possível calcular uma estimativa do tempo que um determinado *Bundle* demoraria a ser entregue a outro Nodo. Dado um *Bundle* com B bits e tendo a Capacidade e Delay do Link para o Contacto já calculadas, a função:

$$D(t) + (1/C(t)) * B \quad (1)$$

indicaria o tempo esperado para a transmissão do *Bundle*.

Se a Duração do contacto for conhecida *a priori* é possível calcular o Volume do Contacto que indicaria a quantidade de informação (em bits) que seria possível transmitir durante dado Contacto. Dada a Duração do contacto D e tendo a Capacidade do Link já calculada, seria possível obter o Volume do Contacto a partir de:

$$C(t) * D \quad (2)$$

Seguindo este conceito de Contactos, é possível categorizá-los conforme a sua Periodicidade. Esta é dividida nas seguintes categorias:

- Persistente
- Intermitente
 - Agendadas
 - oportunistas
 - Previstas

Uma Periodicidade Persistente indica que os Nodos estão em constante alcance um do outro e que a troca de informação pode acontecer a qualquer momento.

Uma Periodicidade Intermitente indica que um *Nodo DTN* nem sempre vai conseguir transmitir a informação que possui, e por isso terá que esperar para tal contacto acontecer. Dentro desta periodicidade podemos ainda caracterizar o contacto como sendo *Intermitente Agendado* (quando o Nodo sabe que num determinado momento existirá um contacto com um determinado *Nodo DTN*), *Intermitente Oportunista* (quando 2 Nodos entram em contacto inesperadamente) e *Intermitente Prevista* (quando um Nodo estima um contacto com um Nodo que pode ou não conhecer).

2.1.5 Fragmentação e Reconstrução de Bundles

A estrutura de dados *Bundle* está bem preparada para sofrer Fragmentações. Com o uso de Flags de fragmentação e identificação única utilizando múltiplos campos, um *Bundle* pode ser fragmentado em múltiplos *Bundles*, sendo que estes podem ser novamente fragmentados. Com a mesma lógica de identificação única, os *Bundles* podem ser reconstruídos novamente.

A Fragmentação e, por consequência, a Reconstrução de *Bundles* é uma ferramenta bastante útil, pois trás uma certa segurança à transmissão de *Bundles* entre Nodos. A maior parte das vezes esta ferramenta é usada para a fragmentação *a priori* de *Bundles* tendo em consideração o *Volume do Contacto* se estiver disponível. Um *Nodo DTN* se for capaz de prever um contacto com um outro Nodo através da sua Periodicidade e se souber o Volume do Contacto, consegue calcular que *Bundles* devem de ser Fragmentados para que a

quantidade de informação a ser transmitida não exceda o Volume do contacto e por sua vez que leve a perda de Informação.

Em raros casos esta fragmentação pode ocorrer durante a transmissão de *Bundles*. Caso um *Bundle* esteja a ser transmitido e a sua transmissão deixa de ser possível devido a um fim inesperado do Contacto, os *Nodos DTN* podem fragmentar o *Bundle a posteriori* de forma a tornar válida a informação que foi transmitida sem erros.

2.1.6 EndPoint ID (EID)

Uma *Rede DTN* tem o objetivo de homogeneizar a Identificação de Nodos de forma a ser possível ter uma rede de troca de dados que utilize múltiplos Protocolos de Transporte. Para isso recorre à utilização de *Universal Resource Identifier (URI)* [12][13] para a identificação de Nodos. A este tipo de Identificação dá-se o nome de *Endpoint Identifier* e tem a capacidade de poder identificar desde 1 Nodo até *Multicast*. Isto é devido ao uso de *Wildcards* como o "*" ou "." que permitem que vários Nodos sejam identificados ao mesmo tempo. Um *Nodo DTN* pode fazer parte de vários *Endpoint Identifier* e é obrigatório que seja identificado pelo menos por um. Se um *Endpoint Identifier* apenas identificar um *Nodo DTN* diz-se que este *Endpoint Identifier* se trata de um *Singleton*.

Certas combinações de *Wildcards* são capazes de fazer com que um *Bundle* seja distribuído em *Unicast*, *Anycast*, *Multicast* ou *Broadcast*.

Um *Endpoint Identifier* tem 2 componentes, um *Scheme* que identifica as propriedades lexicais a serem usadas na decifração do *Endpoint Identifier* e um *Scheme-Specific Part* que é a parte a ser decifrada para identificar os Nodos. É importante referir que os *Schemes* são controlados pela *Internet Assigned Numbers Authority (IANA)* [14][15][16].

2.1.7 Registo

O processo de registo é o processo pelo qual uma aplicação informa o *Nodo DTN* que pretende que todos os *Bundles* direcionados a um determinado *EID* lhe sejam passados. Visto que é difícil saber quando é que um *Bundle* vai ser entregue e que os *Nodos DTN* podem sofrer *Application and System Restarts* esta informação deve de ser guardada em memória persistente.

Neste processo de registo, se uma aplicação pretender "ouvir" um *EID Unicast*, também terá que indicar um tempo de vida útil para que este registo seja cancelado. Também é possível que a aplicação informe o *Nodo EID* que não pretende mais "ouvir" um determinado *EID*. Se a aplicação pretender ouvir um *EID Multicast* terá que indicar um intervalo de tempo $[t_0, t_1]$ onde $t_0 \leq \text{tempoatual}$ e $t_1 > \text{tempoatual}$, pois pode já ter havido tráfego naquele grupo e desta forma o *Nodo EID* pode encaminhar os *Bundles* que ainda possui. Isto causa outros

problemas, como a falta de memória para arquivar todos os *Bundles* de um grupo *Multicast* e por isso pode não ser implementado.

2.1.8 Custódia

A funcionalidade de Custódia é muito importante numa arquitetura de Redes como esta pois, em caso de alguma falha de transmissão de Dados, esta ajuda a encurtar o processo de recuperação dos mesmos. Isto acontece pois, quando esta opção esta selecionada no *Primary Block* de um *Bundle*, todos os Nodos que o receberem e tiverem capacidade de memória vão guardá-lo e apenas o apagam quando transferirem a Custódia para outro Nodo ou o Tempo de vida Útil do *Bundle* acabar. Cada vez que um Nodo toma a Custódia de um *Bundle*, este atualiza o campo correspondente no *Primary Block* a informar que ele próprio possui Custódia. Isto ajuda em casos de pedidos de retransmissão de um *Bundle* pois deixa de ser o *Nodo DTN* Origem a retransmiti-lo e passa a ser o último Nodo que possui a Custódia, reduzindo o número de saltos necessários e poupando bastante tempo.

Como disse nem todos os *Nodos DTN* têm a capacidade de aceitar a Custódia de um *Bundle*. Isto acontece, pois, pode haver uma falta de recursos necessários para o fazer desde falta de memória suficiente, falta de bateria, entre outros. Devido a este impedimento é necessário que o processo de *Transferência de Custódia* seja realizado em forma de um pedido indicado numa Flag do *Primary Block* e que necessite de uma confirmação para possibilitar a libertação de recursos.

Mesmo com estas medidas existe um caso que não está coberto. Uma aplicação pode querer que o *Nodo DTN* Origem tome Custódia de um *Bundle* que ela vai transmitir e este pode não conseguir tomar a Custódia por falta de recursos. Quando este caso extremo acontecer, a Aplicação deve de indicar que quer que o *Nodo DTN* Recetor tome Custódia do *Bundle*, sendo que o processo de envio pode sofrer um atraso devido à mesma falta de recursos.

2.1.9 Segurança

Devido a esta nova abordagem de Transmissão de Informação Numa Rede, é introduzido um novo recurso importante em todos os Nodos da Rede, a Capacidade da Memória de cada Nodo. Este recurso tem que ser protegido pois é o principal fator que permite o reencaminhamento de informação em Redes intermitentes ou que sofrem constante rutura. Assim são aconselhadas as seguintes medidas:

- *WhiteList* de Aplicações com permissões para usar a Rede *DTN*

- Impedir que as aplicações transmitam informação a uma taxa de débito elevado ou com *Class of Service* que não têm permissão de usar
- *Drop* de *Bundles* danificados ou que foram fragmentados indevidamente
- Remover aplicações ou entidades da *Whitelist* caso estas tenham sido comprometidas
- Autenticação de *Nodos DTN*
- Verificação de integridade dos *Bundles*

Existe uma noção de *Overlay* de Segurança onde apenas alguns *Nodos* da *Rede DTN* é que realizariam estes processos de Verificação, mas seria obrigatório forçar o *Routing* de *Bundles* por pelo menos 1 destes *Nodos* o que ainda se encontra em estudo.

2.1.10 Routing

Devido à inconsistência da presença de *Nodos* que é possível encontrar em arquiteturas de redes como esta, o *Routing* de *Bundles* é uma matéria ainda em estudo. Para tirar partido desta rede é necessário contar com todas as variáveis adicionais como a Periodicidade e Volume dos Contactos entre *Nodos DTN*, a Capacidade total e disponível de memória que cada *Nodo DTN* possui, caminhos mais apropriados, capacidade de um *Nodo DTN* conseguir tomar Custódia de *Bundles* e muitos outros.

2.2 PROTOCOLOS DA CAMADA DE TRANSPORTE

Para a realização desta Framework é necessário escolher um *paradigma* de *Protocolos da Camada de Transporte* [17] que consiga satisfazer as necessidades das *Redes DTN* delineados anteriormente. Existem dois *paradigmas* de *Protocolos da Camada de Transporte*, *Comunicação Orientada à Conexão* e *Comunicação não Orientada a Conexão* (ou *Comunicação orientada ao Datagrama*).

Para este trabalho irei analisar os Protocolos *Transmission Control Protocol (TCP)* e *User Datagram Protocol (UDP)* como possíveis opções para Protocolo de Transporte a usar.

2.2.1 Comunicação Orientada à Conexão

Protocolos que seguem o paradigma de *Comunicação Orientada à Conexão* como o *TCP* [18][19][20] por norma têm um padrão de comunicação dividido em três secções distintas:

- Abertura de Conexão ou *Handshake*

- Troca de Informação
- Fecho de Conexão

2.2.1.1 Handshake

Inicialmente, e antes de qualquer troca de informação, é realizado um *Three-way Handshake* entre dois Nodos (um par de *Sockets*, onde uma *Socket* é composta por um Endereço e uma porta). Neste *Handshake* um dos Nodos informa o outro que pretende iniciar uma conexão transmitindo uma Mensagem denominada *Synchronize (SYN)*. O segundo responde com uma Mensagem que confirma que recebeu o pedido de início de conexão utilizando *Acknowledge (ACK)* e que pretende continuar com este processo *SYN*. Finalmente, o primeiro confirma que recebeu a Mensagem (*ACK*).

Durante este processo também são estabelecidos alguns parâmetros como o (*Maximum Segment Size (MSS)*) que indica o tamanho máximo que um *Segmento* pode ter durante esta conexão. Este valor pode variar, mas por norma tem um valor máximo de 1500 *Bytes*. Este valor poderia ser significativamente maior pois o valor máximo teórico para um *Segmento* *Transmission Control Protocol (TCP)* é de 64 *KBytes* (65535 *Bytes*). Esta limitação aparece por causa do *Maximum Transfer Unit (MTU)* das Camadas inferiores da *Pilha de Protocolos*, tal como o da *Trama Ethernet*. Desta forma obtemos que o tamanho máximo do *MSS* é igual ao *MTU* do protocolo usado na *Link Layer*.

Uma vez terminado o *Three-way Handshake* e estabelecida a conexão entre as duas *Sockets*, a troca de informação "útil" entre os Nodos pode começar.

2.2.1.2 Troca de Informação

Nesta transmissão, o *TCP* tem como objetivo manter um fluxo constante de dados, reduzir o congestionamento e evitar situações de *Timeout* que possam forçar a abertura de uma nova conexão. Para conseguir atingir estes objetivos foram implementados algoritmos de controlo de fluxo que funcionam com o conceito de *Janelas* e mensagens de confirmação positiva (*ACK*).

Durante o processo de *Handshake* o recetor da informação também indica ao emissor o espaço do *buffer* que tem para a receção da informação (*Janela*). O emissor possui uma *Janela de Congestão* com valores múltiplos de *MSS* que indica quantos *Datagramas* serão enviados num determinado instante. É importante referir que o emissor apenas envia um novo grupo de *Datagramas* de informação quando tiver recebido a confirmação de chegada dos *Datagramas* enviados anteriormente. Dado que as mensagens de confirmação *ACKs* têm uma propriedade acumulativa, a chegada de um *ACK* referente a um *Datagrama* que não o primeiro do grupo também confirma a chegada de todos os *Datagramas* imediatamente atrás desse. Tendo estas 2 propriedades em consideração, podemos afirmar que o *TCP* assegura

que a ordem de chegada dos *Datagramas* referentes a uma mensagem vai ser a mesma com qual foi enviada, sendo que a única troca de ordem que pode acontecer é dentro de cada grupo.

Inicialmente a transmissão de dados começa num estado designado por *Slow Start*. Neste estado a *Janela de Congestão* do Nodo emissor começa com o valor de 1 *MSS* e é incrementada sempre que é recebido um *ACK*, ou seja, cresce exponencialmente. Este crescimento continua até ser atingido um *Threshold*, detetado um *ACK duplicado* pela terceira vez ou ter ocorrido um *Timeout*. Cada uma destas situações vai fazer com que o comportamento da transmissão evolua para um outro estado distinto.

- Caso seja atingido o *Threshold*, o estado passa a ser *Additive Increase / Multiplicative Decrease (AIMD)*;
- Caso seja detetado um *ACK* duplicado pela terceira vez, o estado passa a ser de *Congestion Avoidance* onde a *Janela de Congestão* e o *Threshold* passam a ter o mesmo valor (metade do valor inicial da *Janela de Congestão*) e é aplicado o *AIMD*;
- Caso tenha ocorrido um *Timeout*, o *Threshold* passa a ser igual a metade do valor da *Janela de Congestão* e esta passa a ser igual a 1 *MSS*. Por fim é iniciado o *Slow Start* novamente.

O *AIMD* faz com que a *Janela de Congestão* cresça de forma linear, mas, em caso de falha (*ACK* duplicado pela terceira vez ou *Timeout*), esta é reduzida para metade ou para 1 *MSS* dependendo da situação observada.

2.2.1.3 *Fecho de Conexão*

Quando toda a informação tiver sido transmitida, um Nodo envia uma mensagem de *Finish (FIN)* para informar que aquela conexão vai ser fechada. O segundo confirma a receção do pedido de fecho da conexão (*ACK*) e envia um *FIN*. Por último, o Nodo que iniciou o processo confirma a receção do *Datagrama ACK*.

2.2.1.4 *Cabeçalho*

O *Cabeçalho* do *TCP* inclui vários parâmetros que são fundamentais para o funcionamento do protocolo. Tais parâmetros são:

- Porta de Origem (16 *bits*)
- Porta de Destino (16 *bits*)
- Número de Sequência (32 *bits*)

- Número de *ACK* (32 bits)
- *Data Offset* (4 bits)
- Espaço Reservado (6 bits)
- Bits de Controlo (6 bits)
- Janela (16 bits)
- *Checksum* (16 bits)
- Apontador de Urgência (16 bits)
- Opções (Variável)
- *Padding* (Variável)

Tendo em conta todos estes parâmetros, o *Cabeçalho* do *TCP* pode variar em tamanho desde os 20 Bytes, quando não tem quaisquer Opções, até os 60 Bytes quando as tem.

Para sabermos o tamanho do *Payload* de um *Datagrama TCP* temos que tomar em conta o tamanho máximo do *Datagrama* e o tamanho do *Cabeçalho*. Sabendo que o tamanho máximo do *Datagrama* é igual ao *MTU* do protocolo usado na *Transport Layer* (1500 Bytes no caso da *Trama Ethernet*) e que o *Cabeçalho* pode variar entre os 20 e os 60 Bytes, podemos afirmar que o tamanho máximo do *Payload* está entre os 1480 e os 1440 Bytes.

2.2.1.5 Vantagens

Com estas características, o *TCP* trás algumas vantagens como:

- Controlo de Fluxo que lida com a perda de *Datagramas* por *Drop* e com a necessidade de recuperar *Datagramas* perdidos;
- Controlo de Erros que garante a integridade da mensagem e que esta foi transmitida de forma correta, iniciando um processo de recuperação quando possível;
- Controlo de Congestão que tenta manter a conexão viva em momentos de sobrecarga na Rede, sem criar mais sobrecarga;
- Um programa pode ter múltiplas *Sockets* a "ouvir" o mesmo par *IP:Porta* na mesma máquina sem causar qualquer erro pois uma conexão é definida por um par de *Sockets* e a *Socket* emissora pode estar alojada em qualquer Porta não ocupada anteriormente. Desta forma é possível redirecionar tráfego para diferentes *Sockets* que estão a "ouvir" o mesmo par *IP:Porta*

Isto faz com que seja uma ferramenta útil e que traga garantias à transmissão de informação entre Nodos da Rede.

2.2.1.6 Desvantagens

Devido a todas as garantias que o *TCP* dá existem algumas limitações que aparecem por defeito que não são desejáveis numa rede caracterizada pela sua instabilidade, perda elevada de *Datagramas*, entrada e saída constante de Nodos, mobilidade e não garantia de presença dos mesmos.

Tudo isto iria levar a que uma conexão estivesse constantemente a tentar recuperar de perdas de *Datagramas*, a encontrar situações de *Timeout* e ter que abrir novas conexões. Por *design* o processo de troca de informação estaria sempre num estado de *Slow Start* devido a todos os erros provenientes das características da rede.

2.2.2 Comunicação não Orientada à Conexão

Como o próprio nome indica, este paradigma não estabelece qualquer Conexão para transmitir ou receber Informação. No caso do Protocolo *User Datagram Protocol (UDP)* [21][22] apenas existe o conceito de enviar *Datagramas* para uma *Socket (IP:Porta)*, para um grupo *Multicast* ou *Broadcast*.

2.2.2.1 Mecanismos presentes no UDP

O Protocolo *UDP* não apresenta qualquer mecanismo de recuperação de erros ocorridos durante a transmissão de dados, de confirmação de receção de *Datagramas* ou de Controlo de fluxo e Congestão como o *TCP*, o que faz com que estas funcionalidades tenham que ser implementadas ao nível da aplicação se forem pretendidas.

O *UDP* apenas apresenta um mecanismo, o *Checksum*, e o seu uso é facultativo. Este serve para verificar a integridade da informação contida no *Cabeçalho* e *Payload* de um *Datagrama* na *Socket* destino.

2.2.2.2 Troca de Informação

Devido à falta de mecanismos como o de Recuperação de Erros, Controlo de Fluxo e Controlo de Congestão do *TCP*, a troca de informação através de *UDP* torna-se bastante simples, e o facto deste não garantir que o *Datagrama* seja entregue faz com que o processo de envio não tenha muitos requisitos. Este apenas necessita de uma *Socket (IP:Porta)*, informação para ser enviada e um par *IP:Porta* que identifique um destino. O facto mais contraintuitivo é que o par *IP:Porta* destino pode não estar associado a uma máquina terminal, e como não existem quaisquer mensagens de confirmação de receção (como um *ACK*) nem mensagens de erro por este par não estar associado, é possível criar "*grupos virtuais*" com pares *IP:Porta* especiais. A estes "*grupos virtuais*" dá-se o nome de *Multicast groups* e *BroadCast*.

Para a receção de informação apenas é necessário uma *Socket* (par *IP:Porta* que não esteja em uso) e um *buffer* capaz de receber um *Datagrama*.

2.2.2.3 *Multicast e Broadcast*

Como já foi referido, os grupos *Multicast* e *Broadcast* são grupos definidos por um par *IP:Porta* especiais definidos pela *Internet Assigned Numbers Authority (IANA)* [23][24] que servem para transmitir um *Datagrama* a um grupo de Nodos presentes na mesma rede sem ser necessário que os Nodos se conheçam. Para interagir com um grupo *Broadcast* é necessário utilizar o endereço *IPv4* 255.255.255.255, sendo que tal não é possível em *IPv6*. Qualquer Nodo de uma rede pode emitir ou receber informação de um grupo *Multicast* ou *Broadcast* sem quaisquer restrições, sendo que não é possível restringir que Nodos de um determinado grupo *Multicast* ou *Broadcast* recebem um *Datagrama* emitido.

É importante referir que um Nodo não necessita de entrar num grupo *Multicast* ou *Broadcast* para emitir um *Datagrama* para os mesmos, apenas é necessário entrar nestes grupos para receber os *Datagramas* que para eles são emitidos.

2.2.2.4 *Cabeçalho*

Visto que o *UDP* não trás garantias de entrega fiável com o seu uso, o cabeçalho pode ser reduzido a apenas 4 campos:

- Porta Origem
- Porta Destino
- Comprimento do *Datagrama*
- *Checksum*

Sendo que cada campo ocupa 2 *Bytes*, o cabeçalho ocupa no total apenas 8 *Bytes* por *Datagrama*. Tal como os *Datagramas TCP*, os *Datagramas UDP* têm, por norma, 1500 *Bytes* de tamanho, com um tamanho máximo teórico de 64 *KBytes* (65535 *Bytes*), sendo que são limitados pelo *MTU* da *Trama Ethernet*, mas o seu *Cabeçalho* tem um tamanho fixo de 8 *Bytes* fazendo com que o *Payload* tenha uma capacidade de 1492 *Bytes*.

2.2.2.5 *Vantagens*

Devido à falta de mecanismos de controlo, o *UDP* apresenta uma grande simplicidade e oferece flexibilidade para a sua adaptação a vários casos de uso, sendo que se tais mecanismos forem necessários estes podem ser implementados ao nível da aplicação, dando mais controlo sobre que informação é transmitida para a rede e quando esta é transmitida, bem como que mecanismos serão implementados. Esta falta de mecanismos leva ainda a

um cabeçalho de tamanho reduzido, o que possibilita a redução do *overhead* e o aumento do *Payload* por *Datagrama*.

Como se trata de um *Protocolo Orientado ao Datagrama* e não à *Conexão*, não é necessário estabelecer, terminar nem manter estados destas. Isto elimina a transmissão de *Datagramas* que poderiam ser perdidos e que por sua vez levariam a um estado de recuperação de erro.

Com o uso de *UDP* surge a possibilidade de uso de grupos *Multicast* e *Broadcast* para a transmissão de dados para múltiplos *Nodos* simultaneamente sem que seja necessário conhecer que *Nodos* estão nesses grupos nem que quantidade.

2.2.2.6 Desvantagens

O facto de não existirem garantias com o uso de *UDP* e se estas forem necessárias para o bom funcionamento de uma aplicação, faz com que sejam perdidos mais tempo e recursos na implementação de procedimentos para assegurar tais garantias. Como se tratam de implementações feitas à medida, podem não ter a mesma qualidade nem segurança que outros procedimentos *Standard*.

O funcionamento do *UDP* faz com que apenas uma *Socket* possa estar associada a cada par *IP:Porta*, sendo necessário controlar que pares já estão a ser usados.

Se for pretendido usar um grupo *Multicast* é necessário verificar que endereços estão livres no *website* da *IANA* e possivelmente reservar os mesmos.

2.2.3 Novos desafios e avanços recentes

Ao longo dos últimos anos, com o aumento exponencial do número de terminais móveis ligados à Internet, os novos avanços em tecnologias de comunicação móvel e fixa, e as novas aplicações multimédia, levaram a comunidade Internet a investigar novos protocolos de transporte [17].

Consequentemente, surgiram novas propostas de protocolos de transporte, seguindo basicamente três linhas distintas de investigação:

- evolução dos algoritmos de controlo de congestão

Os mecanismos de controlo de congestão tradicionais não são adequados aos novos cenários (ex: ligações sem fios), levando à pesquisa de novos algoritmos e novas estratégias.

- introdução de abordagens *multipath*

Muitos dispositivos possuem mais que uma interface de rede e mesmo mais que uma ligação à rede, com caminhos alternativos que podem ser usados em simultâneo para melhorar o desempenho da camada de transporte.

- alternativas implementados fora do sistema operativo

Soluções fora do kernel, entre a camada de aplicação e a camada de transporte tradicional.

Um exemplo de alternativa implementada fora do Kernel é o QUIC [25], introduzido pela Google em 2013 e que muito recentemente foi normalizado pelo [Internet Engineering Task Force \(IETF\)](#) no RFC9000. Faz uso do [UDP](#) como protocolo de suporte e é disponibilizado na forma de biblioteca de funções ou integrado no código das aplicações. Tem como objetivo principal incorporar algumas das características do [TCP](#), como as confirmações, recuperação de erros e controlo de congestão. Pretende além disso multiplexar múltiplas transferências em *streams* distintas, evitando que uma mais pesada possa bloquear as outras.

TRANSFERÊNCIA DE DADOS TOLERANTE A FALHAS

Neste Terceiro Capítulo será Desenhado um Protocolo de Transferência de Dados que tenciona permitir a troca de informação entre Nodos que se encontrem em ambientes não propícios a tal Ação. Inicialmente é especificada uma primeira iteração do funcionamento geral desse Protocolo, sendo que após cada especificação serão apontados os problemas que existem nessa especificação e que decisões e modificações serão tomadas para que a iteração seguinte mitigue tais problemas. Este processo iterativo de mitigação de problemas começa num nível mais técnico, mas com o avançar do desenvolvimento, são introduzidos conceitos mais abstratos e que podem não ser tão óbvios quanto os primeiros. Após a especificação da última iteração, são abordados tópicos como a medição de Estatísticas para o auxílio de controlo de Fluxo de Informação, tipo de Endereços a utilizar, Segurança e Detecção e Correção de Erros.

3.1 ARQUITETURA DO PROTOCOLO DE TRANSFERÊNCIA DE DADOS

Como os ambientes onde estas arquiteturas são destinadas a operar são propícios a movimentações dos Nodos e, por isso, à existência de momentos onde não existe qualquer ligação entre os Nodos *Origem* e *Destino* de uma Transmissão, o Protocolo de Transferência de Dados a ser desenvolvido tem que ser capaz de lidar com a sua interrupção a qualquer instante. Para além disto, é normal a existência de perda de *Datagramas* (ou *Drop* de *Datagramas*) que também tem que ser tomada em consideração. Estes *Drops* podem ocorrer em 3 principais cenários:

- *Drop* na Origem
- *Drop* na Rede
- *Drop* no Destino

Destes 3 cenários, 2 ocorrem devido a um excesso de *Datagramas* enviados ou recebidos num curto período de tempo (*Drop* na Origem e *Drop* no Destino respetivamente). Isto

acontece, pois quando um *Datagrama* é enviado, este inicialmente é colocado num *Buffer* de tamanho predeterminado fixo que está associado a uma *Datagram Socket* que o transmite. No processo de recepção de um *Datagrama* acontece o mesmo, sendo que à chegada este é colocado num *Buffer* de tamanho predeterminado e fixo, associado à *DatagramSocket* que o recebe. Visto que estes *Buffers* têm um tamanho fixo, se o fluxo de *Datagramas* for demasiado elevado, estes acabam por ficar completamente cheios e, a partir desse instante, cada *Datagrama* que necessite de ser colocado nestes vai ser rejeitado, resultando num *Drop*. Em contraste com *Drops* associados à Rede, estes conseguem ser controlados com um mecanismo de controlo de Fluxo na Transmissão de Dados.

Como a Transmissão de Dados é feita numa Rede de comutação de Pacotes, os Dados a serem transmitidos poderão necessitar de ser fragmentados para que não seja ultrapassado o (*Maximum Transfer Unit (MTU)*) do Protocolo da Camada *Data Link* a ser usado e não arriscar a sua fragmentação a nível da Camada de Protocolos da Rede. Sendo que o Protocolo mais comum nesta camada é o Protocolo *Ethernet*, será seguro assumir que o *MTU* será de 1500 *bytes* no pior caso e, com esta assunção, é possível fragmentar a Informação a Transmitir *a priori*. A cada Fragmento criado deverá ser-lhe associado um Identificador, por exemplo um número inteiro, que indique a sua posição relativa aos restantes Fragmentos da Informação original. A esta estrutura de dados que contém um Fragmento da Informação e um Identificador denominarei de *Chunk*. Todos os *Chunks* relativos a um conjunto de Dados deverão de possuir a mesma quantidade de Informação com a exceção do último que poderá ser menor (contido entre 1 bit e o tamanho máximo de um *Chunk* previamente estabelecido).

Tendo em consideração os ambientes em que este Protocolo será utilizado, bem como a limitação do tamanho de um *Datagrama* devido ao *MTU*, é de extrema importância que se consiga maximizar o uso de cada *Datagrama* enviado. Desta forma, todos os *Datagramas* enviados têm que ser capazes de desencadear uma ação quando recebidos e não ficarem dependentes da chegada de outros *Datagramas* ao Nodo Destino. Assim podemos dizer que o foco deste Protocolo é maximizar a utilidade de cada *Datagrama* recebido e minimizar o número de *Datagramas* descartados por serem dependentes de informação presente noutro *Datagrama* que poderá não chegar ao Nodo Destino.

Com isto em mente é possível começar o desenho do Protocolo de Transmissão e Dados.

3.1.1 Processo de Transmissão

Numa Transmissão de uma Mensagem existem 2 atores, o Transmissor e o Recetor, onde o Transmissor possui Informação que pretende partilhar com o Recetor. Inicialmente é necessário que haja uma troca de Meta Informação relativamente à Transmissão em si e aos Dados que irão ser transmitidos nesta Transmissão. Após esta troca de Meta Informação pode ser iniciada a troca de Dados, juntamente com as mensagens de confirmação de chegada dos

Datagramas. Finalmente, quando o Recetor possuir todos os Dados que lhe eram destinados nesta Transmissão, pode enviar uma mensagem que o confirme ao Transmissor.

Daqui podemos afirmar que existem 3 fases distintas numa troca de Dados entre 2 Nodos em que é possível focar a atenção individualmente (Figura 1).

- Uma fase inicial de troca de Meta Informação onde é comunicada a intenção de transmissão de Dados bem como todas as informações necessárias para que essa troca seja realizada;
- Uma fase onde a troca de Dados é efetivamente realizada, juntamente com a confirmação de chegada de tais Dados;
- Uma fase final que serve para concluir este processo de troca de Dados e permitir que os recursos alocados para a realização desta Transferência sejam libertos.

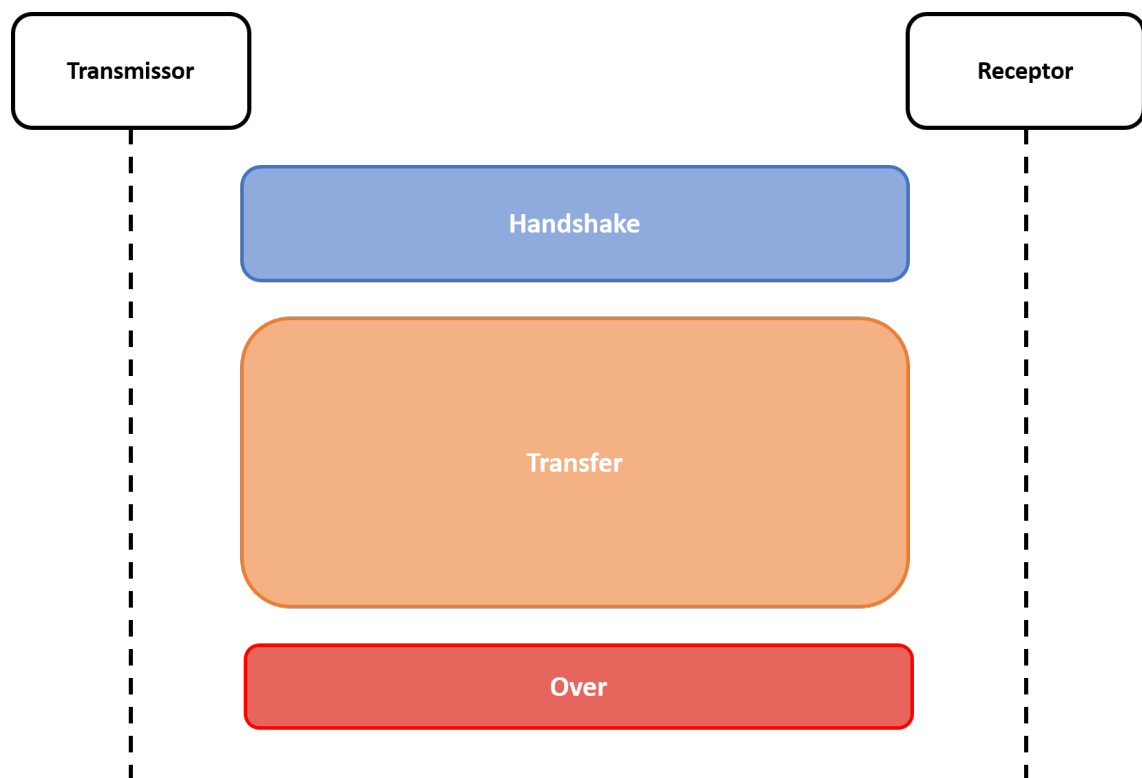


Figura 1: Estrutura geral do Processo de Transferência de Dados entre um Transmissor e um Recetor

3.1.2 Troca de Meta Informação (*Handshake*)

Esta primeira fase da Transferência, ilustrada na Figura 2, tem como objetivo dar início à mesma. Neste momento do Processo de Transferência de Dados apenas se sabe que o

Transmissor possui os Dados a transmitir, a Meta Informação de tais dados e um par *Endereço, Porta* que indica o destino da informação. O Recetor apenas possui uma *DatagramSocket* que está a escutar o par *Endereço, Porta* referido anteriormente.

Para que a Transferência possa ser realizada, o Transmissor deve de criar um *Datagrama* que contenha um *ID* que identifique a Transferência, um segundo *ID* que identifique unicamente o Nodo Transmissor, que seja independente de qualquer informação da Rede a que está conectado e que consiga sobreviver a vários reinícios do Dispositivo. Juntamente com estes 2 Identificadores, deve acrescentar ao *Datagrama* toda a Meta Informação Relativa aos Dados a serem transmitidos. Tal Meta Informação deverá de conter no mínimo:

- Número total de fragmentos (*Chunks*)
- Tamanho máximo que um Fragmento pode tomar em *Bytes*
- *Hash* Identificadora dos Dados
- Algoritmo usado para obter a *Hash* Identificadora dos Dados

Uma vez construído este *Datagrama*, denominado *Transfer Meta Info (TMI)*, este deverá de ser enviado para o par *Endereço, Porta* destino referente ao Recetor.

Quando o Recetor recebe um *Datagrama TMI* este deverá de registar a intenção de partilha daquela informação por parte do Transmissor e construir o seu *Datagrama* de resposta, denominado *Transfer Receiver Info (TRI)*. Este *Datagrama* deverá conter o mesmo Identificador de Transferência presente no *TMI* recebido, um limite de velocidade máxima de emissão de *Datagramas* por segundo, e uma estrutura que represente os *Chunks* que possui ou que não possui. Por questões de simplicidade, se este não possuir qualquer *Chunk*, este campo poderá conter o valor *null* o que indicará ao Transmissor que todos os *Chunks* relativos aos Dados em questão terão que ser transmitidos.

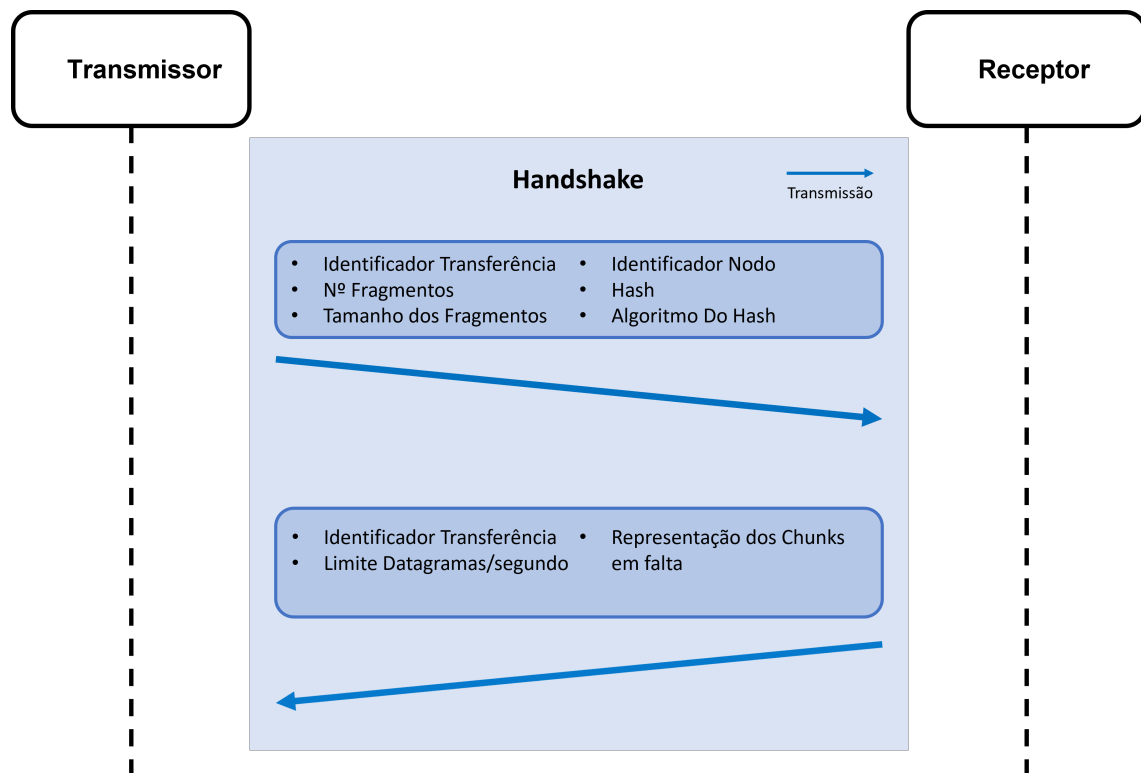


Figura 2: Especificação da primeira Iteração do Bloco Handshake do Processo de Transferência

3.1.3 Transmissão de Dados (*Transfer*)

Uma vez concluído o *Handshake*, o Transmissor pode começar a transmissão dos Dados (Figura 3). Para tal terá que construir os *Datagramas* que possuem os *Chunks* relativos aos Dados a serem enviados. Tais *Datagramas* denominam-se de *Chunk Message (CM)*. Para além de conter um *Chunk*, este terá que conter o mesmo Identificador de Transferência usado nos *Datagramas TRI* e *TRI* anteriores. Após a construção de todas as *CM*, o Transmissor enviará as mesmas para o Recetor e ficará à escuta de *Datagramas* que identifiquem que *CM* necessitarão de ser retransmitidos ou de uma Notificação que indique que a Transmissão poderá terminar. As *CM* deverão de ser enviadas para o par *Endereço, Porta* anteriormente usados no envio do *Datagrama TRI*.

O Recetor, após ter concluído o processo de *Handshake* apenas necessitará de escutar o seu par *Endereço, Porta* para receber as *CM* referentes à Meta Informação que recebeu anteriormente. Com esta Meta Informação, as *CM* recebidas e um sistema de *Timeouts*, o Recetor deverá de ser capaz de informar o Transmissor que *Datagramas CM* necessita para a conclusão da Transmissão.

Este ciclo de envio de *CM* por parte do Transmissor, *Timeout* no Recetor, Recetor informa Transmissor que *Chunks* estão em falta, apenas termina quando o Recetor obtiver todos os

Chunks referentes à Meta Informação recebida no *Handshake*. Quando o Recetor possuir todos os Dados, deverá de prosseguir para a próxima fase.

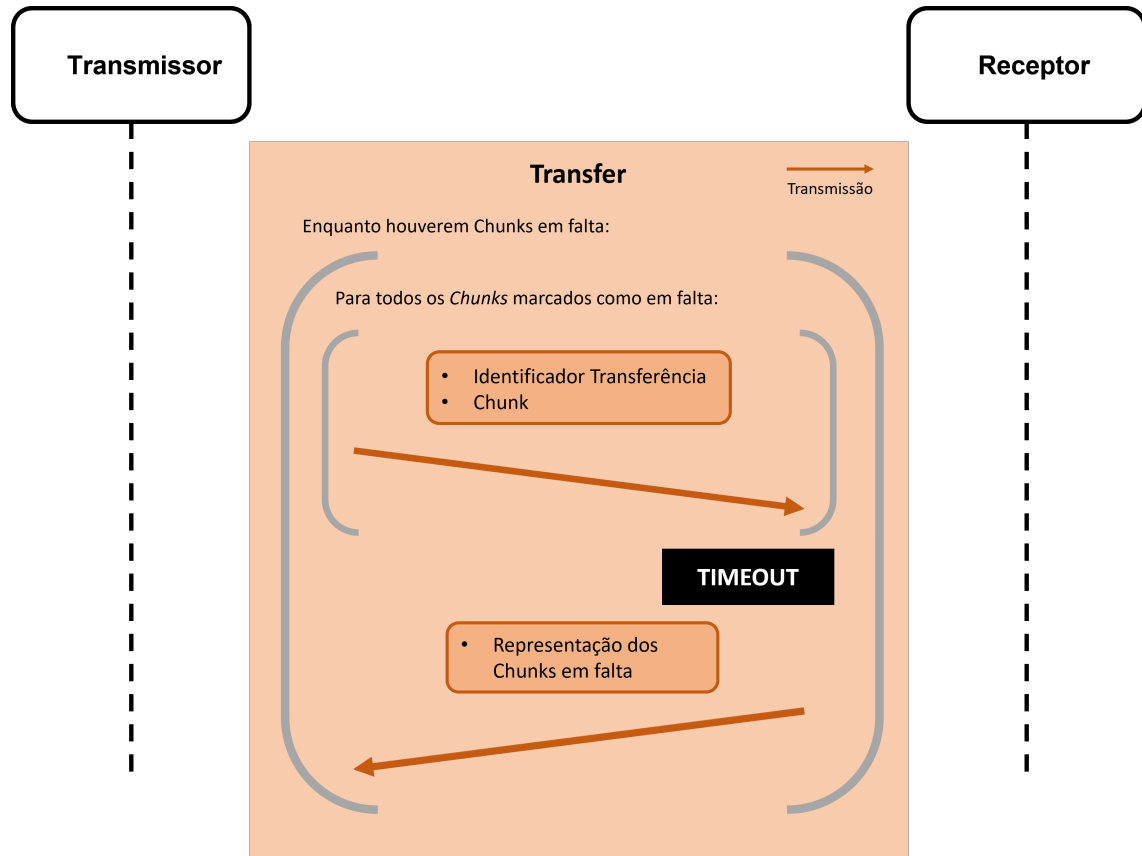


Figura 3: Especificação da primeira Iteração do Bloco Transfer do Processo de Transferência

3.1.4 Finalização da Transferência (*Over*)

Esta fase apenas tem como objetivo indicar que a Transferência de Dados foi concluída e que os recursos alocados podem ser libertos (Figura 4). Desta forma o Recetor deverá construir um *Datagrama* que possua o Identificador da Transferência usado nos *Datagramas* anteriores denominado de *Over*.

O *Transmissor*, ao receber este *Datagrama*, poderá terminar o processo de Transferência e libertar todos os recursos que dedicou à mesma.

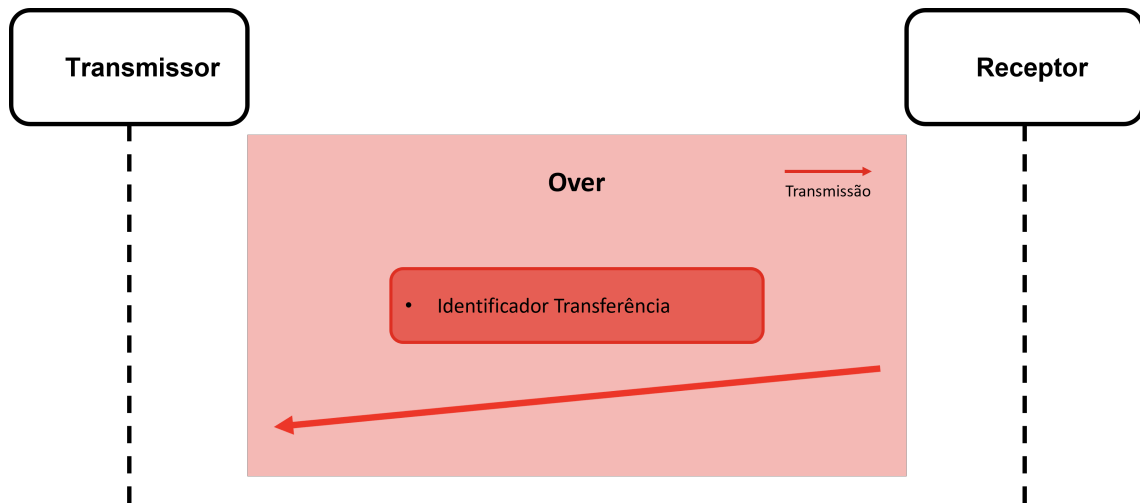


Figura 4: Especificação da primeira iteração do Bloco Over do Processo de Transferência

3.1.5 Falhas no Protocolo de Transferência de Dados

Nesta primeira iteração do Protocolo de Transferência existem pontos que podem ser melhorados e outros que necessitam de mais atenção e reestruturação. Estes pontos são:

- Indicação de *Chunks* em Falta
- *Drops* na Origem
- *Drops* no Destino
- Estrutura do Bloco *Handshake*
- Estrutura do Bloco *Transfer*
- Utilidade do Bloco *Over*
- Resiliência a movimentações de Nodos na Rede
- *Network Address Translation*

3.1.5.1 Indicação de *Chunks* em falta

Até este momento não foi referida qualquer Estrutura para a Indicação de *Chunks* em falta pois este ponto pode ser realizado de múltiplas formas, sendo que a Estrutura escolhida pode ter um grande impacto no funcionamento do Protocolo num todo. Anteriormente apenas foi referido que um *Recetor* que não possui qualquer *Chunk* pode enviar esta Estrutura com o valor *null* para o indicar ao *Transmissor*. Os problemas começam a aparecer quando o *Recetor* já possui alguns *Chunks*, obtidos por exemplo numa Transferência realizada anteriormente

que foi terminada de forma antecipada ou inesperada. Isto pode levar a que o conteúdo da Estrutura ultrapasse o *MTU* definido para os *Datagramas*, levando à fragmentação ao nível da Camada de *Data Link*, que já foi referido que deve de ser evitado.

Assim sendo é necessário definir uma Estrutura que consiga conter a informação dos *Chunks* em falta, que respeite o *MTU*, que siga a filosofia de que cada *Datagrama* tem que desencadear uma Ação no Nodo que o recebe e que seja independente da chegada de outros *Datagramas*.

Primeiramente é necessário estabelecer que informação será colocada nesta Estrutura e consequentemente enviada em *Datagramas*. Como já foi analisado, no Protocolo *TCP* é enviado um *Datagrama ACK* por *Datagrama* de Dados recebidos ou um *Datagrama ACK* que age como um *ACK* cumulativo que confirma a chegada de uma série de *Datagramas* que façam parte de uma sequência seguida de *Datagramas*. Este *ACK* cumulativo agiliza bastante o processo de confirmação de chegada de *Datagramas* pois elimina a necessidade de envio de outros *ACKs*, mas este mecanismo apenas funciona pois o *TCP* garante que a ordem de entrega de *Datagramas* segue a ordem original da informação com uma margem de erro de um grupo de *Datagramas* do tamanho da *Janela* de transmissão. Como é possível observar na Figura 3, no Bloco *Transfer* são enviados todos os *Datagramas* em Falta sem a garantia de ordem e, apenas após um *Timeout*, é que o Recetor informa o Transmissor quais são os *Chunks* que continuam em falta. Este pormenor força a que seja necessário abordar este problema de outra forma.

Numa primeira iteração desta Estrutura seria possível aproveitar o facto de cada *Chunk* já possuir um Identificador que o relaciona de forma espacial com o resto dos *Chunks* dos Dados transmitidos. Esta abordagem resolveria o problema, mas causaria outro. A intenção inicial era que apenas seria enviado um destes *Datagramas* por ciclo do Bloco *Transfer* após o *Timeout*, mas com algumas contas é possível perceber que esta abordagem ultrapassaria rapidamente o *MTU* de um *Datagrama* de um dispositivo que use o Protocolo *Ethernet*. Assumindo que tais Identificadores seriam do tipo *Integer*, cada um ocuparia 4 *Bytes* e, assumindo que o tamanho de um *Datagrama* será de 1500 *Bytes*, só seria possível transmitir 375 Identificadores por *Datagrama* no melhor caso possível. Mesmo usando alguma técnica para a compressão destes Identificadores, será necessário modificar o funcionamento dos Blocos onde é necessária a transmissão desta Estrutura para se poder enviar múltiplos destes *Datagramas*. Isto acontece pois não é estabelecido um tamanho máximo para os Dados a serem enviados, fazendo com que seja possível ser necessário transmitir um número arbitrário de Identificadores.

Assim sendo, a Estrutura dos blocos *Handshake* e *Transfer* necessitam de ser reformuladas para acomodar esta mudança (Figuras 5 e 6). Em ambos os Blocos será necessário permitir o envio e receção de múltiplos destes *Datagramas*, sendo que, no bloco *Handshake*, no *Datagrama TRI*, serão colocados o maior número possível de Identificadores sem ultrapassar o *MTU* da

máquina atual e, se necessário, serão enviados mais Identificadores nestes *Datagramas* que apenas contêm Identificadores.

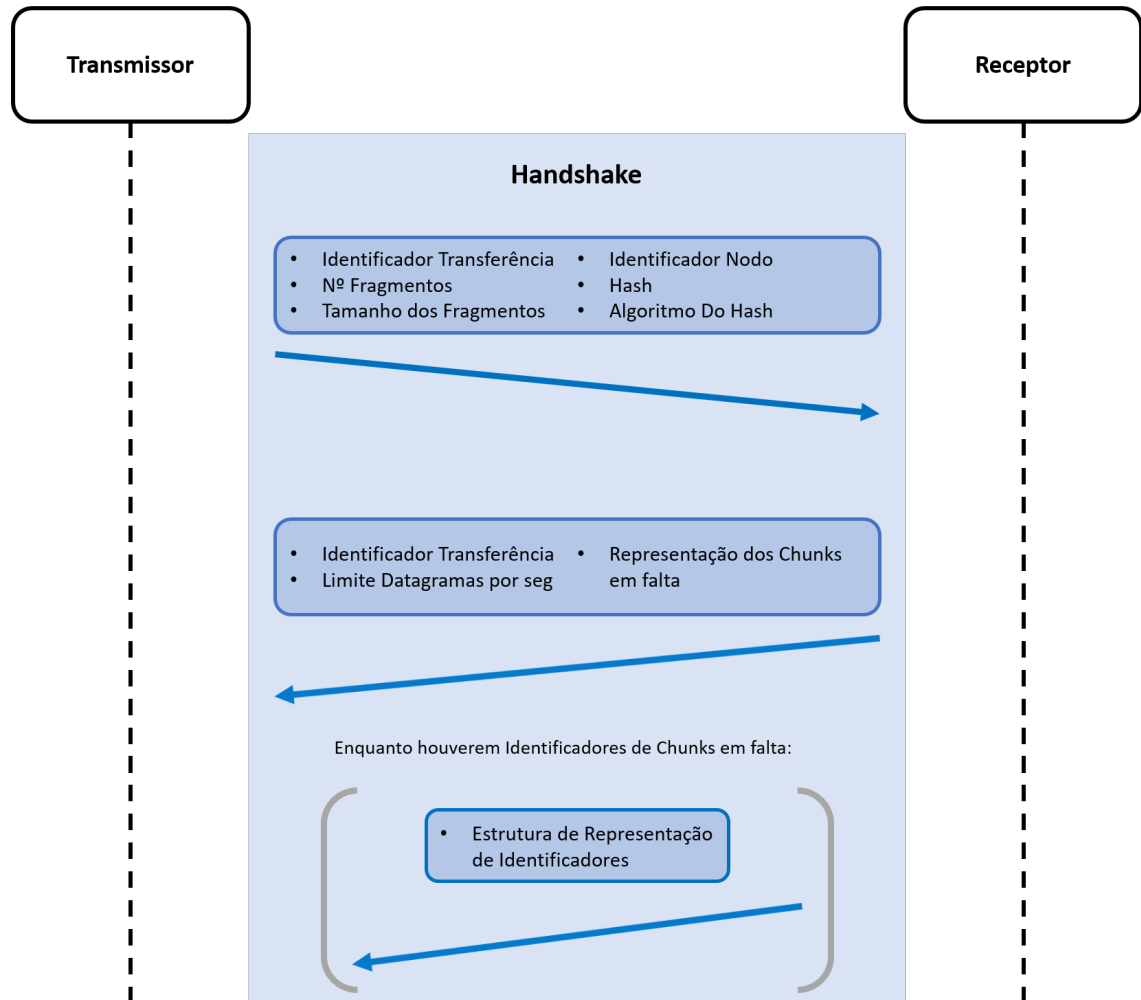


Figura 5: Especificação da segunda Iteração do Bloco Handshake do Processo de Transferência

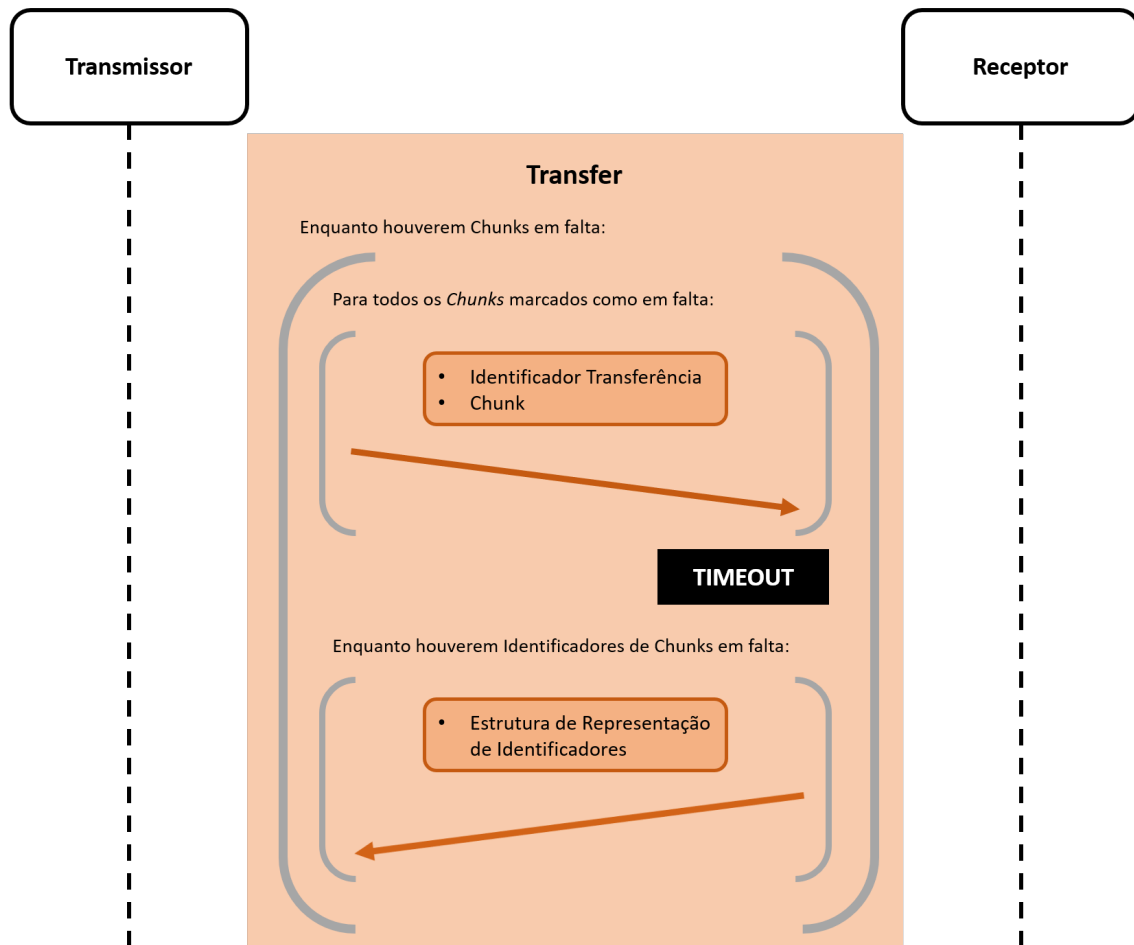


Figura 6: Especificação da segunda Iteração do Bloco Transfer do Processo de Transferência

3.1.5.2 Compressão dos Identificadores

De forma a tentar comprimir a Informação destes *Datagramas*, será utilizado o facto dos Identificadores apresentarem uma grande proximidade entre si de forma a agrupá-los e reduzir a informação redundante. Em cada Estrutura estarão presentes 3 tipos de dados:

- Um Inteiro que corresponde ao Identificador com o valor mais *baixo* presente no Datagrama que será usado como referência (*ReferenceID*)
- Um *Bit Array* que tem como objetivo indicar a presença de um Identificador relativamente ao Identificador imediatamente *inferior* dentro de um determinado intervalo de valores (*IDPresence*)
- Um *Byte Array* que tem como objetivo indicar a diferença final entre o próximo Identificador e a soma do último Identificador com múltiplos do intervalo de valores (*FinalIncrements*)

Com estas 3 subestruturas é possível reduzir significativamente o tamanho ocupado por um determinado número de Identificadores aproveitando a sua proximidade (Figura 7). É importante referir que estes dados (*Inteiros* e *Bytes*) deverão ser do tipo *Unsigned* ou deverão tomar o seu valor mais baixo como valor base de forma a aproveitar o alcance dos mesmos. Desta forma será possível representar 4 294 967 296 Identificadores com os *Inteiros* enquanto os *Bytes* conseguem representar intervalos de 255 Identificadores.

ReferenceID	1285					
IDPresence	0	0	1	0	1	1
FinalIncrements	10	55	1			

Figura 7: Exemplo prático da Estrutura usada para a compressão dos Identificadores {1285, 1805, 2115, 2116}

Com esta informação é simples reconstruir os Identificadores. O primeiro encontra-se no *ReferenceID*, e os restantes conseguem ser obtidos a partir de somas tendo em conta os Arrays *IDPresence* e *FinalIncrements*. Para obter um novo Identificador a partir do *ReferenceID*, inicializa-se uma variável *CurrentID* com o valor do *ReferenceID* e percorre-se o Array *IDPresence*. Conforme o valor presente no Array *IDPresence* é tomada uma das seguintes ações:

- se *Bit* igual a 0, adicionar 255 ao *CurrentID* e prosseguir;
- se *Bit* igual a 1, percorrer o Array *FinalIncrements* e adicionar o valor presente nesse Array ao *CurrentID*. Guardar esse Identificador e prosseguir.

Desta forma tem-se que:

$$ID_0 = ReferenceID$$

$$ID_1 = ID_0 + (n * 255) + FI[0]$$

$$ID_2 = ID_1 + (m * 255) + FI[1]$$

...

Onde *FI* é o Array *FinalIncrements*, $0 \leq n, 0 \leq m$.

Com esta representação, cada Identificador para além do *ReferenceID* usa $nBits + 1Byte$ em vez de 4 *Bytes*, sendo que a Versão comprimida apenas se torna maior que a Versão original quando:

$$nBits + 1Byte > 4Bytes$$

$$nBits + 8Bits > 32Bits$$

$$nBits > 24Bits$$

Assim sendo, apenas se ocupa mais espaço que a forma original quando um ID_n e um ID_{n+1} do mesmo *Datagrama* se encontram a pelo menos $(24 * 255) + 1 = 6121$ Identificadores de distância. Se cada *Chunk* tiver um tamanho de 1400 *Bytes* significa que esta perda só é possível quando os Dados a serem enviados têm mais que $(6120 * 1400) / (1024^2) \sim 8,17MB$, e que esses 2 *Chunks* perdidos estejam separados por esse intervalo de 8,17 MB.

Outra problema que pode ser observado é quando o *PresenceID* termina com k *Bits* a 0, com $k > 0$. Isto pode acontecer em 2 situações distintas:

- Dados 2 Identificadores ID_i e ID_{i+1} , com ID_i já representado na Estrutura, $ID_{i+1} - ID_i > 255$ e com o *Datagrama* perto do limite do seu *MTU*
- Dados 2 Identificadores ID_j e ID_{j+1} , com ID_j já representado na Estrutura, $ID_{j+1} - ID_j >>> 255$ e com o *Datagrama* quase vazio.

Dada a forma como esta compressão foi desenhada, este k pode estar contido entre $[0, MTU*8 - 8]$ e a única característica que o impede de crescer descontroladamente é a proximidade entre os Identificadores.

A primeira perda não é muito prejudicial, pois a Estrutura já se encontra cheia de Identificadores e esta é de apenas alguns *Bits*. Este Identificador ID_{i+1} que não conseguiu ser representado neste *Datagrama* será o *ReferenceID* do próximo *Datagrama* e não será perdido.

A segunda perda é um problema bastante mais prejudicial, pois, no pior caso, ID_j e $ID_j + 1$ estão tão distantes que ID_j é o *ReferenceID* e o *Datagrama* apenas é constituído pelo *Array IDPresence* cheio de *Bits* com o valor 0. Esta perda está relacionada com todas as outras perdas desta Estrutura de compressão, mas apenas pode acontecer quando existe uma perda de 2 *Chunks* de uma Transferência que se encontrem distantes e sem outras perdas entre eles. Por isso, pode-se dizer que se trata de uma perda que vai acontecer raramente.

Mesmo com estas falhas, este algoritmo de compressão de Identificadores adequa-se ao uso que lhe é dado, porém estas perdas podem ser mitigadas se se tratarem os Identificadores a representar nesta Estrutura como grupos de Identificadores, como se encontra ilustrado na Figura 8, onde se cria um novo grupo sempre que se inicializa uma Estrutura principal ou quando se encontram 2 Identificadores que causariam uma das perdas descritas anteriormente. Cada grupo continuará a ser representado pelos 3 mesmos parâmetros (*ReferenceID*, *IDPresence*, *FinalIncrements*), mas a Estrutura principal passará a ser representada por *Arrays* destes 3 parâmetros, onde cada entrada nestes *Arrays* representará um grupo de Identificadores. Esta mudança eliminará todas as perdas referenciadas anteriormente, mas em contraste será introduzido um maior *Overhead* para possibilitar a utilização de Estruturas mais complicadas (*Arrays de Arrays*).

ReferenceIDs	RID0			RID1		
IDPresences	b0	b1	b2	b0	b1	b2
FinalIncrements	inc	inc	inc	inc	inc	inc

Figura 8: Exemplo da Estrutura usada para a compressão dos Identificadores que segue a abordagem de Grupos de Identificadores

3.1.5.3 Conteúdo da Estrutura que armazena Identificadores

Com a Estrutura dos Blocos *Handshake* e *Transfer* definidos e a Estrutura de representação dos Identificadores estabelecida, é possível começar a analisar que Identificadores devem de ser enviados ao longo do processo de Transferência dos Dados. Existem 2 possibilidades que valem a pena analisar:

- Confirmação Positiva
- Confirmação Negativa

Numa situação de Confirmação Positiva, o *Recetor* iria informar o *Transmissor* quais foram os *Chunks* que conseguiu receber e, numa situação de Confirmação Negativa, seria feito o contrário.

A Confirmação Positiva segue uma filosofia semelhante à do protocolo *TCP*, mas que neste contexto trás alguns problemas. Se considerarmos que o Bloco *Transfer* é executado em ciclos onde inicialmente são enviados **todos** os *Chunks* em falta e, em resposta, são transmitidos Identificadores de forma a indicar ao *Transmissor* que *Chunks* se perderam e necessitam de ser retransmitidos, é possível identificar os seguintes problemas:

- Perda de *Datagramas* de Identificadores leva à incerteza de quais *Chunks* realmente se perderam, podendo levar a:
 - Necessidade de envio de Mensagens de confirmação de chegada de tais *Datagramas*, como *ACKs*, para eliminação de incerteza que por sua vez levariam a um maior atraso devido a potenciais retransmissões;
 - Funcionamento sem Mensagens de confirmação de chegada, fazendo com que certos *Chunks* sejam dados como "perdidos" indevidamente, levando à sua retransmissão de forma desnecessária.

Para além disto, para que a informação fosse verdadeiramente a correta, seria necessário que todos os *Datagramas* de Identificadores chegassem ao seu destino, o que faz com que exista uma dependência entre *Datagramas* e, por isso, as ações tomadas pelo *Transmissor* poderiam não ser corretas, quebrando uma das regras que se impôs no início deste Projeto.

A Confirmação Negativa, por outro lado, visa informar o *Transmissor* que *Chunks* estão efetivamente em falta e, por consequência, faz com que cada *Datagrama* seja independente dos outros, eliminando a necessidade de Mensagens de confirmação de chegada para eliminar incerteza e a retransmissão de *Chunks* já recebidos pelo *Recetor*. Porém esta abordagem apresenta uma desvantagem que é a necessidade de um longo *Timeout* para garantir que a "Janela" de transmissão terminou. No entanto isto pode ser atenuado e será explicado na secção 3.1.5.5.

Em termos de Tráfego na rede, a Confirmação Positiva pode ter 2 comportamentos dependendo da solução que é implementada:

- Se não se enviar qualquer Mensagem de confirmação de chegada e se se trabalhar com incerteza, o mais natural seria que o *Recetor* enviasse todos os Identificadores dos *Chunks* recebidos sempre que se concluísse um ciclo do Bloco *Transfer*. Isto levaria à existência de redundância e a um crescimento do número de Identificadores a enviar a cada ciclo do Bloco;
- Com Mensagens de confirmação de chegada, o número de Identificadores a serem enviados pelo *Recetor* poderia ser reduzido para o número de *Chunks* recebidos num determinado ciclo do Bloco *Transfer*, sendo que o número total de Identificadores enviados seria igual ao número total de *Chunks* daquela transferência em particular. Mas aqui seria introduzido mais um passo no ciclo do Bloco *Transfer* que poderia atrasar o processo num ao ambiente de grandes perdas de *Datagramas*.

No caso da Confirmação Negativa, o número de Identificadores a serem enviados também depende das perdas de *Datagramas* no ciclo do Bloco *Transfer*, mas tem sempre tendência para reduzir conforme a transferência avança. No melhor dos casos, todos os *Chunks* foram recebidos e não há necessidade de enviar qualquer Identificador. No pior dos casos, o *Recetor* apenas conseguiu receber 1 *Chunk* e seria necessário enviar todos os restantes Identificadores.

Em ambos os casos pode ser estabelecida uma regra em que, abaixo de uma determinada percentagem de *Chunks* recebida, pode ser enviado um *Datagrama* com esta Estrutura a *null*, desencadeando a retransmissão de todos os *Chunks* e a consequente retransmissão desnecessária de alguns.

De forma a visualizar a diferença entre as 3 estratégias, consideremos a seguinte simulação de uma transferência, com os resultados destas apresentados nas Figuras 9, 10 e 11, e a comparação destes na Figura 12.

Nesta transferência apenas serão recebidos 50% dos *CM* enviados pelo *Transmissor* por ciclo, exceto no quinto onde serão recebidos os restantes sem falhas. É importante referir que, mesmo com estas falhas na transmissão de *CM*, neste exemplo o processo de transmissão de *Datagramas* de Identificadores não é afetado pelo mesmo problema.

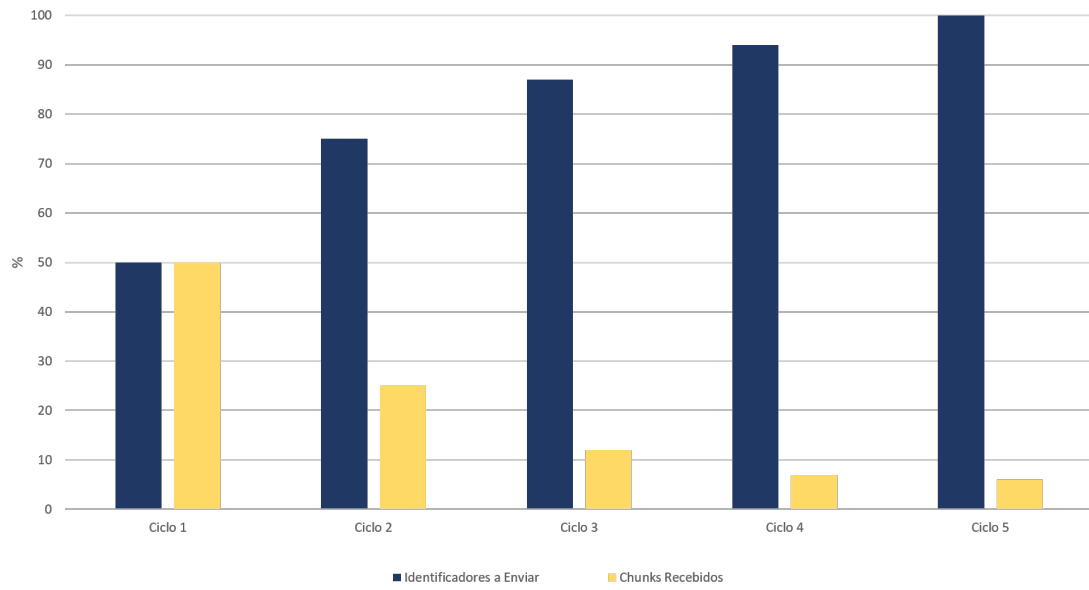


Figura 9: Gráfico de comparação entre % de Chunks recebidos e % de Identificadores a enviar ao Transmissor por ciclo com Confirmação Positiva e sem Mensagens de confirmação de chegada

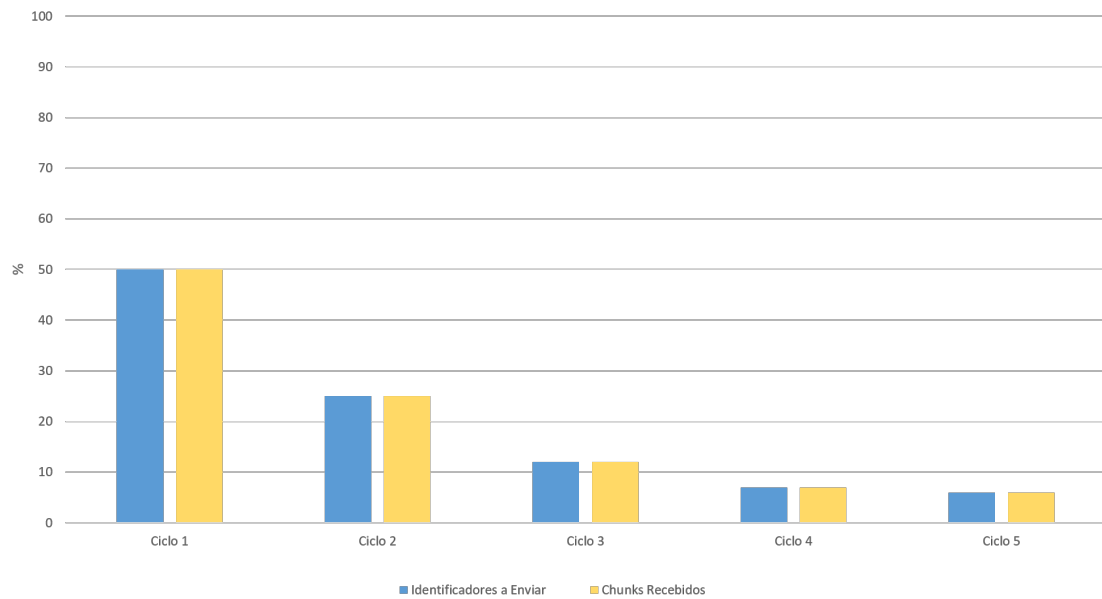


Figura 10: Gráfico de comparação entre % de Chunks recebidos e % de Identificadores a enviar ao Transmissor por ciclo com Confirmação Positiva e com Mensagens de confirmação de chegada

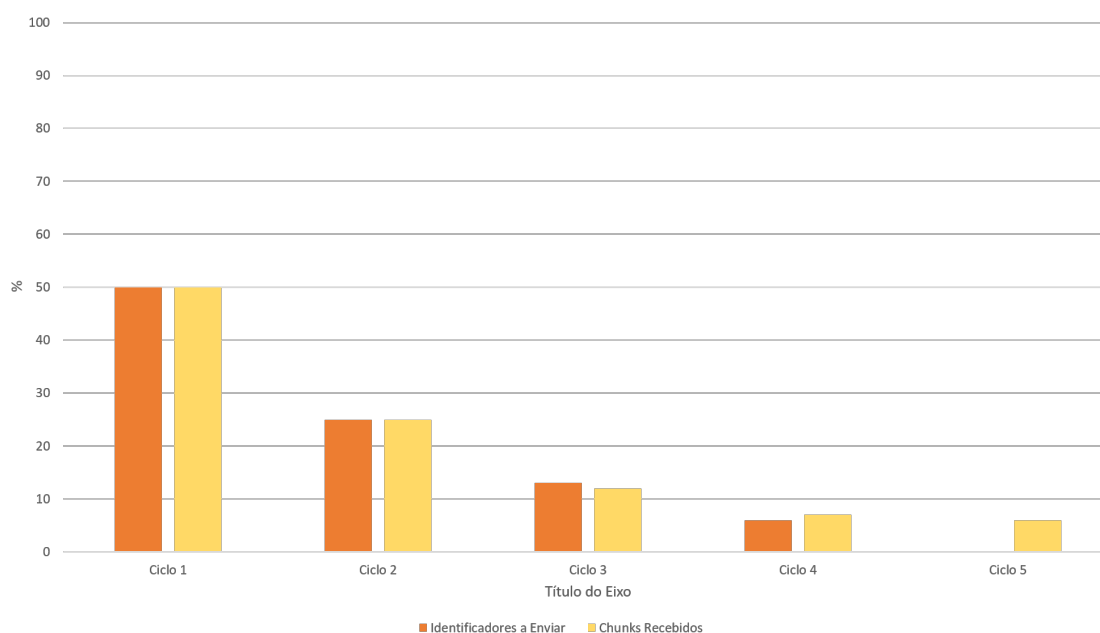


Figura 11: Gráfico de comparação entre % de Chunks recebidos e % de Identificadores a enviar ao Transmissor por ciclo com Confirmação Negativa e sem Mensagens de confirmação de chegada

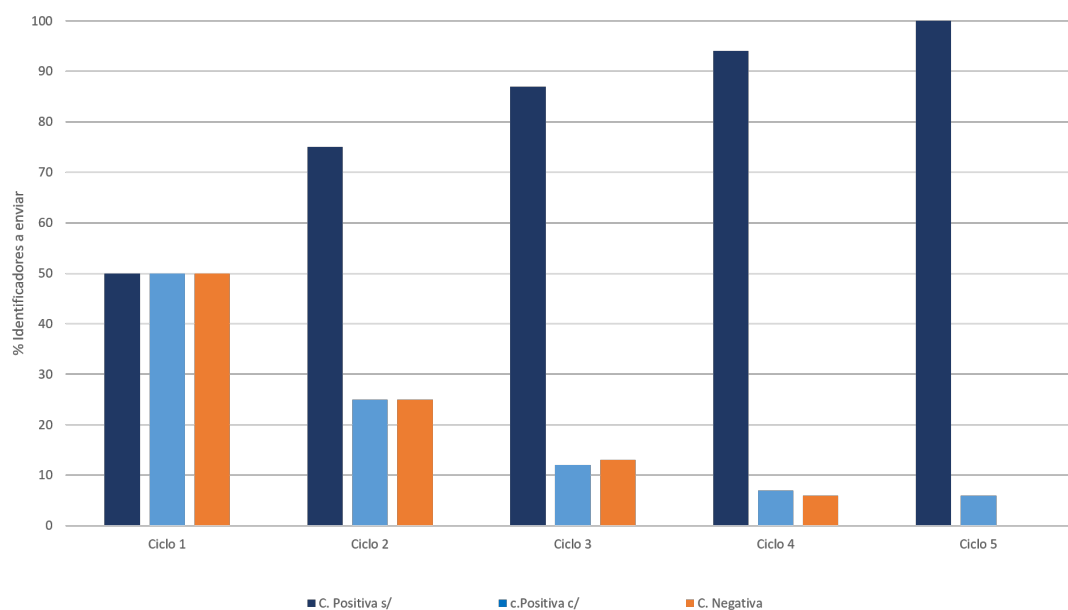


Figura 12: Gráfico de comparação entre % de Identificadores enviados ao Transmissor por ciclo em cada uma das 3 estratégias

É importante referir que, com a introdução de falhas no processo de transmissão dos Datagramas de Identificadores, existe uma influência no número de CM recebidos no ciclo seguinte. Em geral, o número de CM enviados pelo Transmissor, quando é usada uma

estratégia de Confirmação Positiva, é sempre igual ou superior que o número de *Chunks* em falta devido à incerteza de que *Chunks* estão realmente em falta. Em contraste, a falha de entrega de *Datagramas* de Identificadores numa estratégia de Confirmação Negativa leva uma redução de envio de *CM* por parte do *Transmissor* no próximo ciclo. Esta redução apenas significa que, os *Chunks* que não conseguiram ser marcados como em falta neste ciclo, não serão enviados pelo *Transmissor* no ciclo seguinte e estarão em falta novamente nesse mesmo ciclo. Isto faz com que o processo de transferência possa ser atrasado.

Com as simulações de Transferências das Figuras 9, 10 e 11, a comparação destas na Figura 12 e a tendência de transmissão redundante de *Chunks* desnecessária numa estratégia de Confirmação Positiva em consideração, serão enviados os Identificadores usando a estratégia de Confirmação Negativa sem Mensagens de confirmação de chegada. Dado que foi esta a estratégia escolhida, os *Datagramas* que até agora foram referidos como sendo *Datagramas* de Identificadores passar-se-ão a designar-se por *Missing Chunk ID (MCID)*.

3.1.5.4 Drops na Origem e Destino

Outro assunto que requer mais atenção é o de *Drops* de *Datagramas*, especialmente na Origem e Destino da Informação. Estes *Drops* podem ocorrer por vários motivos, sendo que os mais comuns são erros ocorridos durante a transmissão que alteram o conteúdo dos *Datagramas* e falta de espaço nos *Buffers* usados pelos *Sistemas Operativos* para guardar temporariamente os mesmos. Destes dois, ao nível deste Protocolo, apenas vão ser atenuados os *Drops* de *Datagramas* derivados da falta de espaço nos *Buffers* alocados pelos *Sistemas Operativos*, isto porque o *Controlo e Gestão de Erros* no conteúdo dos *Datagramas* vai variar de arquitetura para arquitetura dependendo do ambiente onde estas serão utilizadas. Por exemplo, numa arquitetura de Redes *Ad Hoc*, na ocorrência de um destes erros, é sensato pedir ao *Transmissor* que retransmita o *Datagrama* em questão quando possível, enquanto numa arquitetura de Redes *DTN*, onde os contactos entre dois *Nodos* podem ser de tempo limitado e com *Delays* significativos, seria sensato implementar um sistema de recuperação da informação que sofreu alterações na transmissão e, apenas em último recurso, pedir a sua retransmissão. Posto isto, o *Controlo e Gestão de Erros* deverá de ser analisado e implementado ao nível da arquitetura em questão, sendo que este Protocolo apenas se focará nos *Drops* associados ao espaço disponível nos *Buffers* disponibilizados pelo *Sistema Operativo*.

Para isso é necessário que nos certifiquemos que tais *Buffers* não estejam demasiado cheios, mas também não convém ser demasiado conservativo porque senão irá ser deixada *Performance* em cima da mesa.

Com este objetivo em mente temos que considerar que existem 2 *Buffers* por *DatagramSocket* (*Send Buffer* e *Receive Buffer*) e, por isso, existem 2 *Buffers* envolvidos por Transferência, o *Send Buffer* do *Transmissor* e o *Receive Buffer* do *Recetor*, e que ambos necessitam de ser controlados. Infelizmente, não é possível observar o conteúdo destes *Buffers* nem perceber quanto espaço

"vazio" eles têm, logo não é possível tomar decisões baseadas no espaço restante que estes *Buffers* possuem. Por esta razão, as decisões de *Controlo de Fluxo* terão que ser tomadas sobre a percentagem de ***Drops* observados no Destino**.

Uma das formas de garantir que não se está em nenhum dos extremos (causar *Overflow* dos *Send* e *Receive Buffers* e não aproveitar os mesmos) é o uso de um simples sistema de *Controlo de Fluxo* com *Feedback* do *Recetor*, visto que é este que vai detetar se estão a ocorrer grandes perdas por *Drop* na transmissão dos Dados. Para tal é usado um inteiro no *Datagrama TRI* do Bloco *Handshake* que tem como objetivo indicar ao *Transmissor* o limite superior de *Datagramas* por segundo que podem ser enviados. Apenas é necessário fazer chegar este *Feedback* do *Recetor* ao *Transmissor* no fim de cada ciclo do Bloco *Transfer*. Como anteriormente foi escolhida uma estratégia de Confirmação Negativa sem Mensagens de confirmação de chegada, não é possível usar a informação atualmente presente nos *Datagramas MCID* como *Feedback*, pois não é garantido que estes *Datagramas* cheguem ao seu Destino e, por isso, não se consegue garantir a medição correta de *Drops* no transmissor. Se fosse seguida uma estratégia de Confirmação Positiva com Confirmação de chegada, poderiam ser usados estes *Datagramas* para estimar as perdas por *Drops* no Destino com grande certeza. Por isso será necessário enviar este *Feedback* de outra forma.

Potencialmente poderia ser criado um novo *Datagrama* que apenas contivesse este *Feedback*, mas visto que iria trazer mais complicações como verificação de chegada de tais *Datagramas* ao *Transmissor* ou envio de múltiplos *Datagramas* para aumentar a probabilidade de chegada de pelo menos 1, será mais sensato introduzir esta informação num *Datagrama* já existente. Sabendo que este *Feedback* só pode ser dado no fim de cada ciclo do Bloco *Transfer* (que é onde a medição de perda de *Datagramas* é feita), faz sentido introduzir esta informação no *Datagrama MCID*. Visto que esta informação vai ser repetida em múltiplos *Datagramas*, não é necessário criar um sistema de verificação de chegada, pois se nenhum destes *Datagramas* chegar ao *Transmissor*, não haverá *Chunks* marcados como em falta, logo o *Transmissor* não enviará qualquer *Datagrama* e, por isso, não necessita deste *Feedback*.

Mesmo com este *Controlo de Fluxo*, é possível fazer melhor. Uma maneira simples de não encher os *Buffers* tão rapidamente é torná-los maiores. Para tal efeito existem pedidos aos *Sistemas Operativos* para o aumento de tais *Buffers*, mas estes não passam de "sugestões" e são estes que tomam a última decisão no tamanho dos *Buffers*, fazendo com que esta solução não seja a ideal.

Outra alternativa é fazer com que existam mais *Buffers* para o envio e receção dos *Datagramas*. Para isso é necessário fazer com que a receção de *Datagramas CM* seja modular para ser possível termos múltiplas *DatagramSockets* a escutar múltiplas Portas e assim obtermos mais *Buffers*. Isto também faz com que o processo de transferência possa ser paralelizado, obtendo um aumento de velocidade da transferência sem aumentar a carga sobre os *Buffers* e um conseqüente aumento de *Drops*. Visto que se pretende realizar uma

forma modular de receber *Datagramas* CM em Portas que apenas receberão destes *Datagramas*, seria uma vantagem se a ordem de *Parsing* destes fosse alterada de forma a aumentar a recepção de *Datagramas* por segundo e, por consequência, o aumento da velocidade a que o *Receive Buffer* de cada *DatagramSocket* é "esvaziado". Isto faz com que o limite máximo descrito anteriormente possa ter valores ainda mais elevados.

Devido a estas alterações, agora é necessário comunicar ao *Transmissor* que *Portas* se encontram disponíveis para a recepção de *Datagramas* CM por via do *Datagrama TRI* no Bloco *Handshake*, visível na Figura 13. Devido a esta alteração o nome do *Datagrama* passou de *Transfer Receiver Info* para *Transfer Multi Receiver Info (TMRI)*. Para além disso, é necessário alterar o funcionamento do Bloco *Transfer* para acomodar a transmissão de Dados com múltiplas *DatagramSockets* e a adição do *Feedback* resultante do sistema de *Controlo de fluxo* (Figura 14).

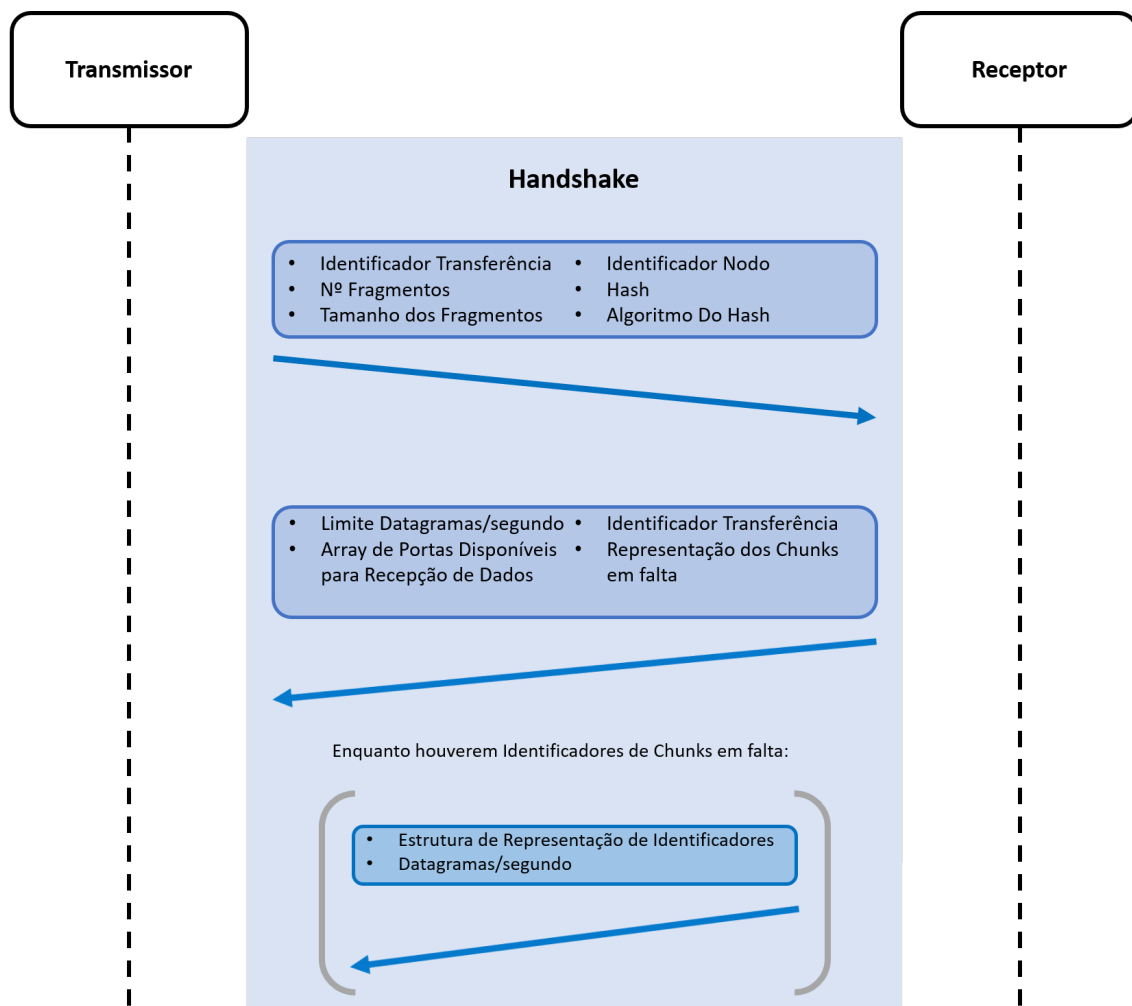


Figura 13: Terceira Iteração do Bloco Handshake

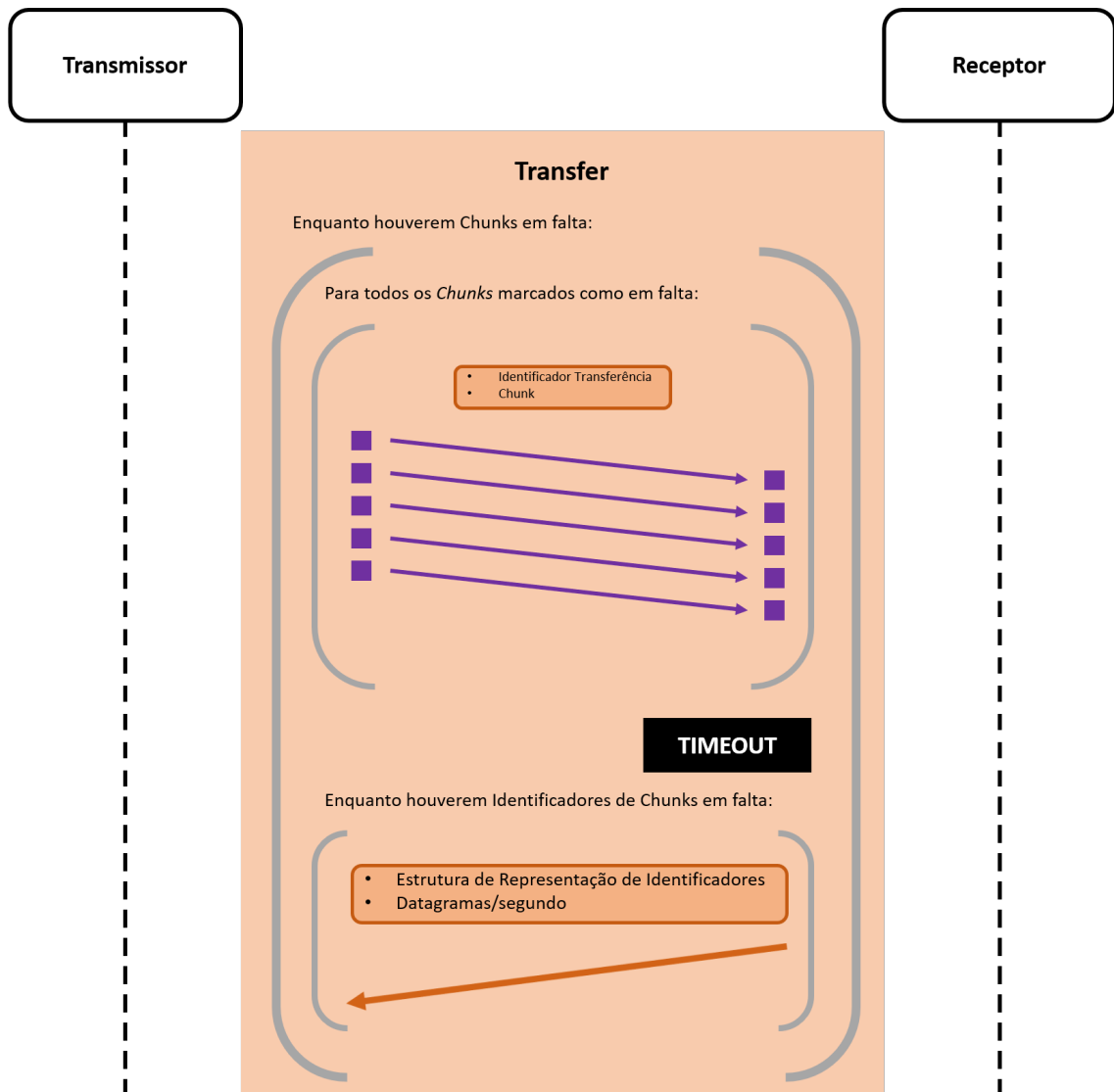


Figura 14: Terceira Iteração do Bloco Transfer

Com estas alterações, existe um potencial aumento significativo de recursos necessários para a execução de uma transferência devido à paralelização da mesma. É importante referir que é o *Nodo* que participa como *Recetor* na Transferência que decide a alocação de recursos que os 2 *Nodos* deverão de fazer, no entanto este pedido de alocação feita pelo Recetor trata-se uma sugestão e indicação da sua alocação **máxima** de recursos, sendo que o Transmissor poderá escolher fazer uma alocação diferente.

3.1.5.5 Timeout

Com estas mudanças no Bloco *Transfer*, ilustradas na Figura 14, seria possível tratar uma Transferência como múltiplas Transferências mais pequenas e, por isso, pode-se transformar

o *Timeout* de longa duração em múltiplos *Timeouts* mais pequenos e independentes entre si, atenuando a perda de tempo referida anteriormente. Por sua vez, isto permite que o envio de *Datagramas MCIDs* seja feito após cada um destes *Timeouts*, sendo que iria haver uma redução no número de Identificadores que cada *Thread* teria que enviar ao *Transmissor*, aumentando o número total de *Datagramas MCIDs*. Porém, esta divisão de *Timeouts* levaria a um aumento de perdas na construção dos *Datagramas MCIDs*, pois cada *Thread* iria criar os seus *Datagramas* independentemente das perdas das outras *Threads*, levando à existência de múltiplos *Datagramas* que não se encontram totalmente preenchidos.

3.1.5.6 Utilidade do Bloco Over

Finalmente temos o Bloco *Over* que é apenas constituído pelo *Datagrama Over*, que por sua vez só contém o Identificador da Transferência e serve para sinalizar o Fim do Protocolo e permitir a libertação de recursos. Tendo em conta que este Protocolo será usado em ambientes com grandes perdas de *Datagramas*, a estrutura actual do Bloco *Over* não consegue garantir que o *Datagrama Over* chegue ao seu Destino. Para tal seria necessário introduzir uma Mensagem de confirmação de Chegada, como um *ACK*, mas esta também pode sofrer com os *Drops* de *Datagramas* característicos de tais ambientes. Assim sendo, o *Over* não terá qualquer garantia de chegada ao seu Destino, mas vai ser seguida uma estratégia de *Best Effort* para aumentar a probabilidade de que pelo menos 1 *Over* chegue ao *Nodo Transmissor*. Para que tal seja possível é necessário enviar múltiplos *Datagramas Over* ao longo de um período de tempo. Estes valores (número de *Datagramas* a enviar e período de tempo) podem ser baseados nas estatísticas observadas durante o Bloco *Transfer*.

Um dos últimos problemas deste Protocolo é que, quando uma transferência é iniciada, para que esta termine sem ser devido a falhas na Rede, é necessário que se acabe de transmitir todos os Dados referentes a esta Transferência. Assim seria possível que um dos intervenientes (*Transmissor* ou *Recetor*) deixasse de responder, mas isso levaria a um estado de incerteza e por isso é necessária a introdução de um *Datagrama* que indique a interrupção da transferência. Visto que o *Datagrama Over* já tem a função de indicar o fim de uma Transferência, apenas será necessário introduzir uma *Flag* que indique se se trata de uma interrupção ou se a Transferência foi concluída com sucesso. Com esta mudança, o *Datagrama Over* deixa de ter um lugar fixo no Protocolo de Transferência de Dados visto que pode acontecer a qualquer instante.

No caso de não ser possível enviar um *Over* para sinalizar o fim ou interrupção da Transferência continua a ser necessário que os *Nodos* intervenientes sejam capazes de detetar que não é possível recuperar a Transferência e esta seja marcada como *interrompida* automaticamente. Mais uma vez é possível usar o sistema de *Timeouts* para este efeito. Uma Transferência pode ser marcada como *interrompida* automaticamente se forem detetados um número predefinido de *Timeouts* consecutivos.

Porém, este mecanismo faz com que seja possível a existência de transferências que conseguem transmitir toda a Informação pretendida mas, por algum motivo, os *Datagramas Over* não chegam ao Transmissor. Isto faz com que possam existir transferências que são marcadas como *interrompidas* no Transmissor e *terminadas* no Recetor. Para resolver este caso extremo, quando o Transmissor tentar recomençar esta transferência com um *Datagrama TMI*, o Recetor em vez de enviar um *Datagrama TMRI* enviará um *Datagrama Over* designado por *Early Over*, pois ocorre durante o Bloco *Handshake*. Em adição a esta função de sincronizar o estado da transferência nos Nodos intervenientes, este *Early Over* faz com que não seja possível começar uma transferência de Dados que o Recetor já possua, eliminando as ocorrências de transferências duplicadas desnecessárias.

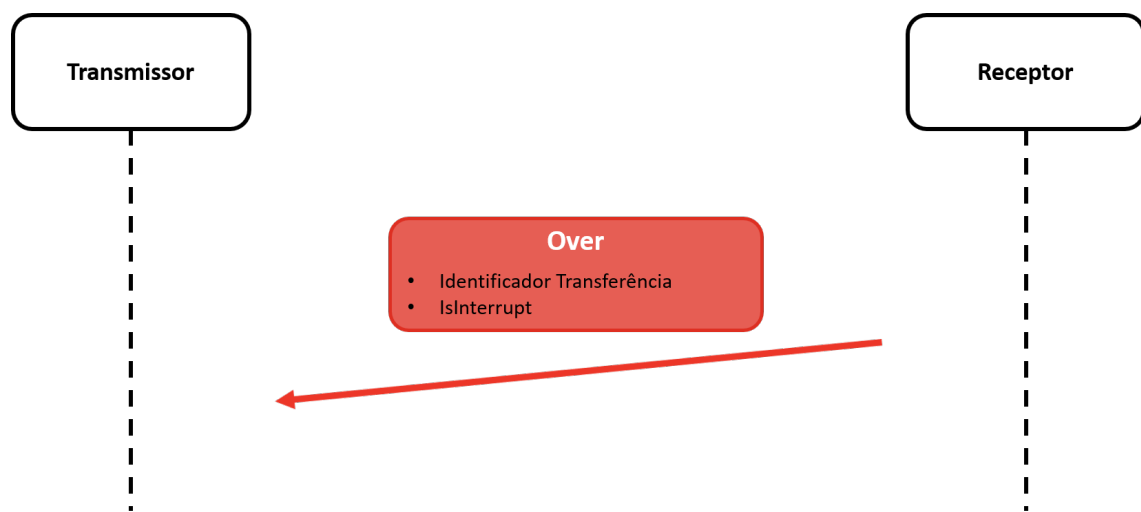


Figura 15: Segunda Iteração do Datagrama Over

3.1.5.7 Resiliência a movimentações de Nodos na Rede

Um dos grandes objetivos deste protocolo é resistir à movimentação dos Nodos participantes numa transferência na Rede, mas os mecanismos até agora especificados apenas apresentam resistência à perda de pacotes durante a transferência e ao fim inesperado, e eventual recomeço da mesma. Mesmo que estes mecanismos funcionem e consigam recuperar das falhas para as quais foram desenhados para mitigar, continuam a ser medidas de último recurso pois necessitam que a transferência seja totalmente interrompida para serem ativados. Por outras palavras, são mecanismos de recuperação passiva no que diz respeito à movimentação de Nodos e seria ideal ter uma forma de recuperar o mais rapidamente possível quando isso for detetado.

Inicialmente é necessário definir o que se deve tratar como movimentação de um Nodo na Rede. Assumindo que um determinado Nodo está a participar numa transferência, é certo que este tem pelo menos 1 *Network Interface Card (NIC)* conectado a um *Router* que lhe dá conexão a uma Rede, pelo menos 1 Endereço associado à conexão que tem com o *Router* e

consegue comunicar com o outro Nodo interveniente na transferência. Quando é referido que um Nodo se movimenta na Rede significa que o Nodo que tem uma conexão com um *Router A* e um *Endereço A* associado a essa conexão, fica fora do alcance desse *Router A*, entra em comunicação com um *Router B* e obtém um *Endereço B* resultante dessa nova conexão.

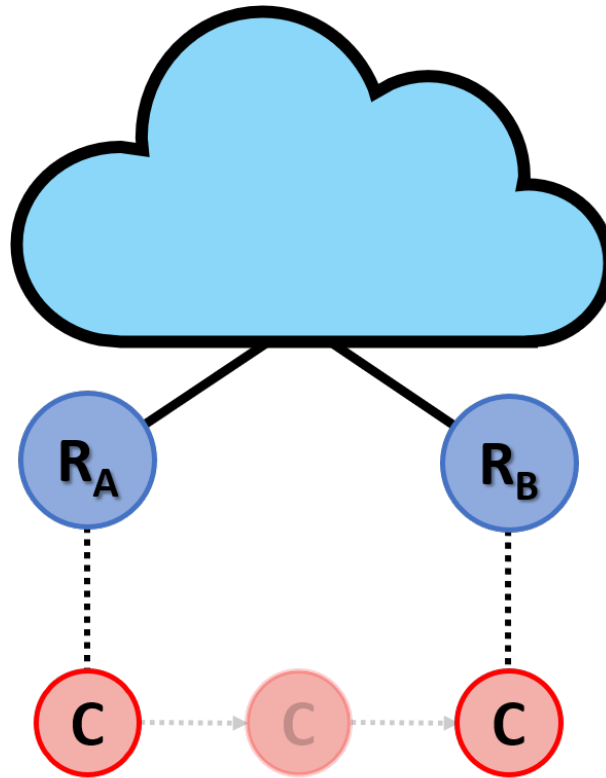


Figura 16: Mobilidade de um Cliente e mudança de Endereço

É este acontecimento que o protocolo tem que ser capaz de detetar e agir para que os *Datagramas* relativos à transferência a decorrer possam ser redirecionados para o novo Endereço do Nodo. Assim, sempre que for detetada uma alteração de Endereço, o Nodo necessita de enviar um *Datagrama* especial para informar o outro interveniente na transferência de tal acontecimento e este poder dar continuidade à mesma. No caso de um Nodo se desconectar de um *Router A* e reestabelecer conexão com esse mesmo *Router A* não existirá alteração de Endereço e será tratado como um período de grandes perdas.

Visto que cada conexão entre um Nodo e um *Router* tem as suas características relativamente ao estado da conexão, nomeadamente a que velocidade é possível realizar a troca de dados entre eles, esta informação também deverá ser introduzida neste *Datagrama* denominado de *Network Status Update (NSU)*. Com isto temos um novo *Datagrama* que possui:

- ID da transferência;

- Novo Endereço do Nodo que envia este *Datagrama*;
- Informação Relativa a velocidade da conexão.

Por último, como é o Recetor de uma Transferência que determina a velocidade a que esta deve de ser feita, quando este recebe um *NSU* do Transmissor deverá de calcular a nova velocidade da Transferência e enviar um *NSU* de resposta para informar o Transmissor desta nova velocidade.

Visto que o processo de desconexão de um *Router* e conexão a outro não é instantâneo pode levar a alguns comportamentos inesperados que serão analisados de seguida.

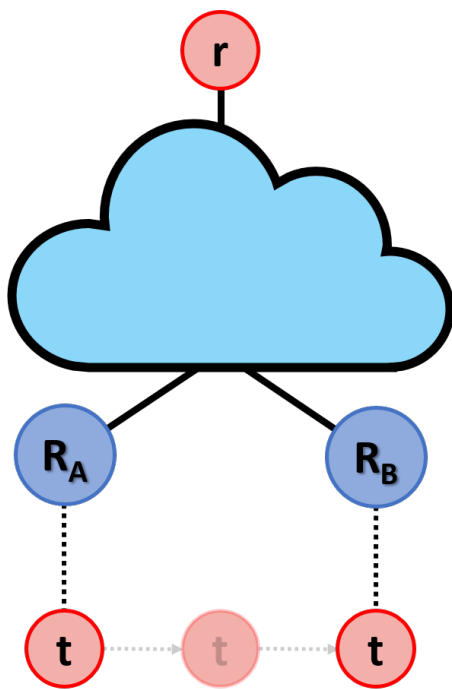


Figura 17: Mobilidade do Transmissor durante uma transferência

Como se pode observar na Figura 17, desde o momento em que o Transmissor perde conexão com o *Router A* até estabelecer uma com o *Router B* passa algum tempo t em segundos com $t > 0$.

Primeiramente é necessário que o Transmissor interrompa o envio de *Datagramas* durante estes t segundos, pois, com a falta de conexão, os *Datagramas* enviados poderão encher o *Buffer* de envio de *Datagramas* anteriormente alocado pelo Sistema Operativo, potencialmente serão perdidos e necessitarão obrigatoriamente de ser reenviados.

Além disso, se este tempo t for suficientemente extenso, o Recetor pode deixar de receber *Datagramas* e eventualmente poderá identificar essa falta como uma perda devido a *Drops* na Rede de forma errada e ativar mecanismos de recuperação desnecessariamente. Visto que o Bloco *Handshake* apenas é composto por uma simples troca de *Datagramas* (o recetor pode ter que enviar mais que 1 *Datagrama* mas não fica à espera de uma resposta direta a esses, daí ser o mesmo

que 1 *Datagrama* neste caso), este comportamento só pode acontecer no Bloco *Transfer* e, por isso, é possível afirmar com grande certeza que o mecanismo de recuperação a ser ativado desnecessariamente é o de envio de *Datagramas MCIDs* a marcar *Chunks* como não recebidos, desencadeando o envio de *Datagramas* Duplicados por parte do Transmissor quando receber. Uma forma de tentar atenuar este problema seria suprimir o mecanismo de recuperação quando o Recetor recebesse um *Datagrama NSU*, mas haverá situações onde o Transmissor enviou todos os *Datagramas CM* que tinha para enviar e seria necessário enviar os *MCIDs* para recuperar desta situação de *Drop* de *Datagramas*. Assim sendo, o mais sensato será construir um sistema de registo do lado do Transmissor de forma a saber que

Datagramas CM não foram enviados, e uma vez recebidos os *Datagramas MCIDs* será possível distinguir quais são os que resultariam numa transmissão duplicada e os que necessitam de ser retransmitidos.

É importante referir que este sistema de deteção de pedidos de retransmissão indevidos não impede a transmissão duplicada, mas sim reduz o número de retransmissões duplicadas significativamente, pois existe um pequeno período entre o Transmissor enviar o *Datagrama NSU* e receber os *Datagramas MCIDs* em que está constantemente a enviar *Datagramas CM* se o tiver que fazer. Neste período o Recetor estará a dar alguns *CM* como perdidos em *Drop de Rede*, mas estes encontram-se a caminho. Todo este processo pode ser visualizado na sequência de imagens da Figura 18.

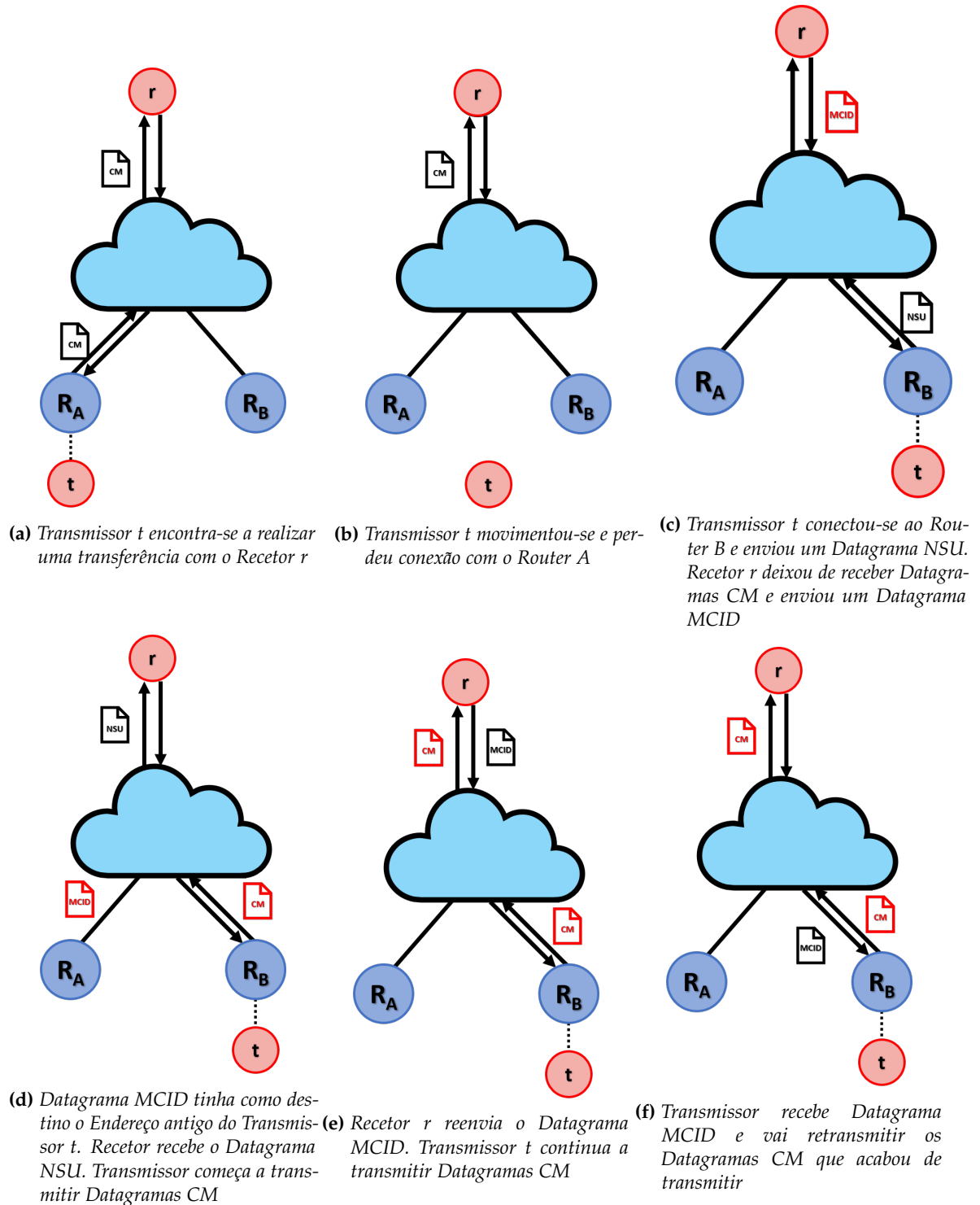


Figura 18: Demonstração do pedido de retransmissão de Datagramas CM indevido com a mobilidade do Transmissor

Caso este momento de reconexão a um outro Router acontecer no fim de uma transmissão, O Recetor ao receber um NSU deverá de retransmitir o Over ao Transmissor para sinalizar o

fim da transmissão. Caso isso não aconteça, o sistema de *Timeouts* no Transmissor deverá de ser capaz de detetar ao fim de um determinado tempo que já deveria de ter recebido *MCIDs* ou um *Over* e dará esta transmissão como *Interrompida* devido à falta de conexão ao Nodo Recetor.

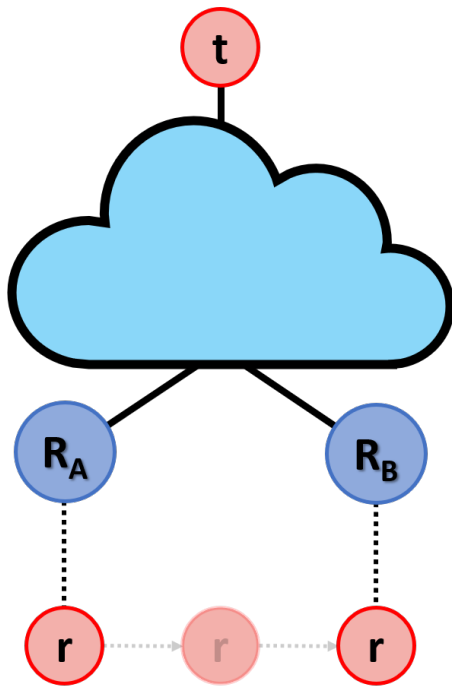


Figura 19: Mobilidade do Recetor durante uma transferência

No caso do Recetor se desconectar do *Router A*, passar t segundos sem conexão ($t > 0$) e reconectar-se ao *Router B*, algo de semelhante irá acontecer. Se t for extenso o suficiente o Recetor irá começar a enviar *MCIDs* ao Transmissor, mas como este não tem conexão durante esses t segundos apenas serão enviados, juntamente com um *NSU*, quando esta voltar a ser estabelecida. Isto resultará na mesma situação descrita anteriormente, onde o Recetor pede a retransmissão de *Datagramas CM* que ainda não tinham sido transmitidos, mas com o sistema de registo já descrito, este problema fica mitigado. Em contraste, neste caso não haverá deteção de *Datagramas CM* duplicados por parte do Recetor (com uma exceção), mas os *Drops* de Rede serão mais elevados, pois o Transmissor continuará a enviar os *CM* para o Endereço inicial do Recetor até receber o *NSU*, mesmo que o Recetor já não se encontre naquela rede. A exceção que permite que sejam detetados *Datagramas CM* em duplicado do lado do Recetor acontece quando existe um intervalo

significativo entre a chegada do *Datagrama NSU* e a chegada do *Datagrama MCID*, permitindo o envio de *Datagramas CM* que serão marcados como em falta. Este acontecimento pode ser visualizado na sequência de imagens da Figura 20.

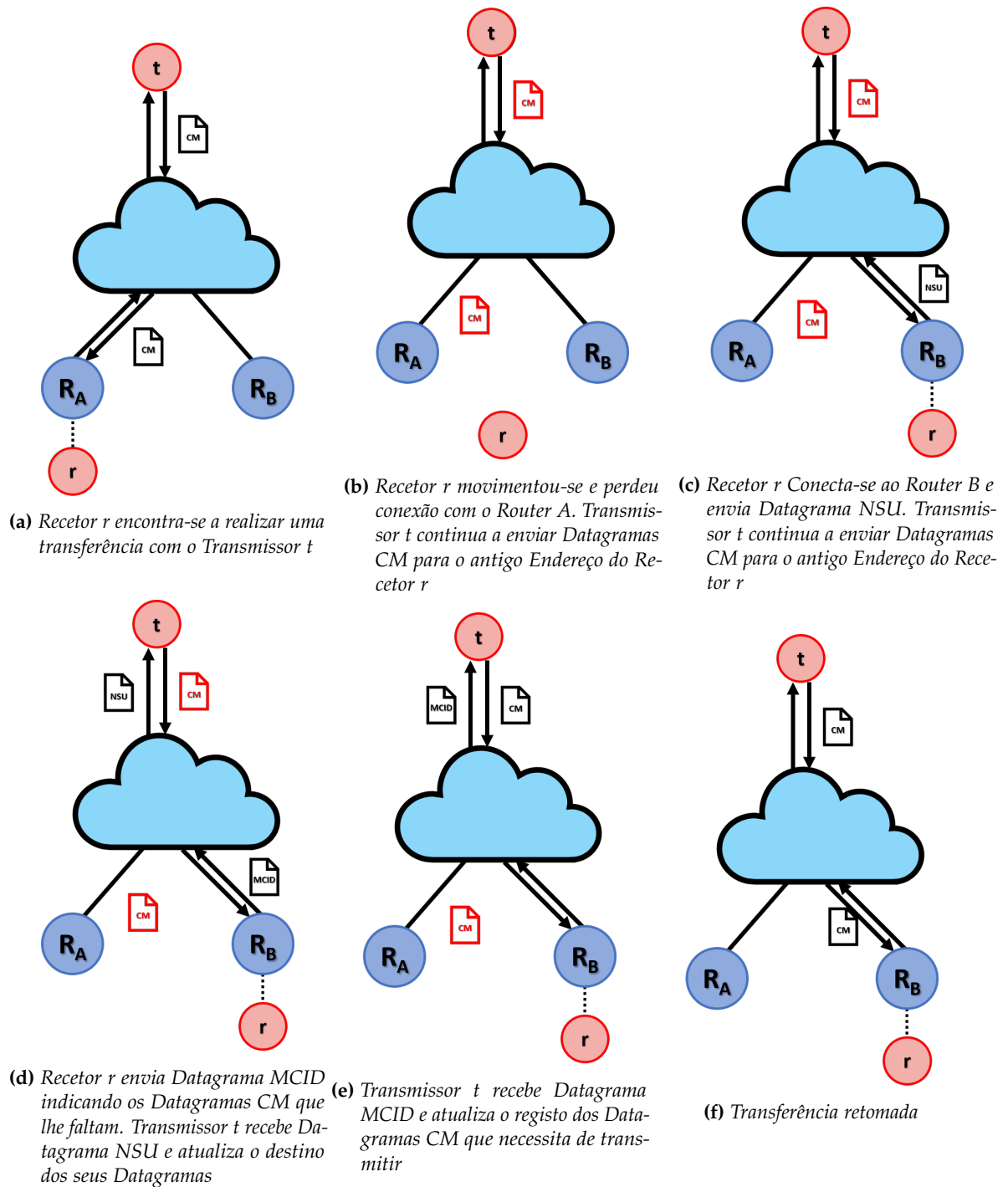


Figura 20: Demonstração da perda de Datagramas CM devido a movimentação do Recetor

Esta abordagem funcionará em muitas arquiteturas de aplicações *tradicionais* onde um dos Nós intervenientes, como por exemplo um Servidor, pode ser considerado *fixo* durante o processo de movimentação e aquisição de novo endereço por parte do outro. No entanto, esta também é aplicável a aplicações que usam comunicação cliente-cliente *Peer-to-Peer* (P2P).

Mas apesar desta solução para a Resiliência na movimentação de Nodos na Rede ser válida, existem 3 casos extremos em que ela não consegue reestabelecer a transmissão de dados entre os 2 Nodos de uma transferência.

A primeira acontece quando, numa arquitetura de comunicação cliente-cliente *P2P*, ambos se movimentam simultaneamente, como se encontra demonstrado na Figura 21. Esta abordagem assume que um dos Nodos completa a sua movimentação enquanto o outro se mantém estático e pronto para receber o *Datagrama NSU* com o novo endereço do Nodo em movimento. Se ambos se movimentam simultaneamente, ambos enviarão os seus *Datagramas NSU* para os endereços antigos onde já não se encontram Nodos a ouvir e, por isso, a transmissão vai ser *interrompida*.

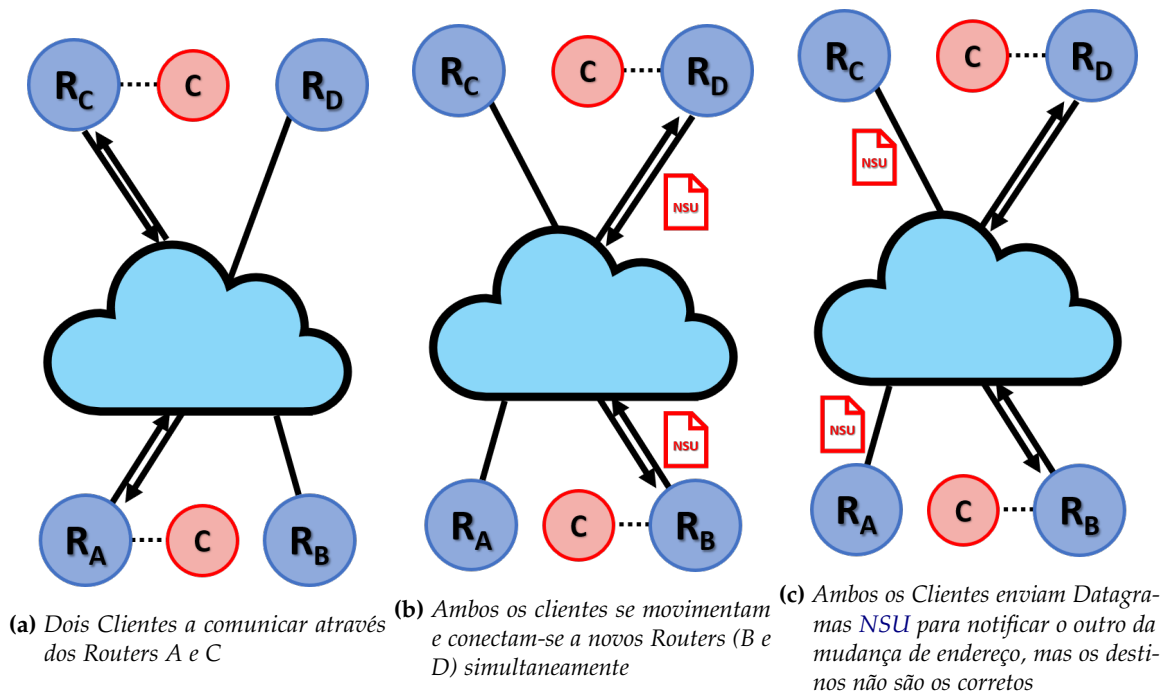


Figura 21: Demonstração de movimento simultâneo dos 2 Clientes

A segunda acontece quando um dos Nodos perde conexão a um *Router A* e reconecta-se a um *Router B* que não faz parte da mesma Rede que o *Router A* (Figura 22), cortando a comunicação entre os Nodos e levando a que a transmissão seja *interrompida*.

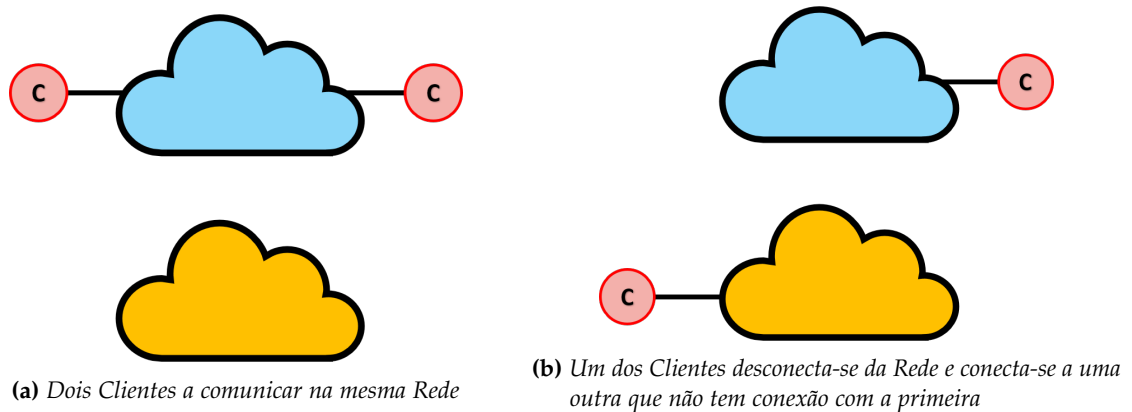


Figura 22: Demonstração de um movimento que não é possível estabelecer conexão entre os Clientes

Por último, se o tempo de reconexão t for demasiado elevado, o sistema de libertação de recursos construído sobre o sistema de *Timeouts* pode ser ativado e mover a transmissão para o estado *interrompida*.

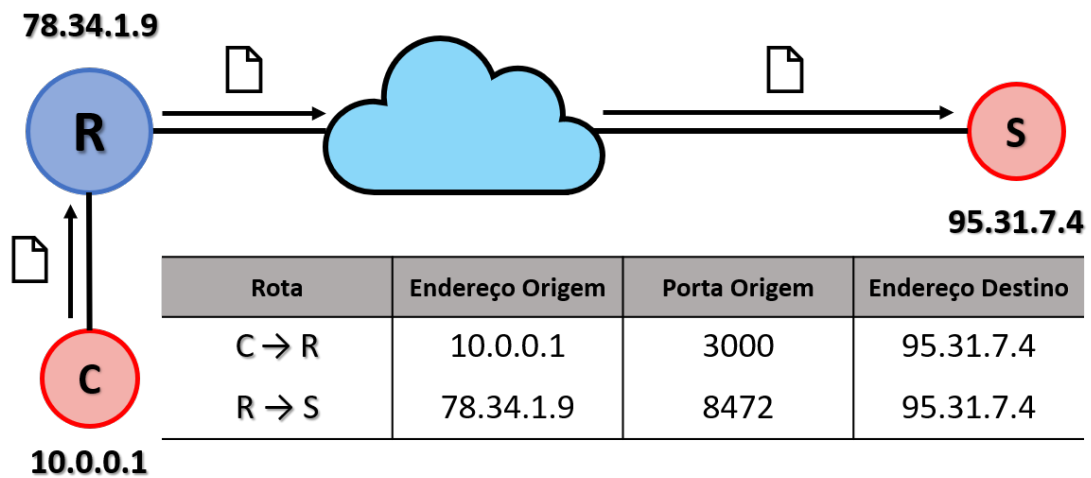
Um ponto interessante é o da Movimentação de Nodos numa Rede *Ad Hoc*, pois nestas redes não existem ligações com *Routers*, logo não existem alterações de Endereços derivados da Movimentação. Assim sendo, não existe nenhum evento que desencadeie a troca destes *Datagramas*, levando a que este problema tenha que ser resolvido a um nível superior.

3.1.5.8 Network Address Translation

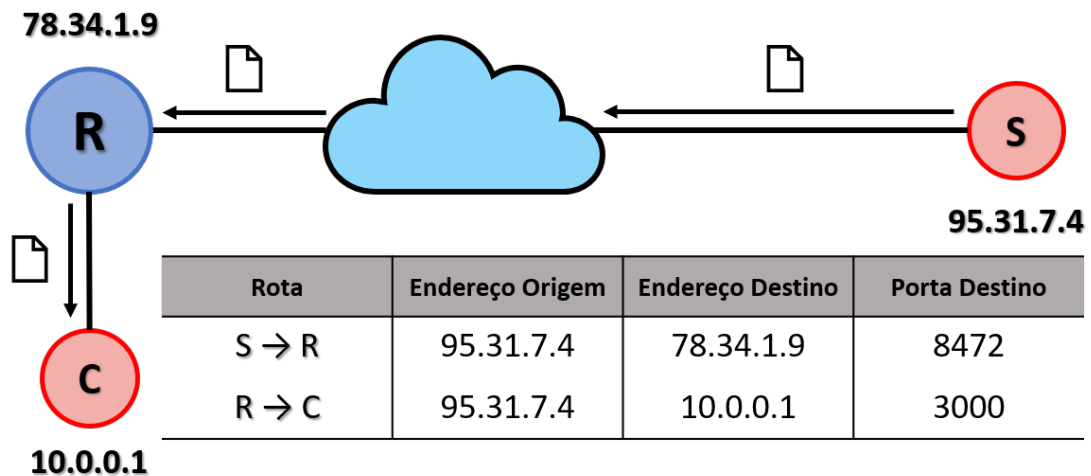
Network Address Translation (NAT) [26][27] é uma tecnologia que foi inicialmente desenvolvida para lidar com instabilidades na Rede e simplificar o processo de atribuição de endereços a novos Nodos que entrassem numa Rede. Hoje em dia ela é usada para lidar com o grande esgotamento no espaço de Endereços *IPv4* devido ao elevado número de dispositivos que requerem uma conexão à Rede. Com esta tecnologia é possível criar redes privadas, geridas internamente, com apenas 1 endereço *IPv4* público reduzindo drasticamente o número de endereços *IPv4* públicos necessários para possibilitar a conexão de mais equipamentos. Isto é atingido através da alteração dos campos Endereço e Porta nos *Datagramas* que passam por um *Router* que usa esta tecnologia. Apesar desta tecnologia ter bastantes benefícios, no *Scope* deste Protocolo pode tornar o funcionamento normal deste um bocado complicado ou até o impedir por completo.

Numa rede do mundo real, este Protocolo, da forma como está desenhado e sem mais nenhuma consideração em relação às Redes *NAT*, apenas funcionaria entre 2 Nodos que estivessem na mesma Rede *NAT*. Se for necessário o uso do Protocolo entre 2 Nodos que se encontram em 2 Redes *NATs* distintas seria necessário configurar a Rede de forma a "fundir" essas Redes numa só Rede *NAT* ou implementar um mecanismo que consiga "contornar" o funcionamento destas Redes.

A *Network Address Translation* tira partido do facto de existirem muitas mais Portas de comunicação que aquelas que são necessárias para garantir uma conexão a um dispositivo. Um *Router* que use esta tecnologia (Figura 23), como o próprio nome indica, faz uma tradução de Endereços e Portas, mapeando o Endereço e Porta Origem de um *Datagrama* que “sai” da sua rede interna para o seu Endereço público e uma Porta que fica reservada para esta conexão. Quando um *Datagrama* de resposta chega a esse mesmo *Router* ele faz a tradução contrária, trocando o Endereço e a Porta Destino pelo par Original, redireccionando a resposta para o dispositivo correto.



(a) A passagem do *Datagrama* emitido pelo Cliente C pelo Router R causa a alteração do Endereço e Porta Origem



(b) O *Datagrama* de resposta emitido pelo Servidor S tem como Destino o Endereço do Router R. À chegada ao Router R, o Endereço e Porta Destino são alterados para o Endereço e Porta que o Router inicialmente traduziu

Figura 23: Comunicação entre um Cliente C e um Servidor S através de uma Rede NAT

Em contrapartida, existem problemas quando uma conexão necessita de ser feita a partir do exterior. Isto acontece pois a *NAT* assume que as conexões são sempre iniciadas internamente, dando-lhe uma oportunidade para fazer a tradução de Endereços e registar esta tradução na *Tabela NAT*. Se uma conexão for iniciada "do lado de fora", quando o *Datagrama* chega ao *Router* ele não tem qualquer registo que o ajude a fazer o redireccionamento para o dispositivo pretendido.

Assim sendo, para ser possível estabelecer uma conexão iniciada a partir do exterior são necessários mecanismos elaborados como *NAT Hole Punching* [28][29][30] para o conseguir.

Existem algumas variações nas implementações de *NAT Hole Punching* mas todas elas seguem a mesma filosofia, fazer com que os *Nodos* que se encontram em Redes *NAT* inicializem a comunicação, dando uma oportunidade aos *Routers* de registarem a tradução de Endereços nas suas *Tabelas NAT*. Outra coisa que todas as implementações de *NAT Hole Punching* têm em comum é a presença de um *Servidor*, com um Endereço público, que vai servir de intermediário no processo de estabelecimento de conexão.

Consideremos 2 *Clientes* (c_1 e c_2) cada um deles ligado a um *Router* que tira partido de *NAT* e que estão ligados a uma rede onde também está presente um *Servidor* (Si) que vai servir de intermediário neste *Hole Punching* para conectar os 2 *Clientes*. Neste exemplo, ilustrado na Figura 24, vai ser c_1 a inicializar a conexão com c_2 .

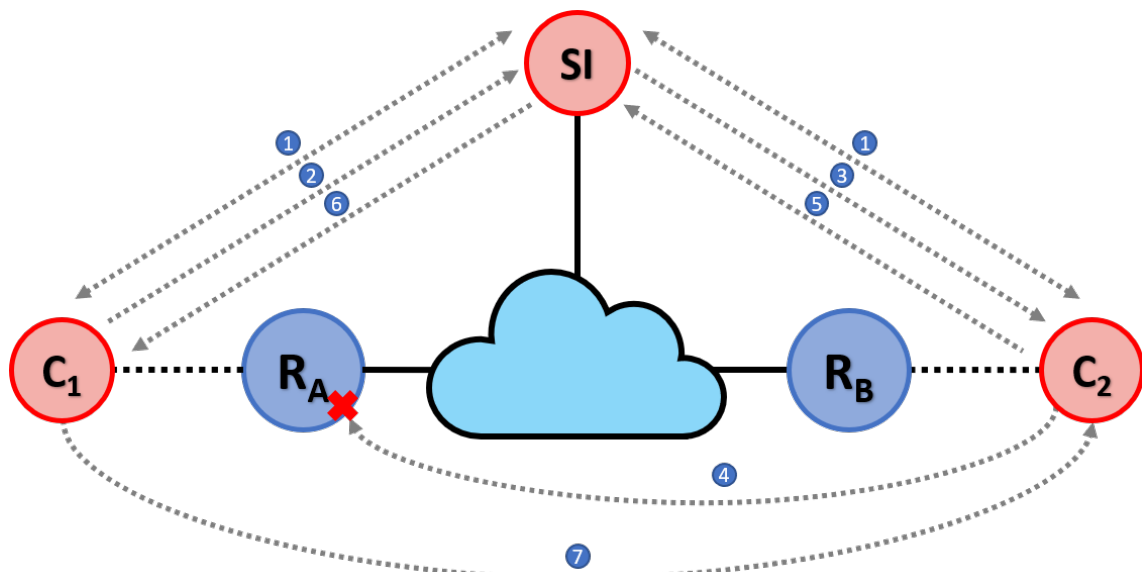


Figura 24: Procedimento genérico de *NAT Hole Punching*

1. Ambos os *Clientes* enviam uma mensagem a Si para se "registarem" e poderem receber pedidos de conexão. *Routers* R_A e R_B registam novas entradas nas suas *Tabelas NAT*. Si envia mensagens de resposta a C_1 e C_2 ;
2. C_1 envia uma mensagem de pedido de conexão a C_2 ao Si ;

3. *Si* envia o pedido de conexão a C_2 ;
4. C_2 envia *Datagrama* diretamente a C_1 . *Router* R_B registra nova entrada na sua *Tabela NAT*. *Router* R_A descarta *Datagrama* por não ter correspondência na sua *Tabela NAT*;
5. C_2 envia mensagem a *Si* para este enviar uma mensagem a C_1 para inicializar a conexão;
6. *Si* envia mensagem a C_1 para este inicializar a conexão;
7. C_1 envia mensagem a C_2 . *Router* R_A registra nova entrada na sua *Tabela NAT*. *Router* R_B redireciona mensagem para C_2 . Neste momento C_1 e C_2 conseguem comunicar diretamente.

Este processo pode ser agilizado e sofrer algumas alterações para melhoria de performance ou adição de funcionalidades, mas segue o mesmo procedimento que outros procedimentos de *NAT Hole Punching*.

Embora este método funcione em muitas situações, arquiteturas que não dispõem de um Servidor para servir de intermediário não conseguem realizar o *Hole Punching* para garantirem que a comunicação possa ser realizada e por isso este ponto requer maior atenção.

Outro problema que aparece com a utilização de Redes *NAT* é a possível alteração do *Scope* dos Endereços com a movimentação de um Nodo na Rede. Este problema, que se encontra ilustrado na Figura 25, acontece devido à tradução de Endereços, pois numa transferência entre 2 Nodos que se encontram na mesma Rede *NAT*, estes estarão a usar os Endereços de *Scope Local* (192.168.0.0/24 ou 10.0.0.0/24) e, quando um dos Nodos se move para uma outra Rede que se encontra ligada à primeira, este continuará a enviar *Datagramas* com o mesmo *Scope* que, neste caso, se trata de um *Scope Local* e não chegam ao Destino pretendido. Assumindo que os 2 Nodos conseguiram estabelecer uma conexão através das Redes *NAT*, quando um dos Nodos se movimenta para a mesma Rede onde se encontrava o primeiro Nodo, estes conseguem manter a comunicação entre eles, pois estarão a usar Endereços de *Scope Externo*, sendo que, quando o *Router* que abriga os Dispositivos recebe estes *Datagramas* consegue perceber que este reencaminhamento *Externo* na verdade se trata do seu Próprio endereço público.

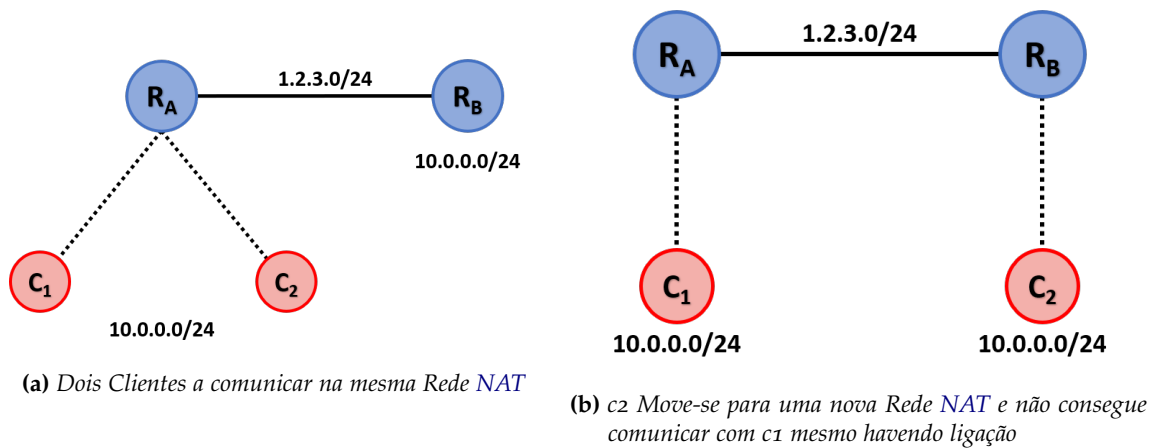


Figura 25: Demonstração do problema existente com o Scope dos endereços

Nestas arquiteturas que não dispõem de um Servidor com um Endereço público existe a dificuldade de encontrar outros Nodos na Rede para comunicar, pois não existe um Serviço central que mantém o registo de Nodos e os seus Endereços, por isso necessitam de outro método para a descoberta de Nodos na Rede. Para este efeito podem ser utilizadas as Mensagens *Multicast* do *UDP*, dando uma ferramenta a essas arquiteturas para poderem descobrir possíveis Nodos que se encontrem próximos e que possam ter interesse em comunicar. Assim foi criado mais um *Datagrama* que é designado de *Knock*. A troca deste *Datagrama* é completamente independente do resto do Protocolo, mas pode ser utilizada para o iniciar. Também é possível realizar uma troca de dados utilizando este *Datagrama*, mas será muito limitada, pois apenas é enviado 1 *Datagrama* que estará limitado ao tamanho do *MTU*.

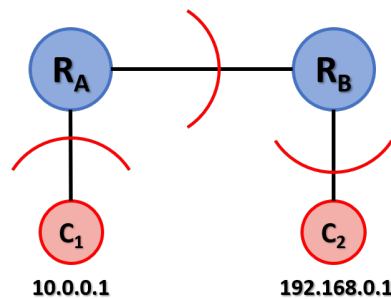
Com a utilização deste mecanismo vem uma obrigação. Se um Nodo precisar de o utilizar para a descoberta de outros Nodos, ele próprio tem que ser *visível* e responder a possíveis *Datagramas Knock* que possa receber via *Multicast*, pelo menos durante o período de tempo que está à procura.

Uma característica deste tipo de troca de informação é que não se trata de uma troca entre 2 Nodos (1:1), mas sim entre um grupo de Nodos que se encontram no mesmo grupo *Multicast* (1:N). Sendo assim, a informação a ser enviada nestes *Datagramas Multicast* não deve de ser personalizada a um potencial Nodo, mas sim uma informação geral que qualquer Nodo que a receba consiga utilizar. Em contraste, os *Datagramas Knock* que são enviados em resposta podem ser personalizados, pois estes são enviados em *Unicast* ao emissor do *Datagrama Knock* em *Multicast*.

Uma característica interessante desta troca de informação é que, quando é utilizada em Redes NAT, o *Datagrama Multicast* cria um registo na Tabela NAT do Router do Nodo origem desse *Datagrama* e este não é descartado por outros Routers NAT, sendo que estes últimos irão reencaminhar este *Datagrama* para qualquer Nodo "interno" que tenha anunciado a sua

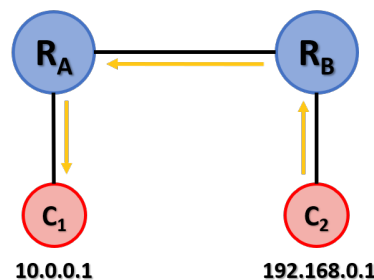
intenção de receber *Datagramas Multicast* desse mesmo grupo [31]. Assim, assumindo que os Nodos que recebem este *Datagrama Multicast* respondem ao Nodo que o emitiu utilizando um *Datagrama Unicast*, também é feito um registo na Tabela NAT dos Routers destes Nodos que emitem a resposta (Figura 26). Estes são os princípios que permitem a realização do NAT Hole Punching descrito anteriormente, mas neste caso é possível realizar este efeito sem que seja necessário qualquer *Servidor Intermediário*. Assim, com esta troca de *Datagramas*, pode ser possível começar um procedimento de *Hole Punching* entre 2 Nodos que se encontram em 2 Redes NAT distintas sem a ser necessário utilizar um *Servidor Intermediário*.

Endereço interno	Porta interna	Porta externa	Endereço interno	Porta interna	Porta externa
10.0.0.1	3000	12845			



(a) C_1 emite um *Datagrama Multicast*, forçando um registo na Tabela NAT do Router A . Quando o Router B recebe este *Datagrama Multicast* não o descarta e reencaminha este para C_2

Endereço interno	Porta interna	Porta externa	Endereço interno	Porta interna	Porta externa
10.0.0.1	3000	12845	192.168.0.1	5000	41293



(b) C_2 responde ao *Datagrama Multicast* com um *Datagrama Unicast*. Este *Datagrama* ao passar pelo Router B força a que este o registe na sua Tabela NAT. Quando este *Datagrama* chega ao Router A , este é capaz de fazer a tradução contrário devido à passagem do *Datagrama Multicast*

Figura 26: Hole Punching usando *Datagramas Multicast*

3.2 ENDEREÇOS

Um pormenor que é necessário neste Protocolo para permitir o seu bom funcionamento nos ambientes especificados são os Endereços que este precisa de suportar. Para além dos

Endereços *IPv4* será necessário o suporte para Endereços *IPv6* [32] que trazem algumas modificações que facilitam o funcionamento em certos ambientes, mas também vem com algumas restrições que têm que ser consideradas. Com os Endereços *IPv6* vem o conceito de *IPv6 LinkLocal* que servem para comunicações a 1 Salto de distância e que normalmente são usados para a comunicação entre um Dispositivo e o *Router* a que este se encontra conectado, tendo a forma única *fe80::*. Nestes casos, o *Router* tem o cuidado de atribuir um *IPv6 LinkLocal* único na sua Rede a cada um dos Dispositivos a que se encontra conectado, mas é possível que o próprio Dispositivo se autoatribua um *IPv6 LinkLocal* baseando-o no Endereço *MAC* da Interface que o irá utilizar, fazendo com que este seja teoricamente único. Apenas é teoricamente único porque o Endereço *MAC* de uma *Network Interface Card* (NIC) é atribuído pelo vendedor da mesma, mas este pode ser alterado, podendo levar a uma colisão de Endereços entre 2 Dispositivos que possuam o mesmo Endereço *MAC* e isto pode ser um problema para a Segurança. A este processo dá-se o nome de *IPv6 Stateless Address Autoconfiguration* [33].

Estes Endereços são importantes neste Protocolo, pois, com o *IPv6 Stateless Address Autoconfiguration* [33], cada dispositivo pode atribuir a si próprio um *IPv6* válido e teoricamente único em ambientes em que não existem Dispositivos como *Routers* para garantirem a distribuição de Endereços únicos. Assim, a atribuição de Endereços, *Routing* e Troca de Dados entre Nodos é significativamente simplificada em Redes *Ad Hoc* e Redes *DTN*. Para além disso permite com que sejam feitas Trocas de Dados entre 2 Nodos numa Rede "Normal" que estejam ligados ao mesmo *Router*, sendo que não existiria qualquer comunicação com o Exterior, visto que estes Endereços são a 1 Salto de Distância do *Router*.

Uma restrição que o *IPv6* traz é não suportar o *Broadcast* de mensagens, mas em contra partida este obriga ao suporte de Mensagens *Multicast* que têm como objetivo a troca de mensagens entre Nodos que pertençam ao mesmo *Grupo Multicast*, substituindo assim as mensagens *Broadcast*.

3.3 ESTATÍSTICAS

Um ponto crítico em qualquer processo automatizado é a medição de estatísticas para que possam ser tomadas as medidas corretas conforme os acontecimentos observados na execução desse mesmo processo. Quantas mais observações e medições possam ser feitas durante a execução de tal processo mais informação estará disponível para se poder tomar as decisões corretas e tentar evitar situações que possam ser indesejadas. Isto é ainda mais verdade num programa que utiliza a Rede para realizar transmissões de dados, pois este não tem qualquer controlo ou acesso ao que acontece nessa mesmas Rede. Como estes programas são completamente cegos ao estado atual da Rede que estão a usar, são necessárias várias

medições do que acontece durante a transmissão e recepção de dados num determinado período de tempo para se estimar o estado da mesma.

Estas medições têm que ser feitas sobre acontecimentos que seguem padrões que um Nodo possa criar e que o outro consiga detetar a sua presença, ausência ou alterações do que estaria à espera. No caso deste processo, com os 2 Nodos intervenientes numa transferência, o acontecimento que tem que ser observado e medido é a transmissão de *Datagramas*, principalmente *Datagramas CM*, durante uma transferência.

De seguida encontram-se algumas estatísticas que podem ser medidas neste protocolo:

- *Round Trip Time (RTT)*;
- Velocidade do primeiro Link de um Nodo;
- *Jitter*;
- Ciclos do Bloco *Transfer*;
- *Datagramas* Esperados;
- *Datagramas* Recebidos;
- Percentagem de Drops;
- Percentagem de Drops por ciclo do Bloco *Transfer*;
- Duração da Transferência;
- Duração dos Blocos *Transfer*;
- Velocidade do Protocolo em bytes/segundo
- Velocidade dos Blocos *Transfer* em bytes/segundo

Neste contexto, o objetivo destas medições é permitir que o programa possa calcular 2 parâmetros bastante importantes numa transferência :

- Velocidade da transferência em *Datagramas por Segundo (DPS)*;
- Intervalo de tempo que pode ser considerado como 1 *Timeout*.

Estes 2 parâmetros são os mais importantes, pois são os únicos que têm influência na interação do programa com a Rede.

3.3.1 *Datagramas por Segundo*

Este parâmetro, como o próprio nome indica, determina a velocidade a que o Transmissor irá tentar transmitir os *Datagramas CM* correspondentes à transferência em causa. É importante calcular este valor com precisão, porque este pode levar a grandes perdas de *Datagramas* na Origem, na Rede e no Destino se for demasiado elevado, ou fazer com que a Largura de Banda disponível entre os 2 Nodos não esteja a ser aproveitada devidamente. Como este protocolo foi desenhado para lidar com falhas na Rede e grandes quantidades de perdas de *Datagramas*, é sempre melhor favorecer um *DPS* mais elevado e arriscar algumas perdas que podem ser recuperadas num ciclo posterior que não aproveitar a Largura de Banda disponível e arriscar não conseguir transmitir os dados num contacto entre 2 Nodos que seja temporário.

Para o diagnóstico e cálculo deste parâmetro temos as seguintes medições:

- Velocidade do primeiro Link de cada Nodo interveniente na troca de informação;
- *Jitter*;
- Ciclos do Bloco *Transfer*;
- *Datagramas* Esperados;
- *Datagramas* Recebidos;
- Percentagem de Drops;
- Percentagem de Drops por ciclo do Bloco *Transfer*.

Uma das fraquezas do *TCP*, nestes ambientes de grandes perdas de *Datagramas*, é o tempo que este demora a conseguir utilizar toda a Largura de Banda disponível no início de uma transferência e após a deteção de uma falha devido a mecanismos como o *Slow Start*, *AIMD* e *Congestion Avoidance* e estes comportamentos não são desejáveis neste Protocolo. Estes mecanismos permitem que o *TCP* eventualmente atinja a Largura de Banda máxima de uma conexão sem a ultrapassar mas fá-lo de uma maneira demasiado passiva, desperdiçando tempo útil de uma conexão que pode ser temporária. Assim sendo, neste protocolo vai ser tomada uma abordagem mais agressiva, marcando como alvo a menor velocidade de ligação de primeiro salto dos Nodos intervenientes numa transferência. Esta velocidade de ligação pode variar ao longo da transferência e se estas alterações forem significativas devem de ser consideradas para a atualização da velocidade da transferência. Com esta abordagem é possível saturar a ligação desde o início, e caso esta seja demasiado elevada apenas resultará em *Drops* na Rede, coisa que este protocolo está preparado para lidar.

Outros fatores muito importantes neste cálculo são as medições de *Datagramas Esperados*, *Datagramas Recebidos* e a consequente Percentagem de *Datagramas* não recebidos (Percentagem de *Drops*). Como já foi referido, podem haver valores elevados de *Drops* durante a utilização deste protocolo, especialmente com a abordagem agressiva tomada anteriormente, por isso é vital que estas medidas sejam tomadas em conta para o ajuste da velocidade em *DPS* final.

Mesmo com esta lógica é possível existir um problema com esta forma de cálculo. Da maneira que este cálculo está a ser feito, apenas pode ser realizado antes do início de cada ciclo do Bloco *Transfer*, pois é no fim destes que se pode calcular a Percentagem de *Drops* observada. Com isto podemos imaginar uma situação em que uma transferência perde conexão nos momentos iniciais e desencadeia um *Timeout*, levando o Recetor a detetar uma elevada Percentagem de *Drops* indevidamente. Quando esta for recomeçada, o Recetor vai observar uma grande perda de *Datagramas* no ciclo anterior e reduzir os *DPS* sem motivos para tal, mantendo esta velocidade reduzida até o início do próximo ciclo, que pode não vir a ocorrer devido a esta velocidade reduzida que favorece uma transmissão sem perdas. Isto faz com que seja desperdiçado tempo que pode ser limitado e, por isso, é necessário introduzir uma medição que detete o estado atual da Rede em *Tempo Real*. Dependendo da implementação do Transmissor, é possível medir o *Jitter* da Rede em uso. De forma simplificada, *Jitter* trata-se do desvio de tempo em relação a um padrão de chegada de *Datagramas*, como ilustrado na Figura 27.

Imaginando um Transmissor que envia de 1 *Datagrama* a cada x milissegundos, se o Recetor receber *Datagramas* nesse padrão, o *Jitter* da Rede em uso é de 0 milissegundos, indicando que a Rede não se encontra sobre *stress* e, se possível, o *DPS* deverá ser aumentado. Caso o Recetor detectar a chegada de um *Datagrama* a cada $x + 100$ milissegundos, este observou um *Jitter* de 100 milissegundos e pode concluir que a Rede em uso está congestionada e os *DPS* devem de ser reduzidos.



Figura 27: Exemplo de *Jitter* quando uma Rede se encontra sobre *stress*

Porém, esta medição sofre com o aumento de perdas de *Datagramas* e espera que estes cheguem por ordem para ser possível fazer as medições corretamente. Isto simplesmente não é possível neste protocolo, mas poderá ser implementada uma versão de *Jitter* que não meça estes desvios de forma correta mas que consiga indicar quando a Rede se encontra congestionada ou não.

3.3.2 Timeout

Este parâmetro é importante pois determina o intervalo de tempo que o Recetor deverá de passar sem receber um *Datagrama* para fazer o pedido de retransmissão de *Datagramas CM* em falta. Tal como o *DPS*, é necessário ter um valor balanceado, pois se isso não acontecer vão começar a ser ativados mecanismos de resposta a falhas da Rede que vão começar a induzir estados inesperados no Protocolo. Isto acontece especialmente quando este intervalo de tempo é significativamente inferior ao que deveria, pois pode induzir uma situação de *Timeout* indevida, e consequentes pedidos de *Datagramas CM* em falta, também de forma indevida durante a transmissão de *Datagramas CM*, levando ao fecho do Bloco *Transfer* atual no Recetor e ao início de um novo em que vão ser recebidos mais *Datagramas* que aqueles que são esperados, levando a uma medição de Percentagem de *Drops* negativa, influenciando o cálculo dos *DPS* do ciclo seguinte. Mas este fenómeno também pode ser observado quando um dos Nodos da transferência se movimenta na Rede, pois quando a conexão é reestabelecida, o Transmissor pode não ter acabado de enviar todos os *Datagramas CM* quando recebe os pedidos de retransmissão, reenviando *Datagramas* duplicados indevidamente.

Infelizmente apenas 2 das medições indicadas anteriormente podem ajudar na determinação da duração deste *Timeout*:

- *Round Trip Time (RTT)*;
- *Jitter*.

Sendo que o *RTT* indica o intervalo de tempo entre a emissão de um *Datagrama* e a receção da resposta a esse *Datagrama*, pode ser usado como valor base para a deteção do *Timeout*, pois colocará este valor dentro da mesma ordem de grandeza do tempo de trânsito dos *Datagramas*. Mesmo assim é imperativo que o valor deste intervalo de tempo seja superior, pois se a Rede em uso sofrer com congestionamento podem existir atrasos ou um aumento deste *RTT* que desencadeiem os mesmos mecanismos de recuperação de falhas na Rede indevidamente. Para este efeito também pode ser usado o *Jitter* que tem como objetivo detetar estes momentos de congestionamento.

Mesmo com este valor estabelecido, deverá de ser utilizado um sistema de *Timeouts* reativo, ou seja, conforme se vão acumulando *Timeouts* consecutivos, o intervalo de tempo que se espera por um *Datagrama* deverá de aumentar até chegar a um valor máximo estipulado *a priori*. Desta forma não são atingidos tantos *Timeouts* consecutivos, atrasando o envio dos *Datagramas MCID* dando uma maior probabilidade que as transferências consigam ser recuperadas após uma movimentação de um dos Nodos. De forma a não serem feitos demasiados pedidos de *Datagramas* em falta, os *Datagramas MCID* não deverão de ser transmitidos em todos os *Timeouts*, mas deverão de ser espaçados.

3.4 SEGURANÇA

A Segurança é dos pontos mais importantes em transmissões de Dados especialmente quando é feita sobre uma Rede hostil. Este facto dá-se pois nestas Redes podem existir Nodos que apenas têm a intenção de interagir de forma maliciosa com os outros Nodos presentes nela, realizando ataques, desde roubo de informação, personificação, como ataques de *Man in the middle*, até à integridade de dados. Por estes motivos é necessário tomar medidas para que se consiga assegurar que a comunicação entre 2 Nodos consiga ser realizada de forma segura e confiável, mesmo que esta venha com uma penalização de *Performance* do Protocolo devido à introdução de *Overhead* relativo às medidas de Segurança.

Como o Protocolo se trata de um Protótipo, e de uma experiência, não serão tomadas medidas de forma a assegurar a segurança de uma transmissão de Dados, pois este assunto é complexo e existem vários caminhos que podem ser tomados neste sentido que alterariam o funcionamento do Protocolo de formas diferentes e que possuiriam vantagens e desvantagens que teriam de ser discutidas individualmente e não haveria tempo para o realizar da forma mais correta. Outros motivos para que este assunto não seja abordado a fundo neste documento são as diferentes necessidades dos programas que viriam a utilizar esta *Framework* e os inevitáveis avanços nos métodos de segurança e ataques usados que iriam tornar a segurança implementada neste Protocolo obsoleta. Relativamente ao primeiro, é impossível de criar um Sistema de Segurança da informação trocada neste Protocolo que seja útil para qualquer aplicação que possa vir a usá-lo, pois, qualquer que seja o nível de Segurança e a dificuldade dessa mesma, existirão sempre aplicações que não necessitam de tanta Segurança, e que a presença desta tem um grande impacto no funcionamento da aplicação, e outras que necessitam de Segurança que seja bastante mais resistente à intrusão ou que seja mais flexível (entre outros) e que a presença desta apenas seja um bloqueio para outras implementações. Em relação ao segundo ponto, é impossível que um Sistema de Segurança seja considerado seguro eternamente, sendo que, desde o momento que este é implementado, é necessário gastar recursos para que este seja atualizado, corrigido quando forem encontradas vulnerabilidades e, eventualmente, trocado por outro que apresente mais benefícios que o atual. É claro que este último ponto não deve de ser a razão pela qual não sejam implementados Sistemas de Segurança neste Protocolo, pois seguindo essa lógica não seriam implementadas quaisquer medidas de Segurança, mas isto apenas se trata de um protótipo e o tempo dedicado a este ponto poderia ser gasto em vão.

Apesar disso, nada impede que se façam recomendações para possíveis Sistemas de Segurança a utilizar.

3.4.1 Segurança da informação

Primeiramente é necessário compreender o que se quer dizer quando se fala de Sistemas de Segurança e Segurança da Informação a ser trocada. Existe 4 princípios que não devem de ser quebrados para que se possa dizer que a Informação está Segura [34]:

- Autenticação;
- Integridade;
- Confidencialidade;
- Disponibilidade;

Autenticação diz respeito à capacidade de identificação de Nodos quando se realiza uma troca de Dados, isto é, os Nodos intervenientes numa transferência têm que ser capazes de identificar que a informação que estão a receber tem como Origem o Nodo com quem se encontram em comunicação e serem capazes de identificar a informação que estão a transmitir como tendo Origem neles próprios.

Integridade diz respeito à não alteração dos Dados após a sua transmissão, ou seja, o Nodo Recetor tem que ser capaz de detetar que os dados recebidos sofreram alterações e o Nodo Transmissor tem que ser capaz de marcar tais dados de forma que seja possível detetar essas mesmas alterações potencialmente feitas por Nodos Maliciosos.

Confidencialidade diz respeito à capacidade de esconder a informação de tal forma a que os únicos Nodos que consigam obter informação útil sejam os Nodos Transmissores e Recetores, significando que qualquer outro Nodo, que tenha recebido essa informação de forma indevida ou maliciosa, não consiga saber o que foi trocado entre os 2 Nodos de uma transferência, mesmo que receba a informação na sua totalidade.

Disponibilidade diz respeito à capacidade de obtenção da informação a qualquer momento, isto é, determinada informação não deve de ser mantida numa infraestrutura que, quando atacada, deixa de ser possível aceder à informação em questão. Este último ponto não vai ser explorado, pois diz mais respeito à organização de infraestruturas, replicação de informação e medidas de Seguranças na Rede (como *Firewalls* e *Intrusion Prevention Services (IPS)*) o que não vai de encontro ao âmbito deste Projeto.

Assim sendo, é necessário utilizar mecanismos para assegurar que cada um destes tópicos seja satisfeito utilizando métodos criptográficos.

3.4.2 Métodos Criptográficos

Em termos de criptografia, existem duas principais abordagens neste tema de Segurança de Dados:

- Sistemas de Chaves **Simétricas**;
- Sistemas de Chaves **Assimétricas**.

Ambos Sistemas têm o mesmo objetivo, tornar Informação Útil (*Plaintext*) em dados cifrados (*Criptograma*) usando uma chave que, posteriormente, podem ser reconvertidos na Informação Útil que inicialmente se tinha. A diferença entre estes dois tipos de Sistemas, para além de como funcionam internamente, são as chaves que usam. Sistemas de Chaves **Simétricas** usam a mesma chave para para converter a Informação Útil em *Criptogramas* e para converter estes em Informação Útil novamente, enquanto os Sistemas de Chaves **Assimétricas** usam uma Chave **Pública** para transformar a Informação Útil em *Criptogramas* e uma Chave **Privada** para realizar a operação contrária.

Esta diferença, faz com que existam duas filosofias de Segurança na Troca de Informação entre dois Nodos, pois cada uma apresenta benefícios e contrapartidas. Enquanto um Sistema de Chaves **Simétricas** é mais simples e apresenta uma maior performance de Dados processados por segundo devido a essa simplicidade, apresenta um problema que está diretamente ligado ao problema que se tinha no início. Se é usada a mesma Chave **Simétrica** para Cifrar e Decifrar, como é que seria possível gerar uma nova chave e transmitir esta para outro Nodo sem que esta corra o risco de ser interceptada na sua transmissão? Seria necessária a existência de uma outra Chave, já possuída pelos dois Nodos, para Cifrar o conteúdo desta e para que possa ser transmitida sem correr qualquer risco, mas como é que esses dois Nodos conseguiram partilhar essa Chave entre eles sem que esta corresse riscos de ser alterada? E se já possuíam uma Chave que ambos conheciam, porque é que necessitam de outra Chave? Claramente a desvantagem destes Sistemas é a partilha da Chave **Simétrica** entre dois Nodos através da Rede, logo esta transmissão, sem o uso de outro mecanismo de Segurança, não consegue ser garantida como segura.

Por outro lado, os Sistemas de Chaves **Assimétricas** foram desenhados para que esta troca de Chaves possa ser feita sobre uma Rede hostil sem que a transmissão de chaves de forma desprotegida comprometa a eventual Troca de Informação futura. Estes Sistemas conseguem garantir isto, pois cada Nodo possui duas Chaves, uma Chave **Pública** que pode transmitir a todos os Nodos com quem tiver contacto para que estes consigam cifrar toda a informação que lhe irão transmitir, e uma Chave **Privada** que matem em segredo e que usa para reverter o processo feito pelo Nodo transmissor da Mensagem. Em contra partida, no geral estes Sistemas são bastante mais lentos no processamento de Dados por segundo comparativamente aos Sistemas de Chave **Simétrica**.

Assim, de forma a tentar aproveitar os benefícios de cada um destes Sistemas e tentar fugir às suas desvantagens, sugere-se o uso de um Sistema *Pretty-Good-Privacy (PGP)* [35][36]. Neste Sistema ilustrado na Figura 28, inicialmente o Recetor da informação transmite a sua Chave **Pública** de um Sistema de Chaves **Assimétricas** para o Transmissor. Uma vez recebida esta Chave **Pública**, este irá gerar uma Chave de um Sistema de Chaves **Simétricas** e

vai Cifrar a informação a transmitir com ela, sendo que irá Cifrar a esta Chave *Simétrica* com a Chave *Pública* que recebeu anteriormente e transmitir toda esta informação para o Recetor. Este, com a informação que acabou de receber, poderá usar a sua Chave *Privada* para Decifrar a Chave *Simétrica* e usar esta última para Decifrar os *Criptogramas* relativos à Informação Útil.

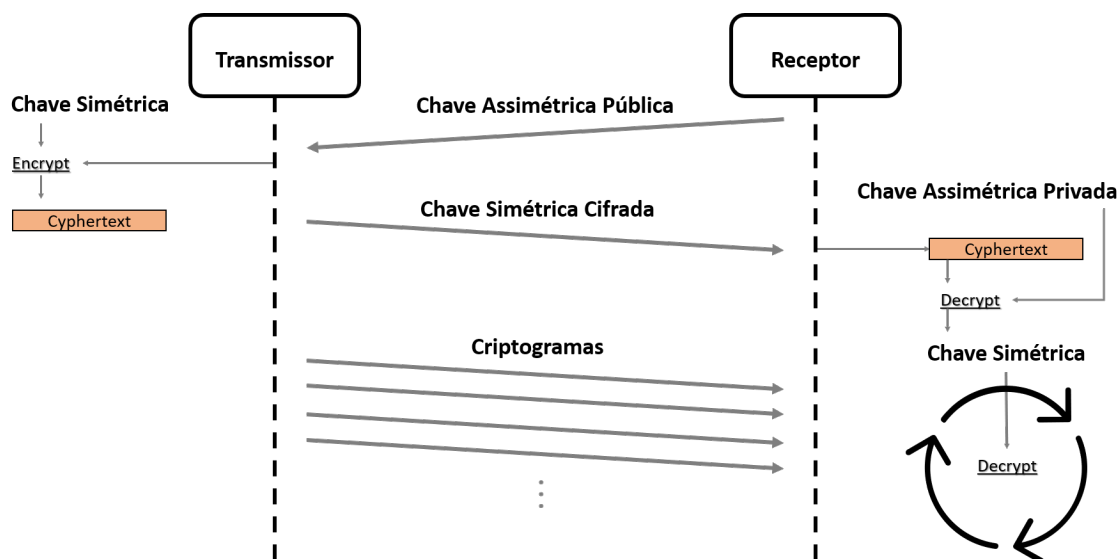


Figura 28: Fluxo de Mensagens e ordem de Operações num Sistema de PGP

Embora este mecanismo faça com que os Dados transmitidos não possam ser Decifrados por um Nodo que não possua a Chave *Simétrica* correta, não impede que os *Criptogramas* sejam adulterados. Assim é possível afirmar que a Confidencialidade é atingida, mas a Autenticação e a Integridade ainda são problemas que têm que ser mitigados.

Para resolver estes problemas é possível recorrer ao mecanismo *Message Authentication Code (MAC)* [37][38] que visa mitigar estes mesmos problemas. A utilização de *MAC* permite que o Recetor de Informação verifique a Autenticidade e Integridade da informação recebida utilizando uma Chave que ambos possuam. Este *MAC* é calculado ao mesmo tempo que a Informação é Cifrada por isso podem ser usadas as mesmas Chaves para ambos, dependendo da ordem que estes são calculados. Existem 3 formas de o fazer (Figura 28):

- *Encrypt-then-MAC*
- *Encrypt-and-MAC*
- *MAC-then-Encrypt*

Estas 3 formas variam na ordem em que cada processo é feito, sendo que alguns modos apresentam fraquezas em comparação a outros. De forma geral deve de ser usado o último modo *MAC-then-Encrypt*.

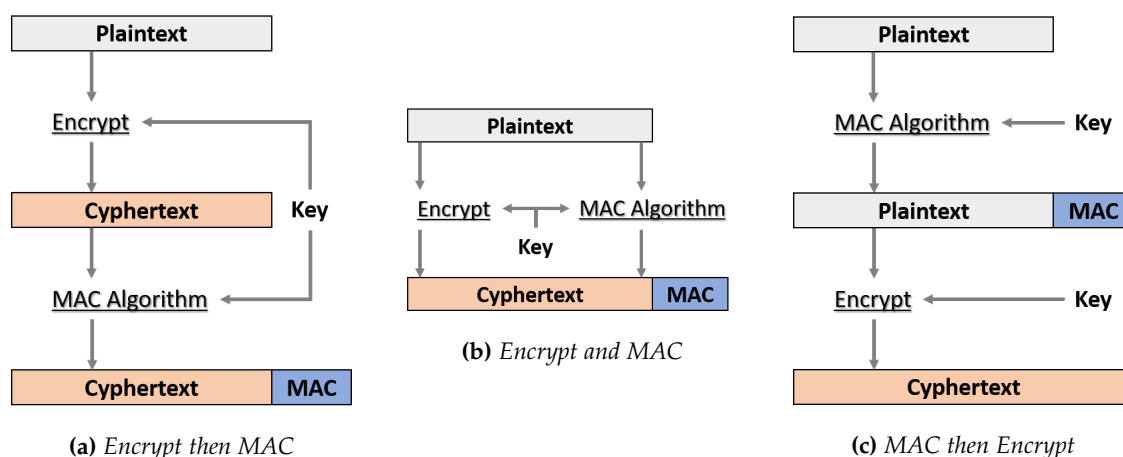


Figura 29: Possibilidades de ordem de realização do Algoritmo de MAC e Cifrar Dados

Caso apenas os dois Nodos que estejam a realizar a Transferência conheçam a Chave utilizada, este mecanismo atinge os padrões de Autenticação e Integridade pois este utiliza uma *Função de Hash* que utiliza a Chave partilhada pelos Nodos intervenientes para criar uma *Hash* representativa da Informação a ser trocada entre eles que será enviada juntamente com a informação. No momento de receção, o Recetor pode usar a mesma chave para realizar o mesmo algoritmo de criação de MAC sobre a Informação Útil recebida e compará-lo com o MAC presente na Mensagem. Caso sejam iguais é confirmada a Autenticidade e Integralidade da Informação, caso contrário esta informação foi adulterada e deve de ser dada como perdida. Se houver um outro Nodo maliciosos que, por qualquer motivo, possua a Chave que está utilizada para Cifrar a Informação e criar o MAC, este mecanismo já não consegue garantir tais padrões de Segurança, pois o Nodo malicioso pode intercepta o *Criptograma*, alterar o seu conteúdo, e criar o MAC correspondente de forma válida sem que seja detetável pelo Recetor.

Mesmo usando estes mecanismos de Segurança, é possível que exista um Nodo malicioso que esteja a intercepta as comunicações entre os Nodos e esteja a fazer uma "ponte" na transmissão de informação, fazendo-se passar por Recetor e Transmissor ao mesmo tempo de forma a enganar os Nodos que estão a tentar transmitir informação entre eles de forma legítima (Figura 30). A este tipo de ataques dá-se o nome de *Man in the Middle Attacks* [39], pois o Nodo malicioso vai-se encontrar no meio dos dois Nodos a roubar a Informação que está a ser trocada entre eles sem que nenhum dos Nodos se aperceba disso.

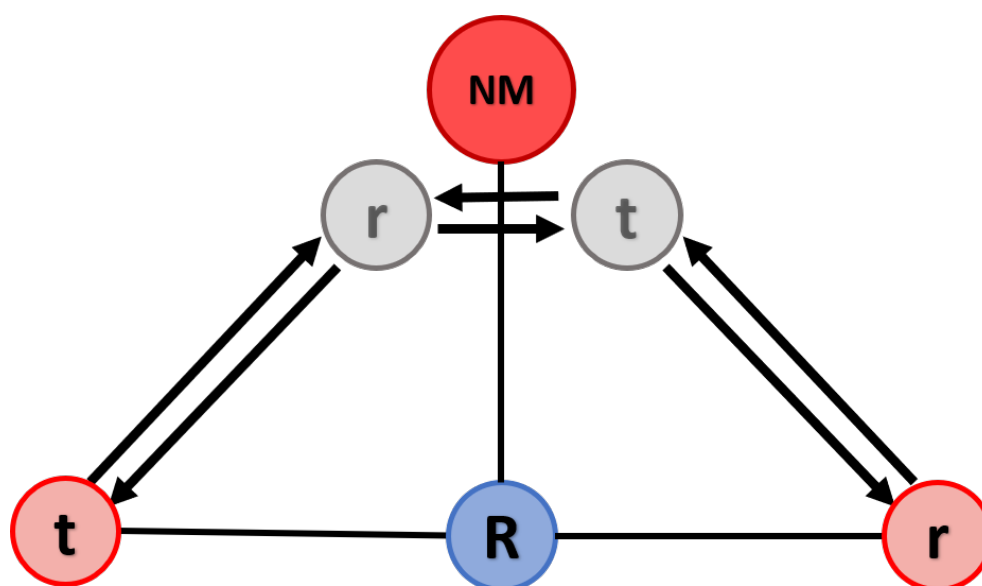


Figura 30: Exemplo de um Ataque Man in the Middle

Para a resolução deste problema é possível utilizar *Certificados*. Estes *Certificados* servem para tentar dar um certo nível de confiança a uma dada informação para que esta possa ser usada como sendo legítima. Nestes ataques de *Man in the Middle*, o Nodo malicioso faz-se passar por Recetor e Transmissor ao mesmo tempo de forma aos outros Nodos não se aperceberem que a sua transmissão está a ser interceptada. Os *Certificados* entram aqui para tentar certificar as Chaves *Públicas* usadas no Sistema *PGP* descrito a cima, dando confiança aos Nodos legítimos que estão a comunicar entre eles de uma forma segura. Estes *Certificados* são *Assinaturas Digitais* feitas por uma Organização de Confiança que garante que um pedaço de Informação é legítimo através da utilização de um Sistema de Chaves *Assimétricas*. Para um *Certificado* ser validado por um Nodo, este tem que "confiar" na Organização que emitiu esse *Certificado*, verificando a validade da *Assinatura Digital* desse certificado com a Chave *Pública* da Organização que o emitiu. Caso não conheça essa organização, pode ser utilizada uma cadeia de *Certificados* para se atingir um estado onde o Nodo "confia" numa determinada Organização que consegue validar e "confiar" na Organização em questão, validando assim o primeiro *Certificado*.

Um ponto que merece mais atenção devido aos princípios deste Protocolo é como se cifram os Dados. Caso estes sejam Cifrados na sua totalidade, o Recetor apenas vai conseguir Decifrar os mesmos quando receber todos os *Criptogramas*, instalando uma dependência de Dados entre os mesmos, o que é indesejável neste Protocolo. Apesar da alternativa de realizar uma Cifra por *Datagrama* ser mais intensiva e requerer mais recursos é a que deverá de ser tomada.

Esta sugestão pode não ser a mais apropriada, mas encaixaria bem no Protocolo desenvolvido, sendo que não necessitaria de muitas alterações para ser incorporada.

3.5 DETECÇÃO E CORREÇÃO DE ERROS

Um ponto interessante é a deteção e possível correção de erros que podem ocorrer na transmissão da Informação de um Nodo para o outro. Estes erros podem acontecer por vários motivos desde interferências durante a comunicação ou o congestionamento do meio de comunicação, defeitos ou danos no *Hardware* dos Dispositivos ou apenas uma elevada distância entre os 2 Nodos que comunicam através de uma Interface *Wireless*, e, de forma a agrupar e simplificar estes erros, têm o nome de *Bit Flips*.

Como foi discutido anteriormente, se forem usados mecanismos de verificação de Integridade para efeitos de segurança, será necessário entrelaçar estes, pois não é possível distinguir um *Bit Flip* de uma tentativa de alteração da Mensagem.

Este ponto é interessante pois, em certos casos de uso deste Protocolo, pode ser vantajoso sacrificar a *Performance* de Transmissão de Dados Úteis com a utilização de *Error Correction Code (ECC)*, como *Hamming Codes*, que introduzem redundância nas mensagens enviadas, mas são capazes de detetar e corrigir estes *Bit Flips* dependendo dos níveis de redundância introduzida [40], enquanto noutros casos esta funcionalidade não faz qualquer sentido.

No geral, quanto maior for o *Delay* de uma Transmissão de Dados entre 2 Nodos, maior será o benefício desta funcionalidade, pois um pedido de retransmissão de uma Mensagem que sofreu *Bit Flips* pode ser mais demorada que a Transmissão dessa mesma informação com redundância suficiente para que seja possível a sua correção no Nodo Recetor. Se o ambiente de comunicação for muito suscetível a *Bit Flips* pode mesmo ser impossível a realização de uma Transferência sem a utilização de um *ECC*, sendo que o pedido de retransmissão de informação seria um mecanismo de último recurso que apenas seria utilizado quando não fosse possível recuperar de um *Bit Flip*.

Isto é o caso das Redes *DTN* onde as distâncias entre os 2 Nodos podem ser extremamente grandes bem como os *Delays* de tal forma que pode não fazer sentido não utilizar um *ECC*.

No entanto, em redes como *Ad Hoc*, pode depender da aplicação que está a utilizar o Protocolo, bem como a Arquitetura desta, como os Nodos se movimentam e como a própria distribuição da informação é feita. Noutras Arquiteturas que sejam mais convencionais, estáticas e que não sofram de grandes *Delays*, não faz qualquer sentido a utilização destes *ECC* pois a Rede é estável o suficiente para ser feito um pedido de retransmissão de Informação que sofreu destes erros.

Mesmo que seja desejada a integração de uma implementação de um *ECC* neste Protocolo, é necessário ter em consideração que existe um problema de baixo nível com isto. Dependendo do Sistema Operativo que está a ser usado no Dispositivo que está a usar

este Protocolo, este pode decidir descartar um *Datagrama* caso verifique que o *Checksum* do cabeçalho do Protocolo da Camada de Transporte não corresponde ao que calculou, dando este *Datagrama* como perdido automaticamente e não sendo possível uma correção como foi referido.

CONCEÇÃO DA FRAMEWORK

Neste Capítulo são feitas as primeiras divisões em Categorias do Protocolo de Transferência de Dados especificado no Capítulo 3, distribuindo responsabilidades pelas Categorias conforme as funções de cada uma delas. Também são apresentadas otimizações em algumas Categorias que reduzem a necessidade de recursos para a execução deste Protocolo. São introduzidas as noções de *Knock* e a sua utilidade na descoberta de Nodos e Transferências *Multipath* e *MultiTransmitter*. Por fim são abordados temas específicos do Sistema de Armazenamento como a sua Estrutura e Compressão de Dados.

4.1 CONTROLADOR

Utilizando este Protocolo resistente a falhas na Rede e com capacidades de recuperar de tais falhas, é possível construir um controlador que consiga realizar transferências em ambientes que anteriormente seriam considerados adversos. Visto a natureza deste Protocolo, este controlador vai ser orientado a transferências de apenas 1 Ficheiro, sendo que, tendo 2 ou mais transferências entre os 2 mesmos Nodos com papéis fixos, será necessário recomeçar o Protocolo com o Bloco *Handshake*.

De forma a simplificar a descrição do controlador, serão designados os 3 módulos independentes para cada uma das funções essenciais que este terá que implementar:

- módulo de Receção de Dados (*Listener*)
- módulo de Transmissão de Dados (*Sender*)
- módulo de Controlo das mensagens *Knock* (*KnockManager*)

Cada um destes módulos, para além de terem que implementar funções que este protocolo requer para ser verdadeiramente resistente a falhas da Rede, deverá de possibilitar a realização de múltiplas transferências com múltiplos Nodos num determinado momento. Um ponto importante é que a Receção de uma transferência não deverá de impedir o Envio de outra e vice-versa.

Outro ponto importante neste passo da especificação é o de Controlo dos *Network Interface Card (NIC)s*. Numa altura em que os dispositivos eletrónicos têm a capacidade de comunicar com uma Rede com múltiplas Interfaces distintas, seria útil ter o controlo de quais são os *NICs* que possuem conexão, quais são os seus Endereços, e tratar cada um deles individualmente.

4.1.1 Contolo de *NICs*

Como foi referido anteriormente, hoje em dia é comum a existência de dispositivos com múltiplas Interfaces para comunicar com uma Rede. O facto de existir esta flexibilidade pode trazer vantagens como a conexão de um Dispositivo a diversas Redes distintas, abrindo um leque de possibilidades no que diz respeito a acessibilidade a outros dispositivos. Estando ciente destas possibilidades, o controlador tem que ser capaz de tirar partido delas.

Esta componente também é importante pois é ela que vai detetar qual é o estado de conexão dos múltiplos *NICs*, desde os momentos em que ganham e perdem conexão, às mudanças de Endereços e, conforme o acontecimento, vai interagir com os outros módulos do Controlador para que sejam tomadas as medidas necessárias.

Ter este nível de controlo traz outras vantagens que não são óbvias inicialmente. Uma delas é a possibilidade de realizar transferências *MultiPath* (Figura 31a) entre 2 Nodos que possuem pelo menos 2 *NICs* conectados a 2 Redes distintas mas que consigam comunicar entre si, fazendo com que o valor teórico da velocidade destas transferências possa ser igual a soma da velocidade das ligações dos 4 *NICs*. Em alusão a esta possibilidade, existe uma outra que segue a mesma abordagem de aproveitamento de várias conexões, mas tira partido da forma como este Protocolo identifica os Dados que estão em falta. A identificação de *Chunks* em falta seguindo uma *Confirmação Negativa* como foi especificado na Secção 3.1.5.1 do Capítulo 3 permite com que sejam realizadas transferências *MultiTransmitter* (Figura 31b), onde 1 Nodo Recetor recebe 1 Ficheiro de múltiplos Transmissores. Assumindo que todos os Transmissores possuem o Ficheiro na sua totalidade, o Recetor pode pedir determinados grupos de *Chunks* em específico a diferentes Transmissores, paralelizando desta forma a Transferência.

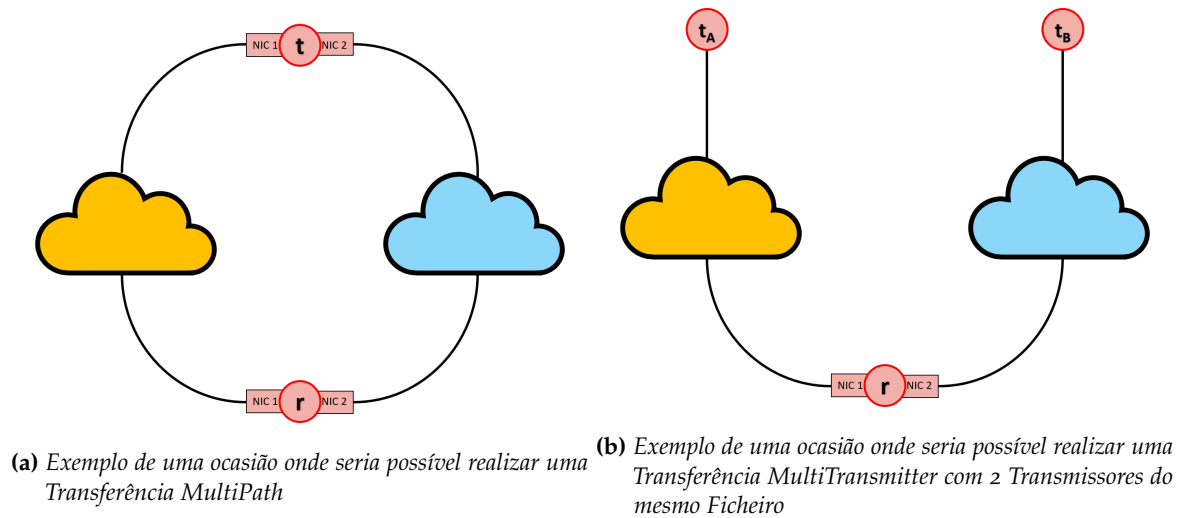


Figura 31: Multipath e MultiTransmitter

Outra possibilidade que se abre quando se tem este nível de controlo é a recuperação de uma transferência *interrompida* utilizando outro *NIC* que possua uma conexão à Rede e que consiga comunicar com o outro Nódo com quem estava a trocar informação inicialmente.

Porém seria necessário fazer alterações ao Protocolo para estas características poderem ser incorporadas e, como não se trata do objetivo principal a ser atingido com este Projeto, não foram ainda finalizadas tais modificações.

Por último, este controlador de *NICs* tem que conseguir gerir a Banda Larga disponível na Ligação que tem, distribuindo-a pelas transferências cativas da forma mais adequada. Aqui podem ser tomadas várias decisões, desde apenas dar a Banda Larga a uma transferência, que iria contra a ideia de transferências simultâneas, a distribuição equivalente por todas elas, distribuição equivalente apenas por aquelas que se encontram num estado *activo*, distribuição que favorece as que começaram primeiro, priorizar as que se encontram perto do fim ou até ter em conta as perdas que se estão a observar. Existem múltiplas opções que ficam à escolha da implementação.

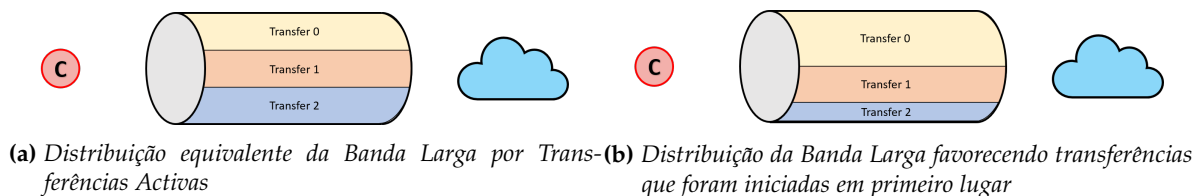


Figura 32: Exemplos de Distribuição da Banda Larga disponível

O Controlo dos vários *NICs* que um dispositivo possui pode trazer um aumento na velocidade a que uma transferência pode ser realizada e um aumento na resiliência do

Protocolo devido aos múltiplos caminhos que podem estar disponíveis para a realização de uma transferência.

4.1.2 Listener

Este módulo do Controlador tem a responsabilidade de tratar da receção de Dados de uma transferência, sendo que vai fazer a ligação ao *Sistema de Armazenamento*.

A maior responsabilidade que este módulo tem é a de implementar o Protocolo anteriormente descrito, sendo que terá que gerir os recursos disponíveis para a execução do mesmo, dando utilidade à informação que lhe é disponibilizada pelas medições de estatísticas. Isto é importante porque, juntamente com a informação que recebe referente às capacidades do Nodo Transmissor, este terá que decidir os recursos que serão utilizados por ambos os Nodos. Uma responsabilidade acrescida é a ligação que terá que gerir com o *Sistema de Armazenamento*, pois terá que saber que Dados já são guardados na totalidade, parcialidade ou que ainda não possui qualquer parte para poder tomar as decisões corretas na execução do Protocolo. Para além disso, é este que terá a tarefa de registar *Chunks* recebidos como tal, bem como guardar os mesmos no Dispositivo.

Adicionalmente, terá que manter registo de quais são os *NICs* que estão a ser usados por cada Receção, pois quando receber uma notificação de uma alteração de estado de um destes (por exemplo perda de conexão), vai ter que ser capaz de reagir a esse acontecimento, agindo conforme especificado anteriormente.

Por fim, é neste módulo que se tem que considerar a realização de múltiplas Receções em simultâneo, partilhando recursos onde possível, reduzindo a carga no Dispositivo para não forçar perdas na Origem. Em termos de otimizações futuras, este controlador pode sofrer imensas alterações para tirar partido delas. Uma das maiores otimizações possíveis neste momento é a redução de Portas em uso quando existem múltiplas transferências. Da forma como o Protocolo está especificado, cada Recetor necessita de ter 1 Porta para enviar *Datagramas* de gestão (*TMRI*, *NSU*, *MCID* e *Over*) e p Portas de receção de *Datagramas CM* dependendo dos recursos disponíveis para a receção destes (Figura 33). Desta forma temos:

$$\sum_{j=0}^t (1 + p_j) \text{Portas}$$

Sendo t igual ao número de Transferências a serem realizadas num determinado momento e com $p_j > 1$ o número de Portas usadas na Transferência j para a receção de *Datagramas CM*.

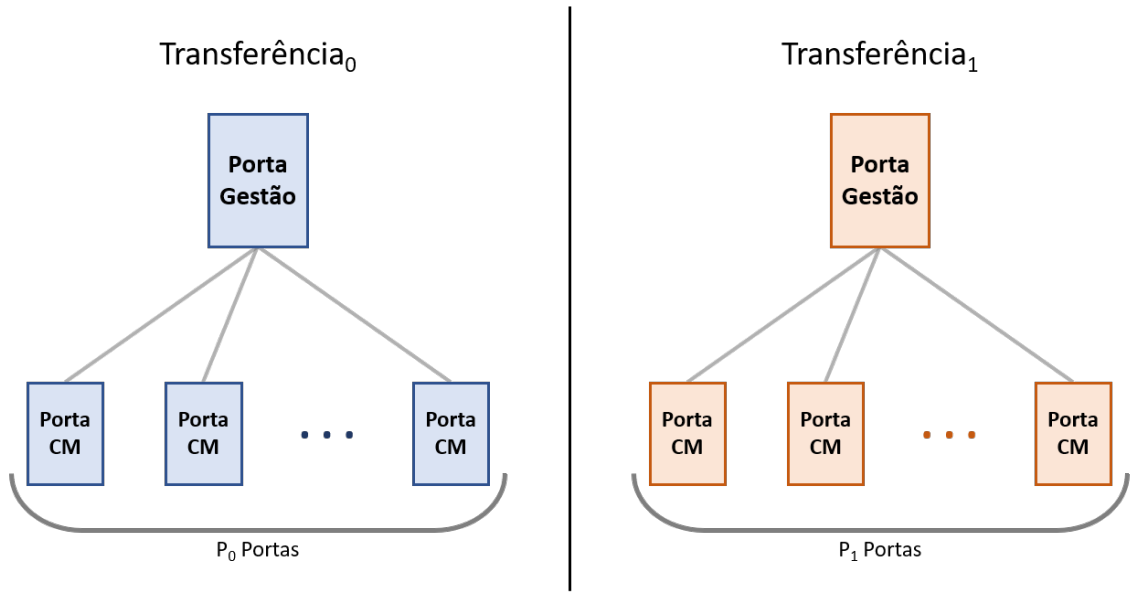


Figura 33: Utilização de Portas conforme é especificado no Protocolo, sem qualquer otimização

Daqui pode-se dizer que existem 2 partes dependentes que podem ser otimizadas independentemente. A primeira otimização é a utilização da mesma porta de gestão para todas as Transferências, reduzindo o seu crescimento linear para 1 (Figura 34). Este agrupamento de Portas não tem impacto na performance geral de cada uma das Transferências, pois o fluxo de *Datagramas* nestas comparativamente ao Fluxo de *Datagramas* nas Portas CM é insignificante e este agrupamento não deverá de ser observável. Apenas poderá existir um impacto a este nível se houvesse um número elevado de Transferências a serem realizadas em simultâneo, mas, nesse ponto, existiriam outros problemas mais importantes que necessitariam de mais atenção. Assim passam a ser utilizadas:

$$(1 + \sum_{j=0}^t p_j) \text{Portas}$$

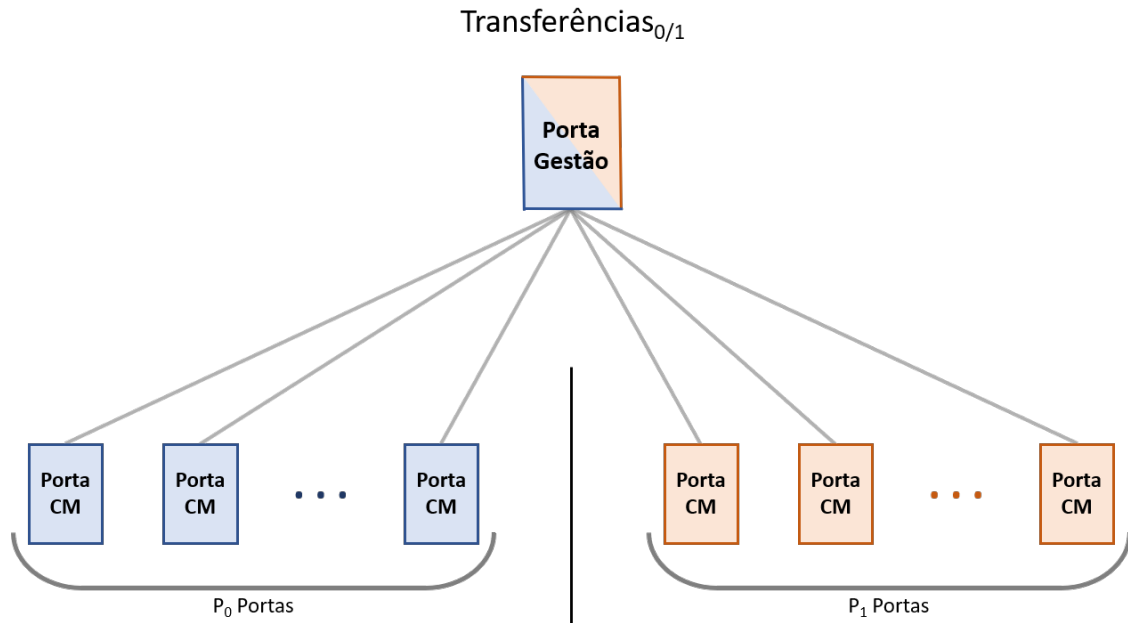


Figura 34: Utilização de Portas conforme é especificado no Protocolo, mas é usada uma única Porta de Gestão para todas as Transferências a serem realizadas

A segunda otimização, referente às Portas de receção de *Datagramas CM*, é um pouco mais complicada de concretizar e é dependente da Performance que o código desenvolvido apresenta. Como foi definido inicialmente, uma transferência necessita de Portas exclusivas para a receção destes *Datagramas* para que seja possível produzir código que seja mais eficiente a retirar os *Datagramas* do *Buffer* de receção de *Datagramas* que o Sistema Operativo disponibiliza e também para o aumento do número destes *Buffers*. No entanto, com a realização de Transferências em simultâneo, este aumento do número de Portas em uso vai sobrecarregar o *CPU* do Dispositivo. Daí tira-se que a utilização destas portas tem que ser controlada para não criar esta situação de sobrecarga. O que pode ser conseguido com o aumento da eficiência a retirar os *Datagramas* dos *Buffers* de tamanho controlado pelo Sistema Operativo. Esta eficiência ajuda o controlo do número total de *Portas CM*, pois permite que se mude a abordagem de criação desta. Passa a ser possível ter d Portas a executar este código otimizado vão conseguir retirar mais *Datagramas* dos *Buffers* que aqueles que conseguem chegar através do *NIC* em uso, sendo possível apenas usar essas d Portas para todas as Transferências a serem realizadas por um determinado *NIC*. Isto também faz com que estas portas sejam fixas, removendo o processo de alocação de portas por cada nova Transferência (Figura 35).

Com esta mudança de filosofia na criação de Portas por Transferência para a criação de Portas necessárias para saturar um *NIC*, temos que o número de Portas em uso serão:

$$(1 + d_{nic})Portas$$

Comparando o número inicial de Portas necessárias quando existem múltiplas transferências a usar vários *NICs* com a mesma situação após as otimizações temos:

$$\sum_{n=0}^{nics} \left(\sum_{j=0}^{t_n} (1 + p_j) \right) Portas$$

Onde *nics* é o número total de *NICs* em uso e t_n o número de transferências a serem realizadas no *NIC* *n* no momento da medição e p_j o número de Portas *CM* em uso na transferência *j*.

$$\sum_{n=0}^{nics} (1 + p_n) Portas$$

Onde *nics* é o número total de *NICs* em uso e p_n o número de portas em uso pelas transferências que usam o *NIC* *n*, que tem como limite superior o número de portas que saturam o *NIC* *n*, no momento da medição.

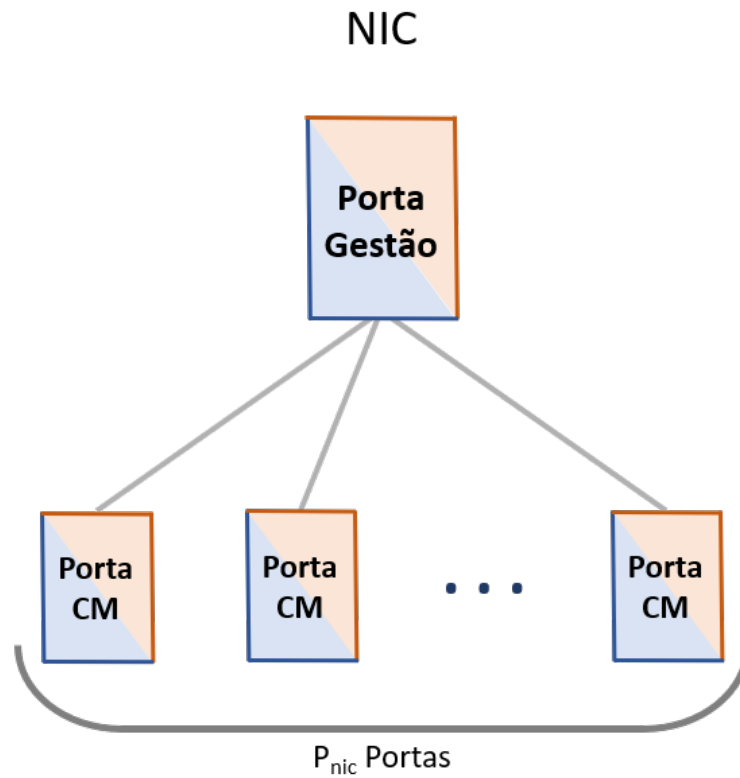


Figura 35: Agrupamento das portas de recepção de Datagramas, tornando-se independente do número de Transferências e reduzindo a carga no *CPU* do Dispositivo

Porém, se esta última otimização for implementada, o processo de medição de Estatísticas referentes a cada uma das Transferências a serem realizadas tornar-se-ia mais complicada,

pois é necessário fazer um passo extra de identificação em *Tempo Real* dos *Datagramas* CM recebidos para que estas possam ser realizadas. Outra forma de abordar este problema seria utilizar uma estrutura de dados para registar a chegada de cada um dos *Datagramas* e abrir a possibilidade de processamento de dados mais tarde, mas qualquer uma das abordagens trás um acréscimo de utilização de recursos, o que dificulta ainda mais a segunda otimização.

4.1.2.1 Métodos

Para interação com este módulo deverá de ser implementado um método que permita a seleção de um *NIC* e de uma Porta pela qual se deverão receber os Dados.

4.1.3 Sender

Este módulo do Controlador tem a responsabilidade de tratar da emissão de Dados de uma Transferência, e, de forma semelhante ao *Listener*, vai fazer a ligação entre os pontos relativos à Transmissão desses Dados e o *Sistema de Armazenamento*.

Tal como o *Listener*, o *Sender* tem como maior responsabilidade a emulação do Protocolo descrito anteriormente, participando nas Transferências como o Transmissor. Também em semelhança ao *Listener*, este terá que gerir os recursos que o dispositivo dispõe, mas não é este que toma a decisão final de quantos recursos devem de ser utilizados, mas sim o *Listener* do Recetor da Transferência que, no momento do *Handshake*, tem toda a informação necessária relativa aos recursos que o *Sender* diz ter disponíveis.

Em termos de interação com o *Sistema de Armazenamento*, o *Sender* apenas tem a responsabilidade de obter os *Chunks* corretos para os transmitir durante uma Transferência. Adicionalmente, antes de começar uma Transferência, pode verificar a integridade dos *Chunks* recalculando a *Hash Identificadora* dos *Chunks* que possui e comparando-a com a corresponde *Hash Identificadora* presente nos Meta Dados desse Ficheiro.

Tal como no *Listener*, se a filosofia do Protocolo fosse seguida na totalidade, poderia chegar um ponto em que um Dispositivo a utilizar este programa ficaria sem recursos muito rapidamente por causa da alta exigência de Portas deste Protocolo. Assim sendo, também é necessário implementar algumas otimizações, semelhantes às que se implementaram anteriormente para o *Listener*, para o número de Portas ser reduzido.

Assim sendo, visto que o Recetor e o Transmissor partilham a estrutura de organização de Portas para uma transferência, seria possível implementar as otimizações discutidas anteriormente para o *Listener* neste módulo. Embora a agregação das Portas de Gestão das Transferências que decorrem em simultâneo numa Porta de Gestão geral seja possível e reduza a utilização de recursos, a agregação de Portas CM não é uma decisão tão direta como a primeira. Em termos de possibilidade de execução, é uma otimização fazível, mas a complexidade desta é bastante superior há que foi feita no *Listener*. Enquanto um Recetor tem

que realizar a recepção de *Datagramas CM* o mais rapidamente possível independentemente da Largura de Banda usada, o Transmissor tem que conseguir controlar o envio dos mesmos de forma a não ultrapassar a velocidade estabelecida no *Handshake* e a formar padrões que possam ser reconhecidos pelo Recetor para poderem ser medidas estatísticas como o *Jitter*. Como um Dispositivo pode ter várias Transferências onde ele é o Transmissor e a cada Transferência pode ter uma velocidade de Transmissão diferente e independente das outras, seria um ponto complicado de abordar nesta fase. Adicionalmente, para se poder transmitir *Chunks*, é necessário carregá-los do disco para memória *RAM* rapidamente para que este tempo de carregamento não interfira com a velocidade de transmissão, mas este tempo de carregamento depende fortemente do *Hardware* do Dispositivo e da carga que este está a sofrer no momento de carregamento, fazendo com que a agregação destas Portas tornasse este ponto mais complexo.

4.1.3.1 Scheduler

Nesta versão do Protocolo é o Transmissor a ter a iniciativa de começo de uma transferência e o Controlador está ciente da possível existência de múltiplos *NICs* num dispositivo, faz com que seja necessário criar um procedimento especial para decidir por qual *NIC* se deve realizar uma Transferência. Neste momento, como apenas é necessário escolher um *NIC* por onde se consiga realizar a transferência, o único critério para fazer esta escolha é a existência de uma ligação entre um determinado *NIC* do Transmissor e o *NIC* destino. Desta forma, quando um Nodo pretende começar uma Transferência, para efetuar esta procura de ligações de *NICs*, apenas é necessário enviar o *Datagrama TMI* para o Endereço correspondente ao *NIC* destino pelos seus múltiplos *NICs* que possuem uma conexão à Rede e esperar a recepção de um *Datagrama TMRI*. Embora isto funcione quando apenas existe uma ligação entre o Nodo Transmissor e o Nodo Recetor, quando existe mais que uma ligação o Recetor pode receber mais que um *Datagrama TMI*. Quando isto acontece, o Recetor deverá apenas responder ao primeiro *Datagrama TMI* rejeitando todas as outras ligações, mas este acontecimento poderia ser usado para o início de uma *Transferência MultiPath* descrita anteriormente.

Neste momento não estão a ser usados critérios complexos para a escolha de *NIC*, mas este ponto poderá ser refinado no futuro.

Finalmente este procedimento adicional tem mais uma responsabilidade que é bastante importante, o agendamento do início e retoma de Transferências. Para conseguir tal efeito serão usados 3 estados para a gestão das operações a fazer (Figura 36):

- *Scheduled*;
- *Ongoing*;
- *Finished*.

Uma transferência, quando é inicialmente agendada, fica com o estado inicial *Scheduled* fazendo com que sejam enviados *Datagramas TMI* pelos *NICs* com uma conexão.

Quando for recebido o *Datagrama TMRI* referente ao *TMI* enviado, o estado da transferência pasará a *Ongoing*, dando início à transmissão de *Datagramas CM*.

Daqui pode haver 3 comportamentos

- É recebido um *Over* a indicar o fim bem sucedido da transferência, passando esta para o estado final *Finished*;
- É recebido um *Over* a indicar uma interrupção da transferência, passando esta para o estado *Scheduled*;
- É atingido o *Timeout* final sem receção de qualquer *Datagrama MCID* ou *Over*, passando esta para o estado *Scheduled*.

Os últimos comportamentos que podem alterar o estado de uma transferência são a receção de um *Datagrama Early Over* devido a uma transferência que foi dada como interrompida por *Timeouts* excessivos e colocada no estado *Scheduled* ou devido a uma tentativa do início de uma transferência duplicada desnecessária.

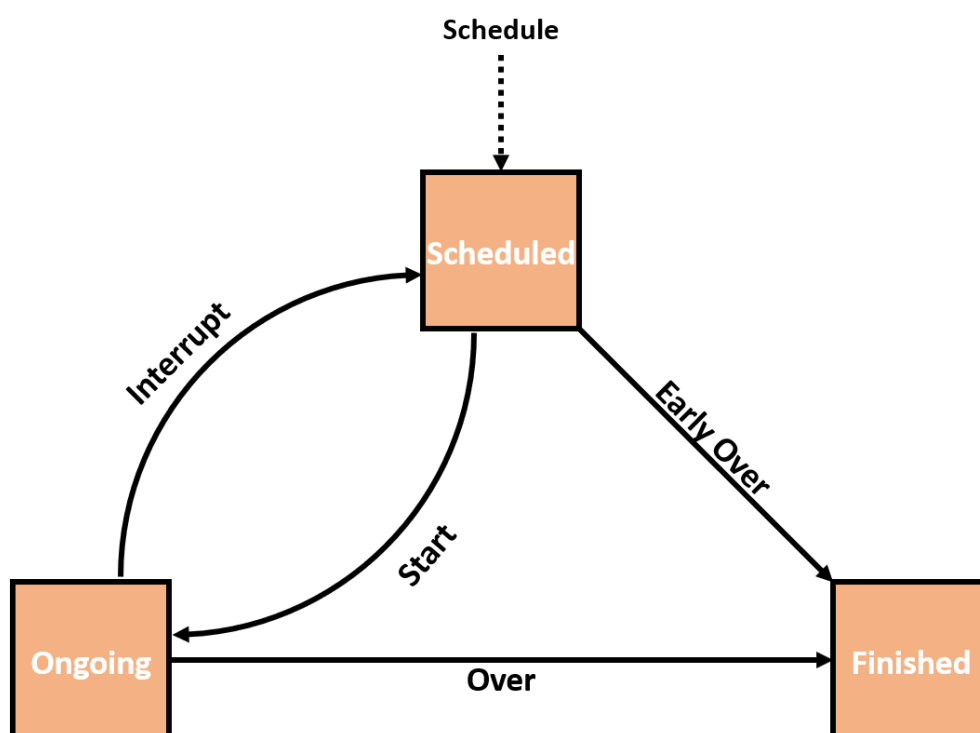


Figura 36: Representação dos estados que as Transferências se podem encontrar e como é que este se altera conforme os acontecimentos do Protocolo

4.1.3.2 Métodos

De forma a este módulo poder ser utilizado, deverá de implementar pelo menos os seguintes métodos para poder haver interação com ela:

- *Schedule*, onde é permitido o agendamento de uma transmissão de determinados Dados para um determinado par *Endereço/Porta Destino*;
- *Cancel*, onde é permitido o cancelamento de uma transmissão agendada ou em execução;

4.1.4 Knock Manager

Finalmente, este terceiro e último módulo do Controlador visa tirar partido dos grupos *Multicast* do *UDP* para a transmissão de dados para outros Nodos, conhecidos ou desconhecidos, que façam parte do mesmo *Grupo Multicast* definido por um par *Endereço:Porta*. Como o próprio nome indica, este módulo vai fazer a gestão completa dos *Datagramas Knock* que serão usados para a comunicação nestes *Grupos Multicast*.

Este módulo será desenhado com a descoberta de Nodos na Rede em mente, sendo que não é entendido o uso deste módulo e *Datagramas Knock* para outros motivos, porém estes últimos terão *payloads* do tipo de dados *Byte[]*, sendo que poderão ser colocados dados neste *payload* mas terão que ser processados pelas aplicações de nível superior.

Como é este módulo que vai tratar da transmissão e receção dos *Datagramas Knock* de forma exclusiva, é necessário especificar o fluxo deste *Datagramas*. Para tal, o Recetor necessita de 1 *Porta Multicast* que esteja associada a um par *Endereço:Porta* de um *Grupo Multicast* e uma *Porta Unicast* para emitir a resposta, e o Transmissor apenas necessita de uma *Porta Unicast* por onde possa emitir o *Datagrama Knock* para o mesmo *Grupo Multicast* e uma *Porta Unicast* para receber as respostas. É essencial que todos os Nodos que pretendam descobrir Nodos na Rede através deste mecanismo também sejam detetáveis pelo mesmo, isto é, se pretenderem emitir um *Datagrama* para um *Grupo Multicast* eles próprios também terão de estar presentes nesse grupo para poderem ser detetados por outros Nodos.

4.1.4.1 Fluxo da Troca de Datagramas Knock

Visto que as Mensagens enviadas para *Grupos Multicast* não são realmente direccionadas a um Nodo em específico mas sim a um grupo de Nodos, que pode não ter qualquer participante, sempre que um Nodo recebe um *Knock Multicast*, este necessita de enviar um *Knock* em resposta ao que recebeu por *Unicast*, pois, se o fizesse por *Multicast*, todos os outros Nodos desse grupo também o iriam receber, desencadeando uma resposta de cada um deles para o mesmo *Grupo Multicast*, levando à eventual saturação da Rede com *Datagramas Knock* desnecessários (Figura 38).

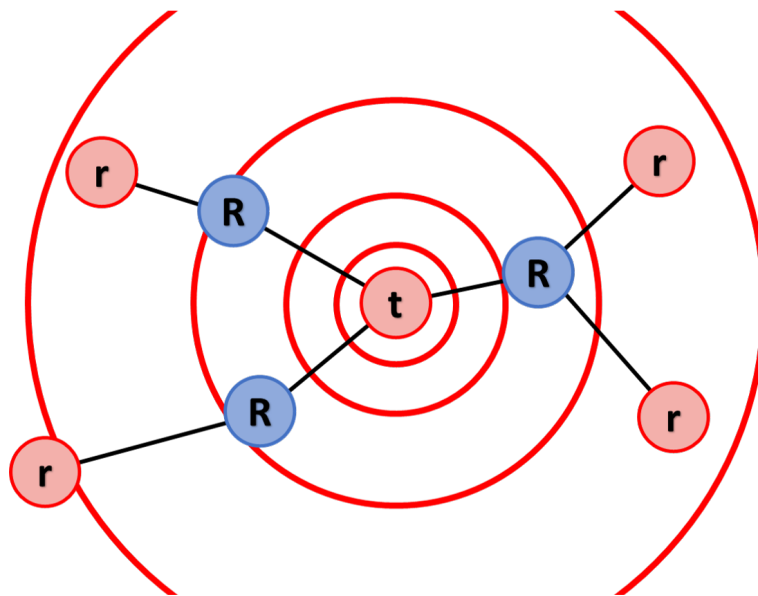
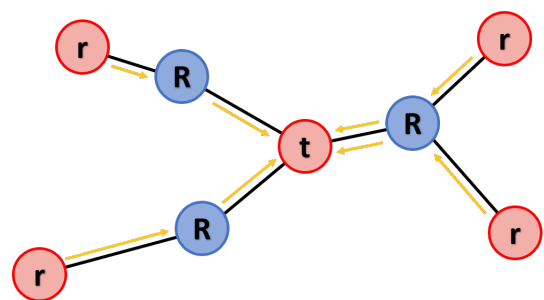
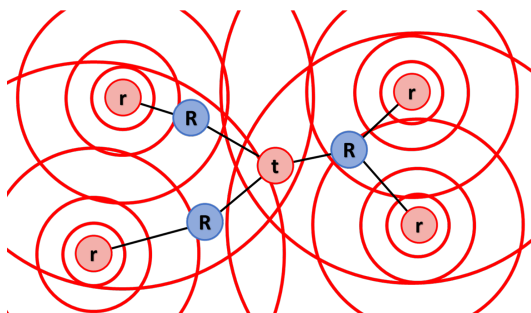


Figura 37: Emissão de Datagrama Knock Multicast para descoberta de Nós na Rede



(a) Respostas ao Datagrama Knock Multicast utilizando Multicast, desencadeando transmissões destes em cadeia **(b)** Respostas ao Datagrama Knock Multicast utilizando Unicast

Figura 38: Diferentes comportamentos perante o uso de diferentes mecanismos de Resposta a uma Mensagem Multicast

Daqui temos o fluxo de *Datagramas* que se pode observar na seguinte Figura.

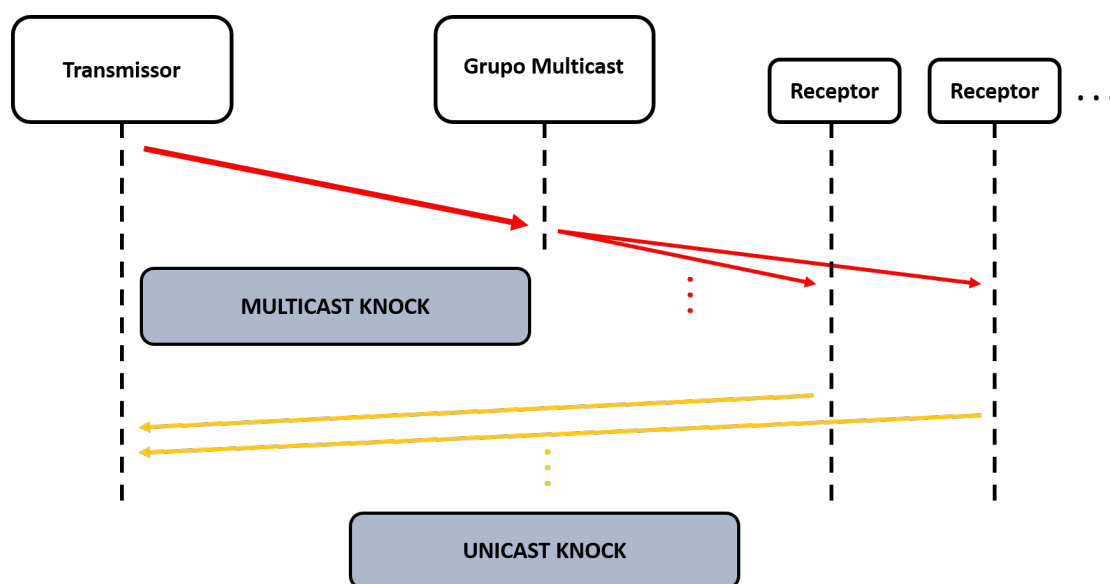


Figura 39: Fluxo de Datagramas Knock Multicast e Unicast

Um pormenor destes *Datagramas Knock* é o *payload* destes. Na implementação deste mecanismo com foco à descoberta de Nodos na Rede pretende-se que um *Knock Multicast* e um *Knock* de resposta em *Unicast* do mesmo Dispositivo contenham o mesmo conteúdo, ou seja, este *payload* tem que ser predefinido.

Uma questão que pode surgir nesta implementação é a utilização de *Grupos Multicast* e não a Utilização de Mensagens *Broadcast*. Esta decisão foi feita porque o *Broadcast* está a entrar em desuso, sendo que já não é possível fazer *Broadcast* quando se utiliza Endereços *IPv6*, apenas *Multicast*. Outro motivo é a organização, com *Grupos Multicast* é possível estabelecer certos grupos para um tipo de Mensagens sem que outros equipamentos não interessados recebam essas mesmas.

4.1.4.2 Métodos

Para a interação com este módulo deverão de ser usados os seguintes Métodos:

- *Start Knock Manager*, que inicia o controlador de *Knocks*, começando a ouvir um determinado *Grupo Multicast*, a responder a *Datagramas Knock Multicast* automaticamente e permitindo o envio dos mesmos *Datagramas*;
- *Stop Knock Manager*, que termina a execução do controlador;
- *Send Knock*, que emite um *Datagrama Knock Multicast* de forma a descobrir outros Nodos na Rede e retorna os *Datagramas Knock Unicast* recebidos como resposta a este;
- *Get Knocks*, que retorna a lista de *Datagramas Knock Multicast* que forma recebidos durante a execução do controlador.

4.2 SISTEMA DE ARMAZENAMENTO

Devido às necessidades específicas deste Protocolo, é necessário desenhar um simples Sistema de Armazenamento que consiga lidar com elas. Para além de terem que ser guardados os *Chunks* da Informação carregada ou recebida através de uma Transferência e devido às necessidades que as Arquiteturas *DTN* apresentam, é necessário que seja guardada Meta Informação relativa aos Dados que um Dispositivo possui bem como Meta Informação relativa a Transmissões que este se encontra a fazer, pois, se o Dispositivo deixar de funcionar corretamente ou necessitar de ser desligado, é necessário que estas Informações consigam ser recuperadas para retomar as Transferências.

4.2.1 Estrutura

Visto que apenas é necessário guardar informação para o Disco do Dispositivo referente aos Dados que este possui e às transferências que este está a realizar, esta Estrutura, visível na Figura 40, estará dividida em 2 Ramos:

- *Data*;
- *Network*.

Sendo a simplicidade o foco principal, no Ramo *Data* estarão todos os Dados, Carregados e Recebidos sem distinção, identificados pelas suas *Hashs Identificadoras*, para não causar qualquer tipo de colisão com os Nomes dos Ficheiros, bem como a Meta Informação de cada um, sendo que no Ramo *Network* estará toda a Meta Informação das Transmissões. Não é necessário guardar informação relativa às Receções de Dados pois neste Protocolo quem toma a iniciativa de início de uma Transferência é o Transmissor, sendo que é este que a tem que recomeçar quando necessário.

Cada Ficheiro, neste Sistema de Armazenamento, é representado por um conjunto de *Chunks*, que contêm os Dados de tal Ficheiro, e um Identificador que os relaciona espacialmente, bem como a Meta Informação relativamente a esses mesmos *Chunks*. Esta Meta Informação é constituída pelos dados:

- *chunkMaxSize*, que indica o maior tamanho, em *Bytes*, que um *Chunk* pode ter;
- *numberOfChunks*, que indica o número de *Chunks* em que os Dados originais foram divididos;
- *numberOfChunksInArray*, que indica o número de *Chunks* que se encontram guardados no Disco do Dispositivo;

- *chunksSize*, que indica o tamanho total dos Dados em *Bytes*;
- *full*, que indica se o Dispositivo possui todos os *Chunks* relativos ao conjunto de Dados;
- *missingChunks*, que mantém registo dos Identificadores dos *Chunks* em falta;
- *Hash*, que se refere à *Hash Identificadora* dos Dados;
- *HashAlgorithm*, que se refere ao Algoritmo usado para obter a *Hash Identificadora*.

Adicionalmente, é possível guardar *Mensagens*, que é um nível superior de abstração sobre esta Meta Informação, e *Documentos*, que são em tudo iguais às *Mensagens* mas possuem um Nome. Por isso, é necessário guardar esta lista de Identificação de *Mensagens* e *Documentos* como sendo Meta Informação Adicional.

Em termos de Meta Informação necessária para se poder recuperar o estado de cada transferência a ser realizada, serão guardados no Ramo *Network* a seguinte Meta Informação:

- *scheduledTransmissions*, que contém a informação necessária sobre Transmissões que estão agendadas;
- *onGoingTransmissions*, que contém a informação necessária sobre Transmissões que estão a decorrer;
- *finishedTransmissions*, que contém a informação sobre as Transmissões que foram concluídas.

Esta separação é alusiva aos 3 diferentes estados em que uma Transferência se pode encontrar num determinado momento, sendo que a última é opcional e, sempre que o programa é reiniciado, todas as Transmissões que se encontram no estado *Ongoing* têm que ser atualizadas para o estado *Scheduled*.

Assim sendo tem-se a árvore de organização do Sistema de Armazenamento representada pela Figura 40.

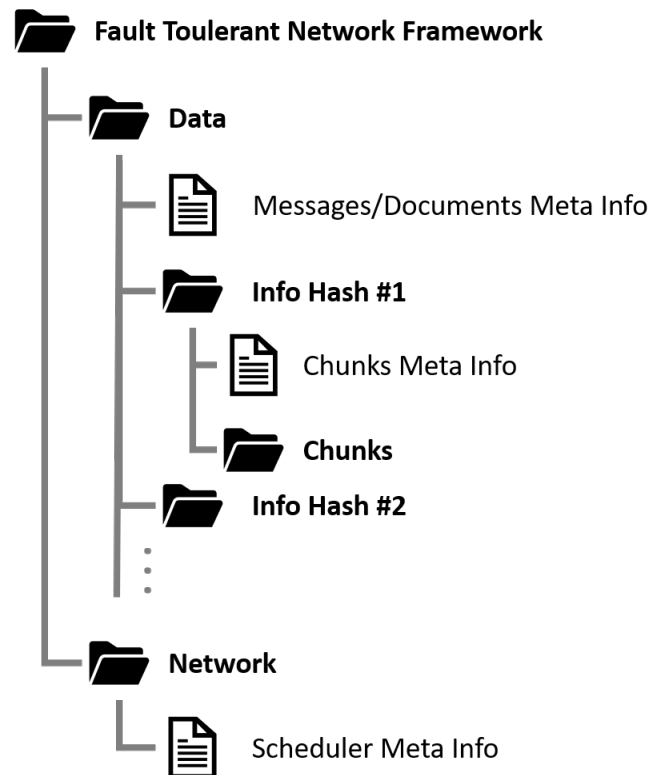


Figura 40: Estrutura dos Ficheiros do Sistema de Armazenamento descrito

4.2.2 Compressão de Dados

Um dos pontos que poderá trazer um bom aumento de performance e possível redução de perda de *Datagramas* na Rede é a compressão dos Dados a serem transmitidos numa transferência. Esta operação sobre os dados será realizada no momento em que o Transmissor carrega os Dados a enviar para Disco, permitindo também a redução de espaço necessário no Disco do Nodo a executar este protocolo. Uma vez transmitidos, o Recetor poderá mantê-los neste estado comprimido em forma de *Chunks* prontos para serem retransmitidos, ou aplicar a operação contrária para reconstruir a informação original. Com esta modificação os Dados originais conseguem ser representados numa forma comprimida que ocupa menos espaço, levando a uma redução no número de *Datagramas* necessários para transmitir os mesmos. É esta redução de *Datagramas* que leva a um aumento da velocidade de transmissão de Dados e a uma redução de possíveis perdas de *Datagramas* na Rede devido ao número reduzido de *Datagramas*.

Mas para esta redução no número de *Datagramas* para os mesmos Dados ser possível é necessário alterar o processo de divisão dos Dados para *Chunks*, colocando a compressão em primeiro lugar (Figura 42) e, após esse processo, a divisão dos mesmos em *Chunks*, pois

se for realizada na ordem contrária vai haver uma redução no tamanho dos *Chunks* e não uma redução do número destes, visível na seguinte Figura.

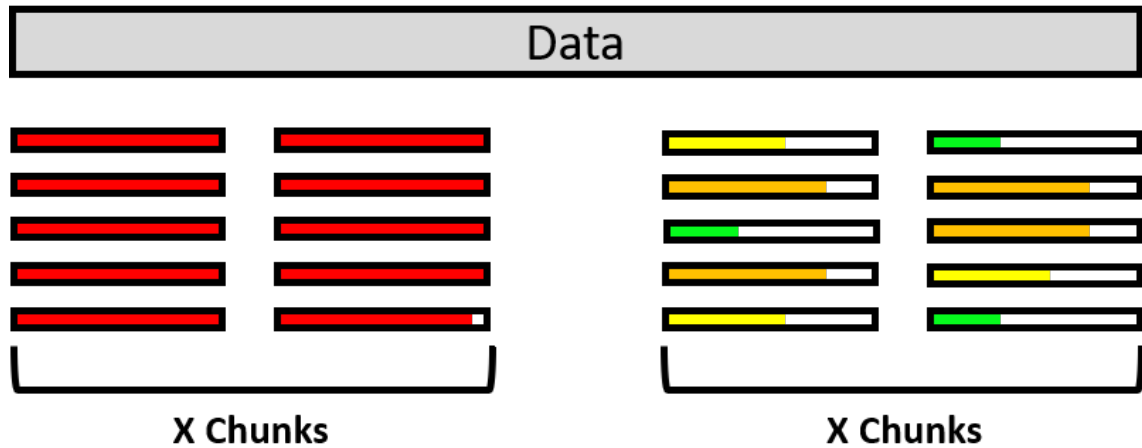


Figura 41: *Processo de compressão após Divisão dos Dados em Chunks originando x Chunks com tamanhos bastantes menores que o tamanho máximo do payload do Datagrama*

Mesmo depois de esta ordem ser estabelecida existe um pequeno detalhe que necessita de ser analisado de forma a manter a ideia inicial deste protocolo, manter os *Datagramas* independentes uns dos outros. Como foi explicado anteriormente, é necessário realizar a compressão dos Dados a enviar antes de os dividir em *Chunks*, mas se esta compressão for feita "de uma vez" na integralidade dos Dados é colocada uma dependência nestes, fazendo com que seja necessário que todos os *Datagramas* *CM* cheguem ao Recetor de uma determinada transferência para que este consiga reconstruir o Ficheiro comprimido e realizar a operação inversa da compressão, obtendo os Dados originais (Figura 43). Assim sendo, é necessário fazer com que este processo de compressão seja gradual para não estabelecer esta dependência. Apesar deste processo ter o resultado pretendido, faz com que o carregamento de Dados para *Chunks* seja dependente de quão boa é a compressão de determinadas secções dos Dados originais, fazendo com que a realização deste processo seja de "tentativa erro", menos eficiente em termos de compressão e mais exigente em termos de recursos.

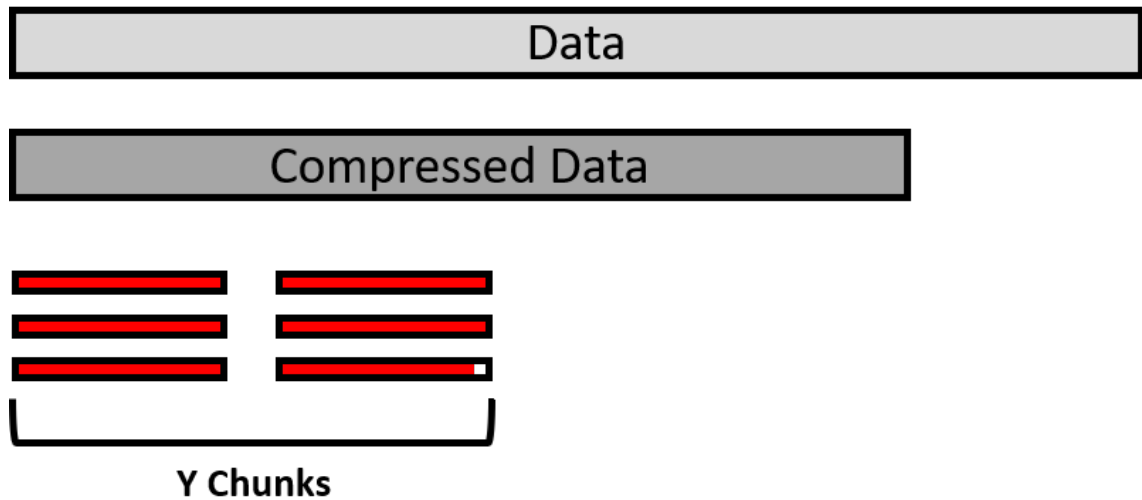


Figura 42: Processo de compressão antes da Divisão dos Dados em Chunks originando y Chunks com tamanhos iguais ao tamanho máximo do payload do Datagrama

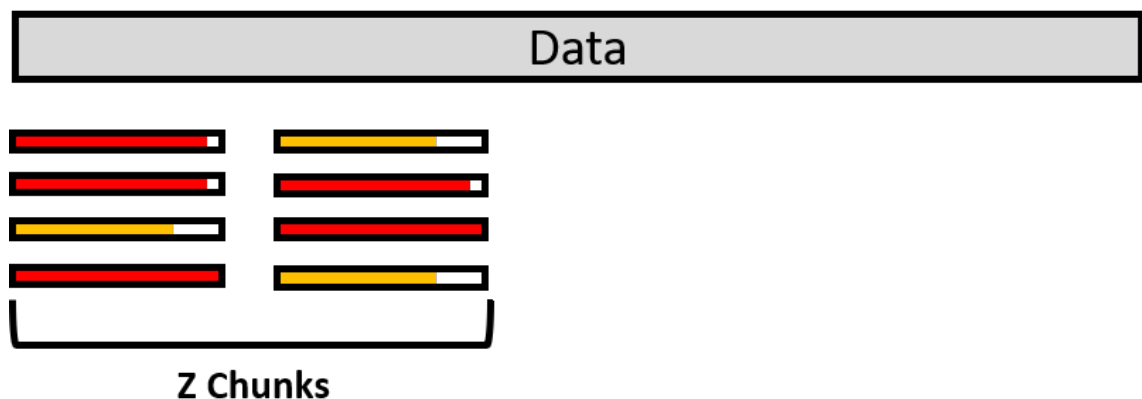


Figura 43: Processo de compressão e Divisão simultânea dos Dados em Chunks originando z Chunks com tamanhos variáveis, mas que maioritariamente ficam perto do tamanho máximo do payload do Datagrama

Tendo em consideração as Figuras 41, 42 e 43, temos que os Dados em consideração geravam x Chunks. Com o processo de compressão e divisão pretendido serão gerados z Chunks independentes mas que não ocupam o *payload* na totalidade. Caso seja tomada a decisão de fazer inicialmente a compressão dos Dados e, apenas após dessa compressão, a divisão dos mesmos em Chunks sacrificando esta característica de independência de *Datagramas*, será possível atingir um maior fator de compressão e, por consequência, uma maior redução do número de *Datagramas* CM (y Chunks). Daqui, assumindo que o processo de compressão não torna os Dados comprimidos maiores que os Dados originais, temos que $0 < y < z < x$, sendo possível fazer um *trade off* em determinadas situações.

Por último é necessário alterar a definição da *Hash* Identificadora dos Dados, pois inicialmente era calculada utilizando os *Chunks* correspondentes a esses mesmos Dados e era referido que esta representava os Dados originais. Visto que foi introduzida a compressão de Dados durante o processo de carregamento dos Dados para *Chunks*, esta *Hash* agora representa a versão comprimida dos Dados Originais.

4.2.3 Métodos

De forma a interagir com o *Sistema de Armazenamento* este deverá de implementar obrigatoriamente os seguinte métodos:

- *Load Message*, que carrega uma determinada mensagem e cria todos os *Chunks* e Meta Informação relativos à mesma;
- *Load Document*, que carrega um determinado Documento e cria todos os *Chunks* e Meta Informação relativos à mesma;
- *Efficient Message Load* e *Efficient Document Load*, que têm as mesmas funcionalidades que os métodos descritos anteriormente, mas têm um comportamento consciente da memória *RAM* disponível no Dispositivo;
- *Get Message Chunks* e *Get Document Chunks* que retornam todos os *Chunks* / um determinado número de *Chunks* / *Chunks* específicos de uma Mensagem e de um Documento já carregado;
- *New Message* que cria um objeto de receção de uma Mensagem e que gere toda a Meta Informação da mesma;
- *New Document* que cria um objeto de receção de um Documento e que gere toda a Meta informação do mesmo;
- *Register Message Chunks* e *Register Document Chunks* que guardam e registam a chegada de *Chunks* recebidos;
- *Get Missing Chunks* que retorna os Identificadores de *Chunks* em falta de uma Mensagem ou Documento;
- *Assemble Document* que reconstrói um Documento que foi recebido e se possui todos os *Chunks*.

Estes são métodos que são necessários para o bom funcionamento deste *Sistema de Armazenamento* mas poderão ser implementados outros métodos que tragam uma maior funcionalidade.

DESENHO DAS ARQUITETURAS DE REDE

Neste Capítulo é desenhada a Arquitetura de Redes *AdHoc* que tira partido de todas as funcionalidades do Protocolo de Transferência de Dados Tolerante a Falhas especificado no Capítulo 3.

5.1 ARQUITETURA DE REDES *ad hoc*

As Redes *Ad Hoc* apresentam um maior desafio para a comunicação entre Nodos, pois estas são Redes que podem apresentar mobilidade, são instáveis, não possuem estrutura fixa e os Dados presentes nelas nem sempre se encontram disponíveis ou acessíveis. Por isso é necessária a implementação de um conjunto de Regras para que seja possível o estabelecimento de uma estrutura de comunicação entre Nodos de Redes *Ad Hoc*. Um pormenor importante é que nesta abordagem não serão consideradas conexões *Unilaterais*, isto é, quando se diz que 2 Nodos conseguem comunicar um com o outro, isto significa que ambos conseguem emitir e receber Dados entre eles, assim são evitados os problemas de ligações *Unilaterais*, qualquer que seja a origem destes.

5.1.1 Estrutura

O primeiro passo para possibilitar a comunicação entre Nodos de Rede *Ad Hoc* é a criação de uma estrutura para que estes consigam obter Dados de uma forma regular.

A primeira grande diferença entre uma Rede *Ad Hoc* e uma Rede "convencional" é a ausência de equipamentos como *Routers* que têm como objetivo a organização da estrutura da Rede, estabelecimento de caminhos de comunicação entre múltiplas Redes e o *Routing* de *Datagramas* entre Dispositivos, logo estas responsabilidades vão ter que ser repartidas pelos mesmos Dispositivos que pretendem obter e transmitir Dados na Rede.

Devido à instabilidade destas Redes é impraticável fazer com que exista uma imagem geral da Rede em cada um dos Nodos, pois, para que seja possível criar e manter qualquer estrutura de Redes, é necessária a existência e troca de Meta Informação da Rede entre os

Nodos desta para que cada um deles tenha noção do estado atual da Rede. Ao contrário de Redes “convencionais”, Redes *Ad Hoc* não podem depender de um Nodo *Master* que possui uma ligação, direta ou indireta, com todos os Nodos da Rede de forma a poder ter uma imagem geral da Rede e tomar decisões de *Routing* por estes Nodos, comunicando tais decisões aos respetivos Nodos. Adicionalmente, se tal estrutura for pretendida, a fluidez e instabilidade anteriormente referidas irão fazer com que haja um aumento do tráfego de *Datagramas* de Meta Informação, o que pode ser prejudicial para o bom funcionamento destas Redes. Finalmente, esta estrutura não é escalável, pois, com o crescimento da Rede *Ad Hoc*, existe um aumento de *Datagramas* de Meta Informação, bem como o aumento da instabilidade da Rede, que por sua vez também leva a um aumento do número deste *Datagramas*.

Assim chega-se à conclusão que é necessária uma estrutura descentralizada, em que cada Nodo consiga tomar decisões individualmente e que permita o *Routing* de *Datagramas* entre Nodos para possibilitar a troca de Dados.

Daqui proponho a criação de uma Estrutura baseada em comunicações *Peer-to-Peer* (*P2P*) entre Nodos em que cada um destes tem uma visualização da Rede centrada nele e caracteriza os outros Nodos à sua volta como sendo seus *Vizinhos*, dividindo estes em múltiplos níveis dependentes da Distância a si próprio. Um *Vizinho* de **primeiro** Nível se encontra a 1 salto de distância, um *Vizinho* de **segundo** Nível se encontra a um mínimo de 2 saltos de distância *ect.* (Figura 44). Apesar disto, quantos mais níveis de *Vizinhos* forem requeridos, maior vai ser a partilha de Meta Informação de gestão da Rede, pois é necessário manter um estado atualizado de um maior número de Nodos e, por isso, nesta definição apenas será feita a atualização ativa de 2 Níveis de *Vizinhos*. Um problema que existe com a falta de equipamentos como *Routers* é a atribuição de Endereços únicos a Dispositivos logo será necessária a utilização de *IPv6 Stateless Address Autoconfiguration* para que cada Nodo seja capaz de se auto atribuir um Endereço baseado no seu Endereço *MAC*.

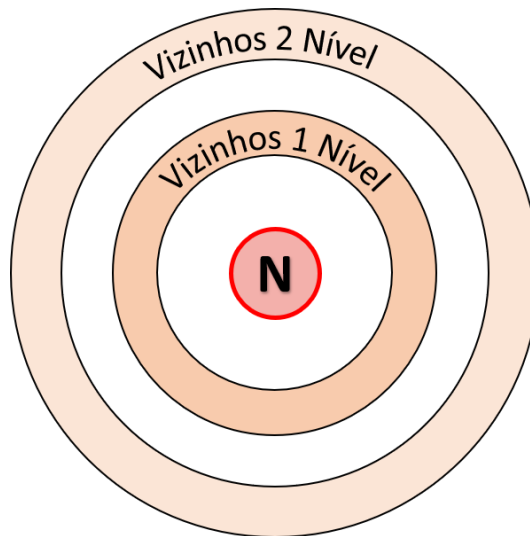


Figura 44: Visualização da Rede centrada num Nó e dividida em níveis de Vizinhos conforme a Distância entre esses Nós e o Nó Central

Desta forma cada Nó possui uma visualização dos Nós imediatamente à sua beira, podendo comunicar diretamente com eles ou utilizá-los como *Gateway* para que possa comunicar com Nós que se encontrem mais distantes.

5.1.2 Adição de Vizinhos

O processo de adição de Vizinhos é um dos mais importantes desta estrutura, pois é esta que abrirá novas ligações entre Nós de forma que estes possam comunicar entre eles.

Considere-se 2 Nós que não se conhecem e cada um deles apenas possui a sua lista atual de *Vizinhos*. Visto que estes não se conhecem, e por isso não têm forma de endereçar *Datagramas* entre si, será necessário utilizar a funcionalidade *Multicast* oferecida pela *Framework* descrita anteriormente. Como a utilização desta funcionalidade permite a emissão de uma pequena quantidade de Dados adicionais, esta pode ser utilizada para informar quais são os Nós que o Nó Emissor conhece, sendo que em resposta virão os Nós que o Nó Recetor conhece. Caso a ligação seja estabelecida, estes Nós que são colocados nestes *Datagramas*, se também não forem conhecidos pelo Nó que os recebe, tratam-se de novos Nós de **segundo e terceiro** Níveis.

Assim tem-se:

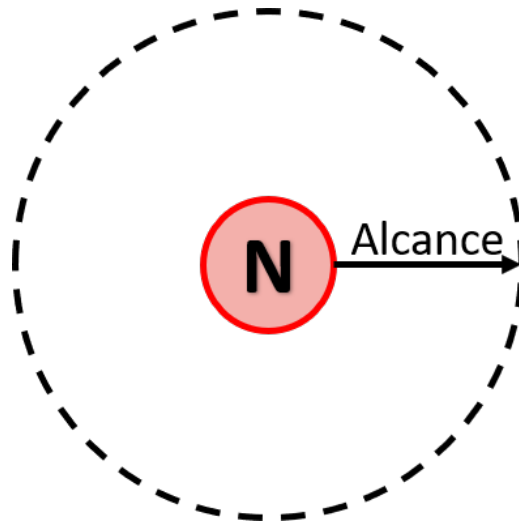


Figura 45: Alcance da Interface do Nó N em que a comunicação com outros Nós é possível

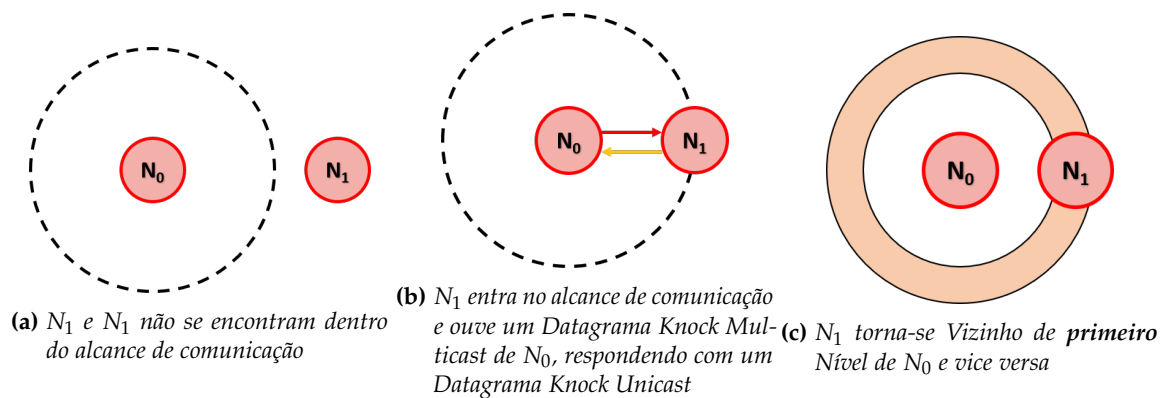


Figura 46: Processo de Adição de um Nó a Vizinho de **primeiro** Nível

Quando um Nó recebe um *Datagrama Multicast* destes, este terá de verificar se este teve Origem num Nó que já conhece, pois se este se encontra dentro do alcance dos seus Vizinhos de **primeiro** Nível, estará constantemente a receber destes *Datagramas* e terá que os ignorar por agora visto que estes poderão ter outra utilidade que se irá tornar pertinente mais a frente. Após esta filtragem, o Nó tem a capacidade de detetar a presença de novos Nós no seu alcance que são potenciais novos Vizinhos. Estes apenas são potenciais novos Vizinhos pois, da mesma forma que o aumento de Níveis de Vizinhos aumenta o envio de Dados de Gestão da Rede, a adição de demasiados Vizinhos de **primeiro** Nível vai ter o mesmo efeito, sendo que deverá ser feita uma gestão do número destes Vizinhos de forma a serem menores que um número predefinido. Consequentemente existirá um menor número de Vizinhos de **segundo** Nível o que também reduz a Carga sobre a Rede e sobre os Dispositivos. Apesar desta limitação, se cada Nó poder possuir 5 Vizinhos de **primeiro** Nível, um Nó poderá ter um conhecimento teórico máximo de 25 Vizinhos

distintos (5 **primeiro** Nível + 20 **Segundo** Nível como se pode observar na Figura 47). Este limite é mais que suficiente, pois para que estes Dispositivos consigam adicionarem-se como *Vizinhos* têm que se encontrar relativamente próximos, o que leva a que existam pelo menos 26 Dispositivos numa pequena área, que consequentemente causarão grandes interferências no meio de transmissão de Dados.

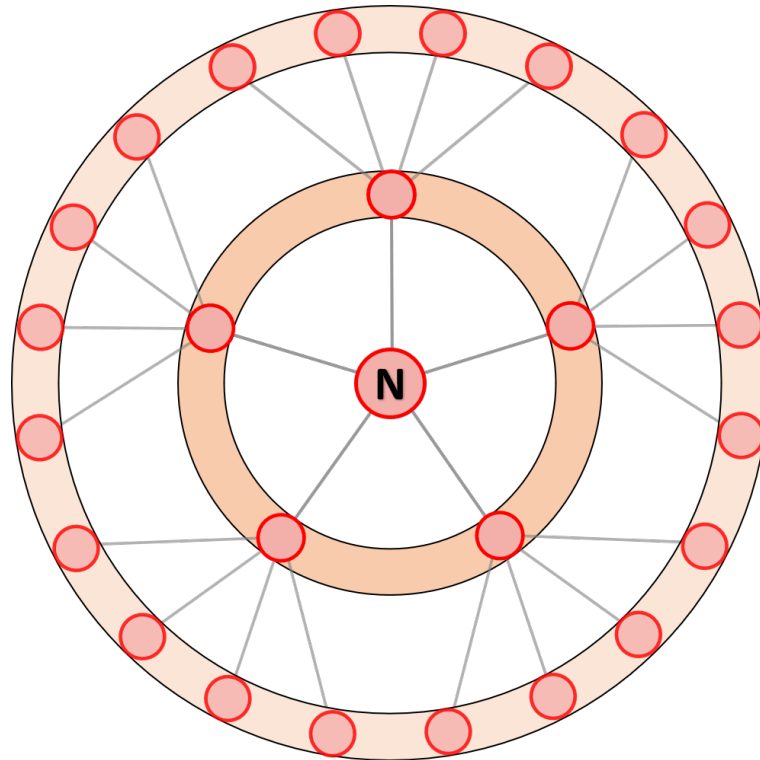
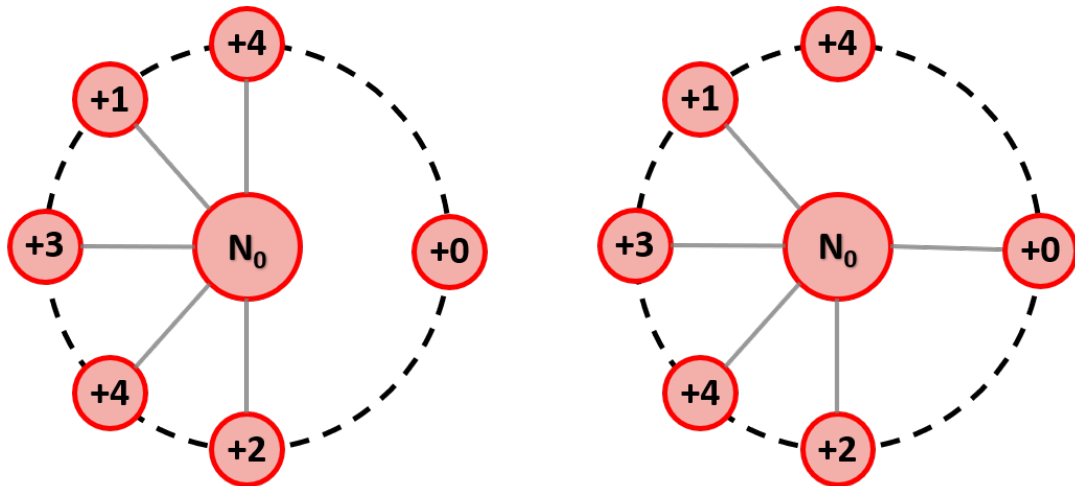


Figura 47: Número Máximo de Nodos distintos que um Nodo N pode conhecer e manter informação atualizada se for utilizado um limite máximo de Vizinhos de **primeiro** Nível

Assim posto, um Nodo apenas deverá de adicionar outro Nodo como *Vizinho* de **primeiro** Nível quando:

- Tem um défice de *Vizinhos* de **primeiro** Nível;
- Pretender reduzir a Distância com o outro Nodo, promovendo-o a *Vizinho* de **primeiro** Nível;
- Após analisar os *Vizinhos* do Nodo que emitiu o Datagrama, verificar que este tem um défice de *Vizinhos* de **primeiro** Nível;
- Após analisar os *Vizinhos* do Nodo que emitiu o Datagrama, verificar que, para além de não conhecer o Nodo, não conhece qualquer dos seus *Vizinhos* de **primeiro** e **segundo** Níveis.

Numa situação em que o Nodo recetor do *Datagrama Knock* repare que o emissor necessita urgentemente de novos *Vizinhos* de **primeiro** Nível e ele próprio estiver com um número elevado destes, este deverá de aceitar a Ligação com o novo *Vizinho*, mesmo que isto o faça ultrapassar o valor máximo predefinido de *Vizinhos* de **primeiro** Nível. Em contrapartida poderá terminar a ligação com o seu *Vizinho* de **primeiro** Nível que possua mais conexões ou que apenas seja despromovido a *Vizinho* de um nível inferior de forma a abrir uma vaga para este Nodo com necessidade de *Vizinhos* (Figura 48).



(a) N_0 possui 5 *Vizinhos* de **primeiro** Nível e deteta a existência de um sexto Nodo que não possui qualquer *Vizinho* (b) N_0 termina uma das suas *Ligações* que possuía com um outro Nodo seguindo um critério pré estabelecido e conecta-se ao Nodo que não possui qualquer *Vizinho*

Figura 48: Adição de um Nodo como *Vizinho* quando um deles não possui qualquer *Vizinho* e o outro não tem qualquer vaga

Numa situação de redução de distância entre Nodos, o Nodo recetor do *Datagrama Multicast* consegue calcular com exatidão se o Nodo emissor se trata de um *Vizinho* de:

- **primeiro** Nível caso este se encontre na sua Lista de *Vizinhos* de **primeiro** Nível;
- **segundo** Nível caso este se encontre na sua Lista de *Vizinhos* de **segundo** Nível;
- **terceiro** Nível caso um *Vizinho* de **primeiro** Nível do Nodo emissor se encontrar na sua Lista de *Vizinhos* de **segundo** Nível ou vice versa;
- **quarto** Nível caso um *Vizinho* de **segundo** Nível do Nodo emissor se encontrar na sua Lista de *Vizinhos* de **segundo** Nível;
- Nível igual ou superior a 5 caso não haja qualquer interceção nos Níveis de *Vizinhos*.

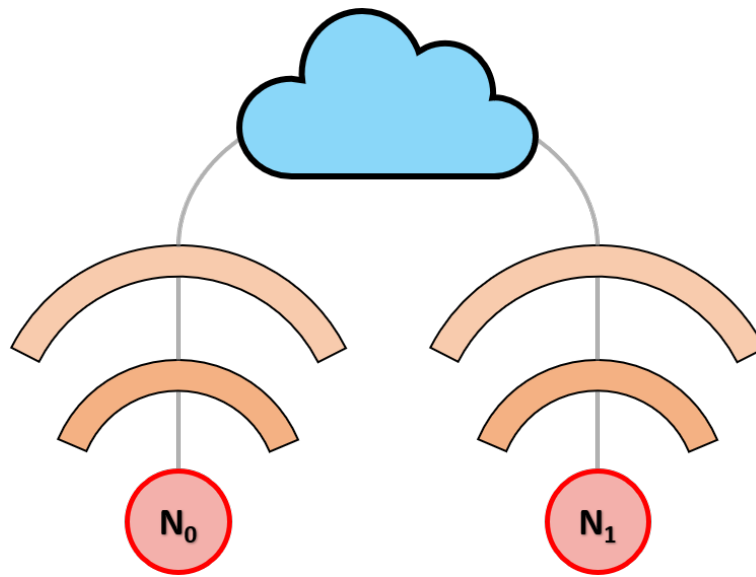


Figura 49: *Calculo da Distância entre 2 Nodos é feito conforme a interceção dos Níveis de Vizinhos. Este exemplo representa uma distância imprecisa.*

Numa situação como a que é apresentada na Figura 50 onde o Nodo recetor verifica que não possui qualquer *Vizinho* em comum com o Nodo emissor, ou seja, caso tenha calculado um Nível de distância impreciso, este deverá de forçar esta adição. Isto acontece pois esta adição, no pior dos casos, estará a encurtar uma distância que anteriormente poderia ser de um nível relativamente grande (igual ou superior a 5 Níveis) para uma de apenas 1 salto, mas no melhor dos casos, era um *Vizinho* com uma distância *infinita*, ou seja, tratava-se de um Nodo que poderia fazer parte de uma Rede completamente diferente, abrindo desta forma a possibilidade de Novas conexões e comunicações entre Nodos que anteriormente não seria possível. Outro benefício desta estrutura de Rede é que esta forma de junção de Redes distintas pode ser feita de forma relativamente silenciosa, não causando grande *Stress* na Rede com um elevado número de *Datagramas* de gestão da Rede.

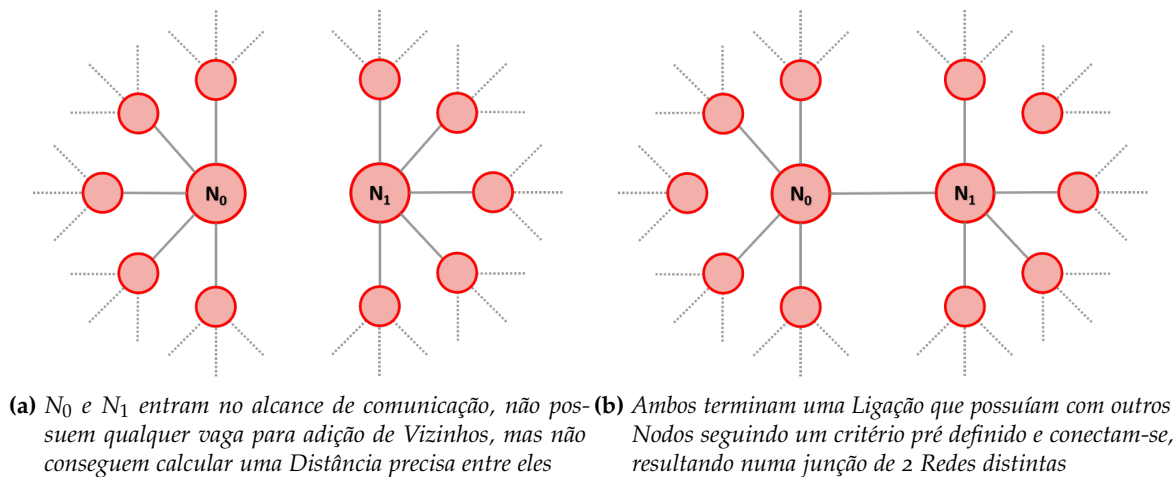
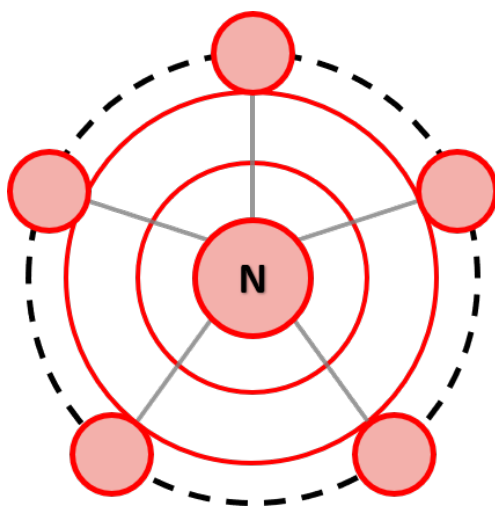


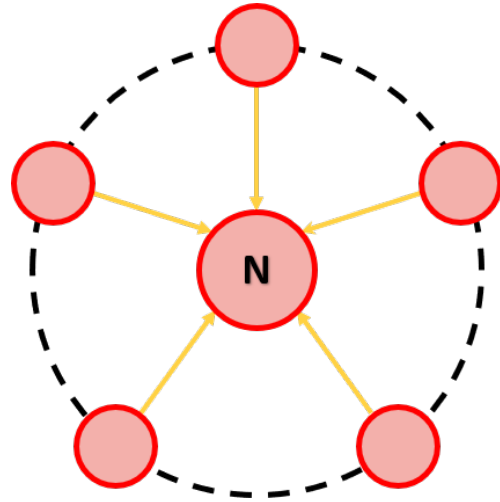
Figura 50: Junção de 2 Redes Distintas

Após a troca de *Datagramas Knock*, os Nodos deverão de Trocar um *Datagrama* adicional de forma a oficializar a ligação entre eles, denominado de *Datagrama NeighbourConfirmation* ou NC. Este *Datagrama* também poderá ser utilizado para o estabelecimento de um protocolo de Segurança para ser utilizado enquanto estes Nodos comunicam entre eles.

Quando uma ligação entre 2 Nodos é estabelecida, é necessário que esta seja mantida e gerida, pois é imperativo que a informação relativa aos *Vizinhos* de diferentes Níveis esteja constantemente atualizada de forma a que os Nodos saibam os seus possíveis *Gateways* a qualquer momento. Para além disso, é necessário que os Nodos consigam saber quando estas ligações terminam, abrindo um novo espaço no número limitado de *Vizinhos* que cada um deles possui. Para isso ser possível, os Nodos que sejam *Vizinhos* de **primeiro** Nível devem de partilhar entre eles a informação relativa aos seus *Vizinhos* para manter a visualização da Rede próxima atualizada e, visto que não é necessário qualquer *Routing* entre estes, devido à sua proximidade, e que isto já é realizado com as mensagens *Multicast* para a Adição de *Vizinhos*, apenas é necessário fazer com que estes Dados sejam pequenos o suficiente para caberem num *Datagrama Knock*, reutilizando estas Mensagens que já teriam que ser emitidas regularmente (Figura 51). Desta forma pode-se afirmar que uma ligação deverá de ser considerada como terminada quando um Nodo não receber um *Datagrama Knock* de um determinado *Vizinho* de **primeiro** Nível durante um período de tempo que englobe a possível emissão de múltiplos destes (Figura 52), ou quando um dos Nodos o faz explicitamente enviando um *Datagrama* que indique o fim da ligação (Figura 53). Esta última forma de acabar com uma ligação é necessária por causa dos casos em que é necessário adicionar um outro Nodo como *Vizinho* mas um dos Nodos não possui qualquer lugar disponível para o fazer, tendo que cancelar uma ligação que já possui.

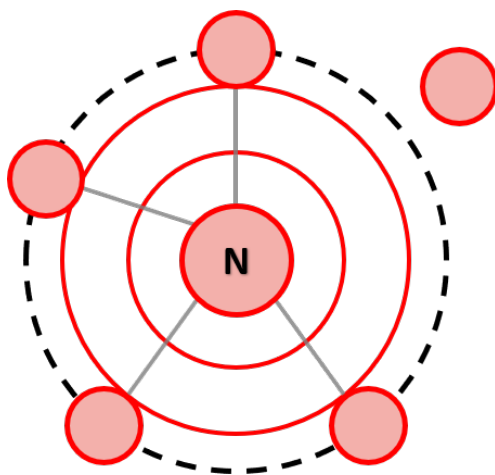


(a) Nodo N emite um Datagrama Knock Multicast

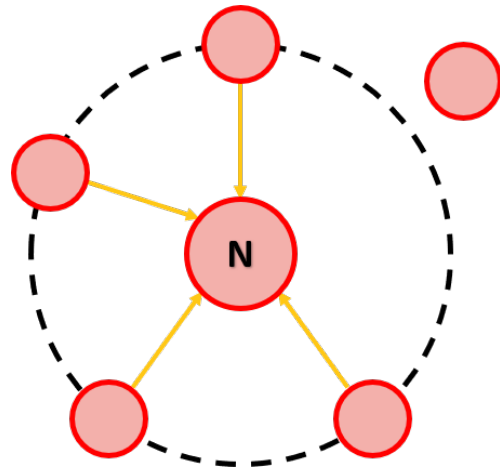


(b) Os Vizinhos de **primeiro** Nível recebem tal Datagrama e respondem com um Datagrama Knock Unicast

Figura 51: Actualização da informação relativa aos Vizinhos utilizando o Controlador de Knocks



(a) Nodo N emite um Datagrama Knock Multicast para atualização da informação relativa aos Vizinhos



(b) Um dos seus Vizinhos já não se encontra no alcance de comunicação, logo não recebe tal Datagrama e consequentemente não responde

Figura 52: Fecho de uma Ligação entre Nodos por estes já não se encontrarem dentro do alcance de comunicação

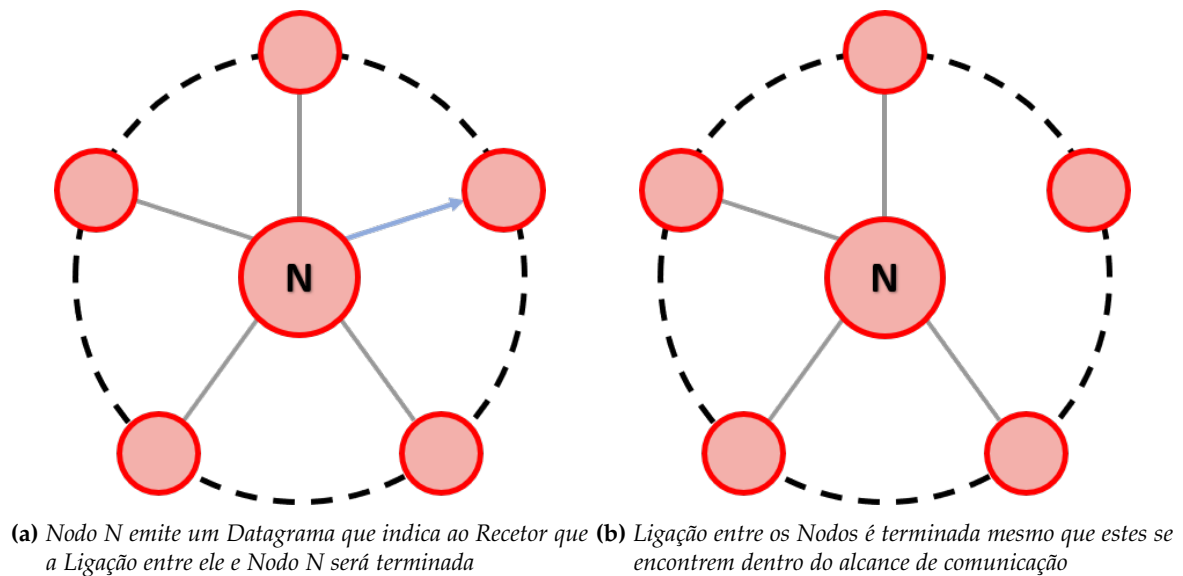


Figura 53: Fecho de uma Ligação que o Nó N possuía com um outro Nó utilizando um Datagrama

Esta estrutura foi construída tirando partido do Protocolo de Transferência de Dados descrito anteriormente, é possível utilizar o mesmo para realizar a troca de Dados entre estes Nós, sendo que esta seria realizada num estilo *P2P*. Embora a utilização deste Protocolo permita a troca de Dados entre Nós numa arquitetura destas, esta seria feita aos bocados, isto é, se a transferência tiver que ser feita entre 2 Nós que se encontrem a mais que 1 salto de distância, vão ter que ser realizadas várias transferências dependentes umas das outras. Uma forma de agilizar este processo seria a utilização de um Protocolo orientado à sessão entre 2 Nós e não à transferência de 1 Ficheiro.

5.1.3 Routing

O *Routing* de Datagramas em Redes *Ad Hoc* é um tema complicado, especialmente se não existir qualquer estrutura e, por não ser o principal objetivo deste projeto, não será abordado neste documento. Apesar disso, a estrutura construída nesta secção permite a troca de Dados direta entre Vizinhos de **primeiro** Nível e um *Routing* de *Datagramas* relativamente simples entre Vizinhos de **segundo** Nível, sendo que quanto maior for esta distância entre Nós, mais complicado este *Routing* de *Datagramas* fica, especialmente se os Nós apresentarem grandes movimentações, desestabilizando a Rede.

5.2 ARQUITETURA DAS REDES DTN

Devido ao longo tempo de desenho e implementação do Protocolo de Transferência de Dados resistente a Falhas na Rede, não foi possível conceber uma estrutura para as Redes

DTN, porém todo o trabalho realizado até este ponto pode ser utilizado como Protocolo da Camada de Transporte de *Bundles* mesmo com as grandes restrições que estes impõe. Os dois grandes desafios que foram encontrados para fazer com que este Protocolo fosse apto à realização de transporte de *Bundles* derivam diretamente de duas características das Redes *DTN*.

A primeira é a obrigatoriedade deste Protocolo ser capaz de ser interrompido devido a falhas no Dispositivo onde este está a ser executado, seja desde um *shutdown* inesperado, a um reinício do Dispositivo já programado previamente. Este Protocolo tem que ser capaz de ser interrompido a qualquer momento em que este se encontra (*Blocos Handshake, Transfer* ou *Over*) e, quando for reiniciado, ser capaz de retomar o estado em que se encontrava anteriormente. Isto foi atingido através da Meta Informação que foi sendo guardada ao longo da execução do Protocolo e ao mecanismo *Scheduler* que procura ativamente uma ligação entre o Nodo Origem e o Nodo Destino.

A segunda característica trata-se da divisão, esperada ou inesperada, de *Bundles*. Esta característica é necessária pois, em Redes *DTN*, o ato de comunicação entre 2 Nodos denomina-se de *contacto* e estes são classificados em 2 categorias:

- Persistentes
- Intermitentes

Neste caso a primeira não apresenta desafios pois os 2 Nodos podem realizar a troca de dados a qualquer altura, mas a segunda pode ser ainda dividida noutras 3 subcategorias:

- Agendadas
- Oportunistas
- Previstas

Neste tipo de *contactos* o tempo de possível comunicação entre os 2 Nodos é limitado e, como a Banda Larga entre estes 2 não é infinita, a quantidade de dados que podem ser trocados entre eles também é limitado. Visto que a troca de Dados nestas arquiteturas é importante por causa destes *contactos* limitados, é necessário transmitir o maior número de *Bundles* possíveis, mas o mais provável é que não seja possível transmitir todos os Dados que representam 1 *Bundle* na sua totalidade, logo é necessário partir estes dados em 2 ou mais *Bundles* distintos. Outra possibilidade é a ocorrência de falhas na transmissão destes *Bundles* em *contactos* de tempo limitado em que não foi possível transmitir 1 *Bundle* na sua totalidade. Nestes casos, o Nodo Recetor tem que ser capaz de utilizar os *Datagramas* que recebeu para realizar uma divisão do *Bundle* original em "sub-*Bundles* mais pequenos de forma a não desperdiçar a informação incompleta que recebeu. Neste Protocolo isto

é possível graças à estrutura de divisão da informação original em *Chunks* e ao princípio de manter a Independência entre estes que foi respeitado ao longo da especificação do Protocolo.

DESENVOLVIMENTO

Neste Capítulo, inicialmente serão explicadas algumas decisões como a escolha de Linguagem de Programação para a Implementação deste Protótipo, bem como a Estrutura geral que o Código apresentará. Posteriormente serão explicadas as diferentes secções de código conforme as diferentes responsabilidades que o Protocolo de Transferência de Dados impõe. Serão analisadas as diferentes classes e apontadas as maiores dificuldades encontradas, bem como estas foram contornadas. Por fim são apresentados todos os problemas encontrados durante a Implementação deste Protótipo que necessitam de maior atenção numa versão futura.

6.1 DECISÕES

Visto que apenas é pretendida a implementação de um Protótipo, uma das primeiras decisões tomadas foi a utilização da Linguagem de Programação *Java*, pois esta apresenta bons níveis de abstração mas também permite com que seja desenvolvido código suficientemente eficiente para o que é pretendido. Para além disso é relativamente estável independentemente do *Hardware* em que o Protótipo será executado, pois o código *Java* é executado utilizando uma *Java Virtual Machine (JVM)* que introduz um nível de abstração relativamente ao *Hardware* e torna o ambiente de execução semelhante independentemente do Dispositivo.

Este nível de abstração que o *Java* permite é bom para o Desenvolvimento de Protótipos pois permite a rápida implementação de uma versão funcional para provar que um conceito funciona como é pretendido, mas o facto de também permitir o Desenvolvimento de código a um nível mais baixo para a implementação das primeiras otimizações também é um fator importante para provar que esses conceitos também têm potencial.

Relativamente à estrutura deste Protótipo, este será dividido em 3 *Packages*:

- *Data*;
- *Network*;
- *Messages*.

Estas divisões foram feitas para separar o código relativo a cada uma das secções descritas na especificação do Protocolo no Capítulo 3 deste Documento:

- *Sistema de Armazenamento*;
- *Protocolo de Transferência*;
- Os múltiplos *Datagramas* descritos.

Outra decisão que teve que ser tomada foi como é que as Mensagens iriam ser *serializadas* para poderem ser colocadas em *Datagramas* e emitidas para outros Nodos através da Rede. Inicialmente foi considerada a utilização de uma Biblioteca chamada *Kryo* que, para além da *serialização* de dados, também permite a compressão desses mesmo, mas rapidamente foi abandonada pois, após a realização de um pequeno teste para a verificar o funcionamento da mesma, apenas eram obtidos erros devido a conflitos com versões do *Java* mais recentes. Com este resultado apenas forma consideradas 2 opções, usar uma estrutura de dados como *Json* ou a própria biblioteca *Serializable* disponibilizada pelo *Java*. Foi escolhida a última para não ser introduzida uma camada de trabalho extra, que por esta altura do Desenvolvimento é completamente desnecessária.

Finalmente, foi tomada a decisão de realizar este Protótipo no *IDE IntelliJ IDEA* utilizando a *Framework Maven* para a possibilidade de construção de um ficheiro *JAR* para a rápida distribuição do executável deste Protótipo para a realização de Testes.

6.2 IMPLEMENTAÇÃO

6.2.1 Messages Package

Este *Package* contém todas as Classes que representam os *Datagramas* descritos ao longo do Capítulo 3 deste Documento. Tais *Datagramas*, e os seus conteúdos, são:

- *TransferMetaInfo*
 - *int* nodeIdIdentifier
 - *int* transferID
 - *int* firstLinkSpeed
 - *int* numberOfCPUCores;
 - *ChunkManagerMetaInfo* cmmi
 - *String* DocumentName
 - *Boolean* Confirmation

- *TransferMultiReceiverInfo*
 - *int* transferID
 - *int[]* ports
 - *int* datagramPacketsPerSecondPerReceiver
 - *CompressedMissingChunksID* cmcID
- *ChunkMessage*
 - *int* transferID
 - *Chunk* chunk
- *MissingChunkIDs*
 - *int* transferID
 - *CompressedMissingChunksID* cmcID
 - *int* DatagramsPerSecondPerSender
- *NetworkStatusUpdate*
 - *int* transferID
 - *InetAddress* newIP
 - *int* firstLinkSpeed
 - *int* newDPS
- *Over*
 - *int* transferID
 - *boolean* isInterrupt
- *Knock*
 - *byte[]* Bytes

Relembrando o *Cabeçalho* do Protocolo [UDP](#), este já possui as Portas *Origem* e *Destino* do *Datagrama*, o tamanho deste e um pequeno *Checksum* para verificação de integridade, logo estas informações não necessitam de ser colocadas no Bloco *Payload*. Isto também é verdade para os Endereços *Origem* e *Destino* dos *Datagramas*, estes podem ser encontrados no Cabeçalho do Protocolo *Ethernet* mas, como é possível observar no *Datagrama* [NSU](#), também é incluído um Endereço para o Recetor deste *Datagrama*. Este Endereço é o Endereço que o Nodo Emissor acha que se trata de ser o seu Endereço público, mas quando existe uma tradução de Endereços por causa de Redes [NAT](#), o Endereço *Origem* do *Payload* e o Endereço *Origem* do Cabeçalho do Protocolo *Ethernet* não vão coincidir e o Nodo Recetor do [NSU](#)

consegue detetar que o Emissor se encontra dentro de uma Rede *NAT* e pode decidir tomar alguma ação caso seja necessária. É importante que esta informação seja incluída neste *Datagrama* pois é este que indica a movimentação do Nodo Origem na Rede ao Nodo que o recebe. De forma a ser permitido o envio de Dados sem garantia de entrega, foi introduzida uma *Flag* denominada *confirmation* no *Datagrama TransferMetaInfo* que, caso contenha o valor **falso**, não será feita qualquer confirmação de chegada de *Datagramas* ao Nodo Destino nem envio do *Datagrama Over*.

Como se pode observar na lista de conteúdos dos *Datagramas Transfer Multi Receiver Info (TMRI)* e *Missing Chunk ID (MCID)*, existe um Objeto denominado de *CompressedMissingChunksID*. Esta estrutura tem como objetivo a compressão de Identificadores de *Chunks* e foi construída conforme a descrição feita anteriormente.

Apesar do *Datagrama Transfer Meta Info (TMI)* possuir o campo *DocumentName*, este poderá ser emitido com o valor *null*, indicando que os Dados a serem enviados nesta transferência se tratam de uma Mensagem.

Finalmente, existe mais um pequeno pormenor nos conteúdos dos *Datagramas NSU*. Estes contêm 2 parâmetros que são referentes a velocidades:

- *int firstLinkSpeed*
- *int newDPS*

Isto acontece pois, neste Protocolo, quem decide a velocidade a que os *Datagramas* deverão de ser emitidos é o Recetor da Transferência, logo, quando existe a necessidade do Emissor enviar um destes *Datagramas*, este irá preencher o parâmetro *firstLinkSpeed* com a sua velocidade de ligação de primeiro salto e esperar um *Datagrama NSU* de resposta do Recetor com o parâmetro *newDPS* preenchido, indicando a nova velocidade a que este deverá de enviar *Datagramas*. No caso do Recetor decidir que a velocidade atual da Transferência continua a ser adequada, este poderá não enviar este *Datagrama NSU* de resposta.

Com isto podemos dizer que existem 2 tipos de *Datagramas* neste *Package*, *Datagramas* relativos às transferências e o *Knock* (Figura 54).

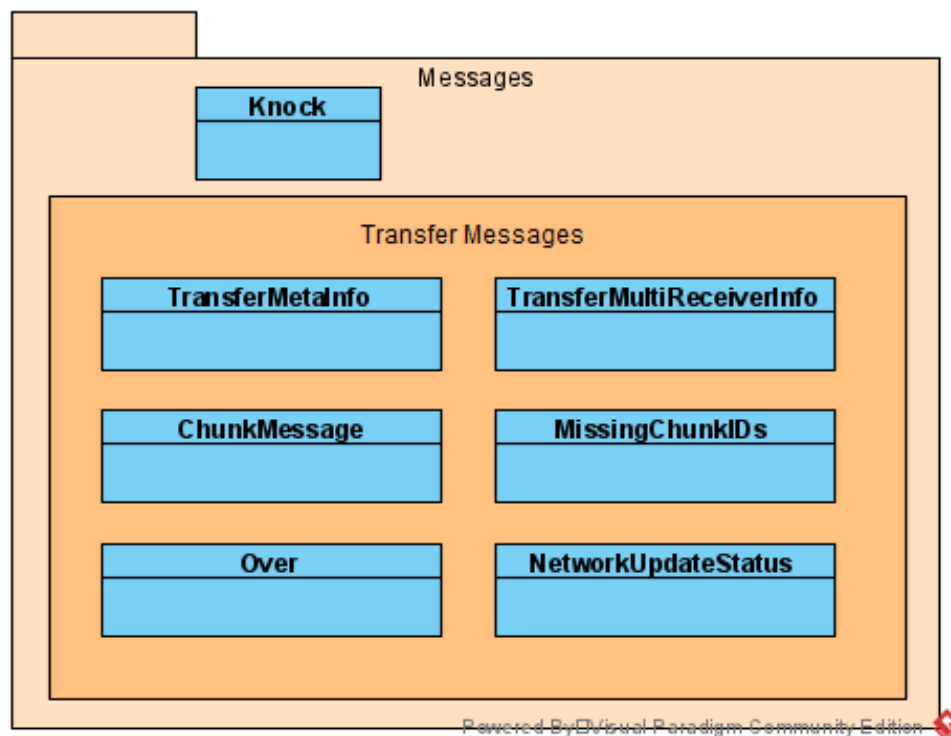


Figura 54: Diagrama de Classes UML simplificado do Messages Package

Todas estas Classes implementam a interface *Serializable* para que possam ser convertidas em *byte[]* para serem transmitidas num *Datagrama* entre Nodos.

6.2.2 Data Package

Como foi referido no Capítulo 3, os Dados que são introduzidos neste Protótipo serão divididos em 2 estruturas:

- *Chunks*
- Meta Informação

Estas estruturas têm como objetivo suportar a divisão de determinados dados e permitir a sua reconstrução posteriormente, sendo possível identificar qualquer erro que possa ter ocorrido recorrendo à Meta Informação destes.

Por isso foi criada a Classe *Chunk* que possui 2 parâmetros:

- *byte[]* Chunk
- *int* Place

O primeiro conterá os dados e o segundo representará o lugar desses dados relativamente ao resto deles de forma a permitir identificar este *Chunk*. Esta classe tem que implementar a interface *Serializable* para que possa ser tornada em *byte[]* para poder ser guardada em disco e transmitida em *Datagramas* entre Nodos.

Para a Meta Informação foi criada a Classe *ChunkManagerMetaInfo*, sendo que esta possui uma multitude de atributos:

- *int datagramMaxSize*, que indica o tamanho máximo em *Bytes* que um *Chunk* pode ter;
- *int numberOfChunks*, que indica o número total de *Chunks* em que os dados foram divididos tendo em conta o *datagramMaxSize*;
- *int numberOfChunksInArray*, que indica quantos *Chunks* o Nodo em que o Protótipo está a executar possui num determinado momento;
- *long chunksSize*, que indica o tamanho total em *Bytes* que os dados ocupam;
- *boolean full*, que indica se o Nodo em que o Protótipo está a executar possui todos os *Chunks* relativos a este conjunto de Dados;
- *boolean[] missingChunks*, que indica que *Chunks* que estão em falta num determinado momento, no Nodo em que o Protótipo esta em execução;
- *String Hash*, que se trata da *Hash* Identificadora dos Dados;
- *String HashAlgorithm*, que se refere ao algoritmo usado para obter a *Hash*.

Como se pode ver na lista anterior, existem alguns parâmetros que são relativos ao Nodo em que o Protótipo está a ser executado e podem ser alterados ao longo do tempo de execução, logo estes não deverão de ser transmitidos como Meta Informação dos Dados no início de uma Transferência. Esta classe tem que implementar a interface *Serializable* para que possa ser tornada em *byte[]* para poder ser guardada em disco e transmitida em *Datagramas* entre Nodos.

De forma a gerir estas 2 Classes e ser possível realizar alterações aos dados que ambas possuem, foi feita uma Classe *DataManager*. Esta é a primeira Classe de grande complexidade, pois é o centro de muitas operações necessárias para o funcionamento do Protótipo, por isso apenas serão descritos os métodos de maior relevância.

Estes são os parâmetros que esta classe possui:

- *String Root*, indica o local do *Sistema de Armazenamento* onde se encontra toda a informação relativa a estes Dados;
- *ChunkManagerMetaInfo mi*;

- `ArrayList<ChunkMessage[]> chunkMessagesToWrite`, que conterá todos os conteúdos dos *ChunkMessages* recebidos durante uma Transferência;
- `long numberOfWrittenChunks`, que indica quantos *ChunkMessages* já foram guardados no Disco;
- `boolean isWrittenToMemory`, que indica se todos os *ChunkMessages* já foram escritos no Disco.

Uma das fontes desta complexidade é resultante das múltiplas possibilidades de criação de um destes Objetos.

A primeira possibilidade de criação é quando se pretende carregar Dados que estão presentes no Nodo. Isto pode acontecer de 2 formas:

- Dados do Disco local;
- Dados da Memória RAM.

No primeiro caso é utilizada a definição:

- `ChunkManager (String root, String localFilePath, String hashAlgorithm, int datagramMaxSize)`

Onde *localFilePath* indica o caminho dos Dados a carregar para estas estruturas de Dados. Adicionalmente pode ser feita uma otimização de utilização de Memória RAM no processo de carregamento destes dados, pois é escusado carregar os Dados na sua totalidade apenas para realizar este processo. Aqui tem-se 2 opções, indicar um inteiro *maxChunksLoadedAtaTime* que representa o número máximo de *Chunks* carregados em memória de 1 só vez, ou ir lendo informação suficiente para preencher o conteúdo de 1 *Chunk* de cada vez e guardar o mesmo para a estrutura de Dados.

A segunda forma é em tudo semelhante à primeira exceto do facto dos dados já estarem carregados em memória, logo será usada uma definição:

- `ChunkManager (String root, byte[] info, int datagramMaxSize, String hashAlgorithm)`

A segunda possibilidade é quando se pretende criar um *DataManager* a partir de um Ficheiro de Meta Informação presente no *Sistema de Armazenamento*. Isto é necessário devido ao caso de uso das Redes DTN em que o Programa é terminado, independentemente da causa, e é imperativo voltar a recuperar toda a informação anteriormente possuída.

- `ChunkManager(String Root, String hash)`

Finalmente, é necessário criar um *DataManager* quando se inicia uma Transferência. Esta forma é diferente das anteriores, pois é necessário iniciar os parâmetros do *ChunkManager-MetaInfo* que são únicos para este Nodo e para o estado atual dos *Chunks* possuídos, sendo que esta informação é vital para o processo de recuperação de erros.

- *ChunkManager* (String root, int datagramMaxSize, String hash, String hashAlgorithm, int numberOfChunks)

Uma outra parte que adiciona muita complexidade a esta Classe é o processo de adição e escrita de *Chunks* para o *Sistema de Armazenamento*, bem como registo da Meta Informação durante a Receção destes numa Transferência. Isto acontece pois o processo de Receção de Dados neste Protocolo é um processo *Multi Thread* com intensa concorrência a Dados que têm que ser controlados cuidadosamente.

Para isso é necessária a utilização de parâmetros extra para este controlo:

- *ReentrantLock* *mi.Lock*, que restringe o acesso ao parâmetro de Meta Informação *mi*;
- *ReentrantLock* *missingChunks.Lock*, que restringe o acesso ao parâmetro *missingChunks* do parâmetro *mi*;
- *ReentrantLock* *memoryWrite.Lock*, que auxilia uma otimização no processo de escrita de *Chunks* para o *Sistema de Armazenamento*;
- *boolean* *isThreadWriting*, que auxilia a mesma otimização referida no ponto anterior.

Esta adição de *Chunks* é um processo complicado devido ao intenso consumo de recursos e por isso é necessário implementar certas otimizações para a redução de responsabilidades de certas *Threads* e a priorização de certos processos. Como já foi referido, a Receção de *Chunks* numa Transferência tem que ser feita o mais rapidamente possível para evitar as perdas de *Datagramas* no Destino por falta de espaço nos *Buffers* de receção de *Datagramas* dos Sistemas Operativos e, por isso, esta tarefa é dividida por múltiplas *Threads*. Com isto é imperativo que estas *Threads* que estejam a receber *Datagramas* não tenham qualquer outra responsabilidade. Assim é necessária a existência de uma *Thread* que recolha os *Datagramas* recebidos num período de tempo e faça este processo de adição e registo de *Datagramas* no *Sistema de Armazenamento*. Mesmo com esta divisão de tarefas, continuam a haver problemas, pois esta *Thread* que recolhe os *Datagramas* tem que realizar 2 tarefas necessárias mas distintas em tempos de execução e prioridade durante uma Transferência:

- Registrar na Meta Informação à chegada de *Datagramas*, que é uma tarefa de extrema importância para o estado atual de uma Transferência a decorrer e relativamente rápida de executar;

- Guardar *Chunks* no Disco, que é um processo necessário mas, para além de não ser importante para o estado de uma Transferência, é extremamente lento comparado ao registo de Meta Informação e varia dependendo do *Hardware* que o Dispositivo possui e da taxa de utilização que este tem no momento de escrita.

Assim é necessário fazer a divisão destas tarefas bem como decidir que *Threads* as irão fazer. O registo de Meta Informação pode ser feita pela mesma *Thread* que recolhe os *Datagramas*, pois esta tarefa é relativamente rápida, no entanto é necessária a criação de uma nova *Thread* para a escrita de Dados para disco.

De forma a reduzir o número de *Threads* necessárias, foi desenvolvido código para que apenas 1 *Thread* esteja a escrever Dados para o *Sistema de Armazenamento*.

Finalmente é necessária uma Classe para encapsular a estrutura de Dados de Identificadores de *Chunks* em falta. Esta é denominada por *CompressedMissingChunksID* e necessita de implementar a interface *Serializable* para poder ser convertida para *byte[]* para ser transmitida num *Datagrama*.

Como a criação desta Estrutura de Dados é feita durante as Transferências e necessita de dados que estão constantemente a ser atualizados, também é necessário realizar o controlo de acesso a Dados neste processo. Um pormenor da criação destas Estruturas é a dificuldade de se saber o tamanho final de cada uma delas antes da sua criação, logo é necessário estabelecer um limite de Identificadores a serem colocados nestas, criá-las, e verificar se os tamanhos destas, em *bytes*, se encontram dentro de um intervalo pré-definido. Conforme o que for observado, será necessário aumentar ou reduzir esta capacidade máxima de Identificadores até que seja atingido um tamanho final aceitável.

6.2.2.1 Document

Uma das decisões que foi tomada foi a divisão entre Documentos e Mensagens, sendo que um Documento apenas é uma Mensagem com um Nome e certos métodos específicos aos Documentos.

Assim os parâmetros de um Documento são:

- *String* Root, que indica o caminho do Documento no *Sistema de Armazenamento*;
- *String* documentName, que representa o Nome do Documento;
- *ChunkManager* cm.

Todos os métodos de criação deste Objeto são análogos aos métodos de criação do Objecto *ChunkManager*.

Um dos métodos específicos a esta Classe é o método de reconstrução da informação para um destino no Disco do Nodo atual. Para tal apenas é necessário indicar o Destino do Documento.

- writeDocumentToFolder(String folder)

6.2.2.2 *DataManager*

Por fim foi criada uma Classe para a gestão de todos os Documentos e Mensagens que um Nodo possui e que está a receber. Esta Classe denominada por *DataManager* tem a responsabilidade de Criação de Documentos e Mensagens, adição de *Chunks* a estes, obter informações sobre os mesmos, apagá-los e todas as outras pequenas responsabilidades de gestão.

Mais uma vez, devido às necessidades das Redes *DTN*, foi criada uma Classe *DataManagerMetaInfo* que serve para armazenar toda a Meta Informação sobre que Documentos e Mensagens um Nodo possui para que, no caso do Programa terminar, seja possível recuperar toda a informação anteriormente possuída.

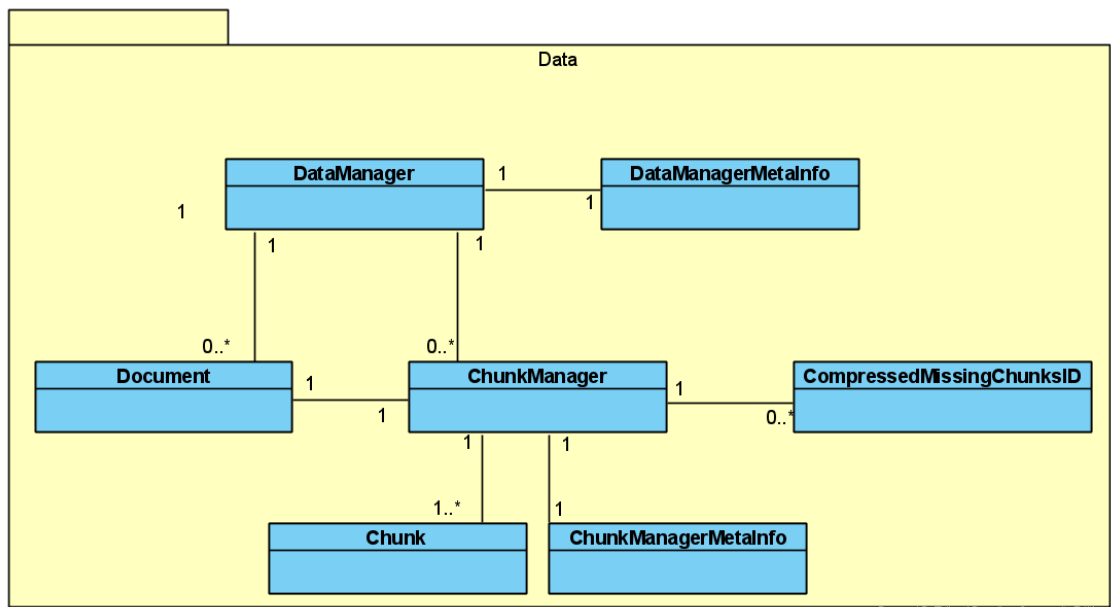


Figura 55: Diagrama de Classes UML simplificado do Data Package

6.2.3 *Network*

Este *Package* tem como objetivo a implementação dos procedimentos do Protocolo de Transferência de Dados em si. Como tal, existem Classes que apenas serão utilizadas para a Receção de Dados e outras que apenas serão usadas para a Emissão de Dados e por isso vai ser feita uma divisão nesta explicação que demonstra esta divisão de tarefas. Para além destas Classes também existem outras que serão revistas no fim deste Capítulo.

6.2.3.1 Listener

Nesta Secção serão explicadas as Classes que foram implementadas para realizar a Recepção de Dados numa Transferência. Começando pela implementação da Classe *FastUnicastListener* que é a Classe responsável pela recepção de *Datagramas CM*, sendo que se trata da única responsabilidade desta, pois esta tarefa tem que ser feita de uma forma extremamente eficiente.

Assim, visto que está a ser usado *Java* e a correspondente biblioteca de *UDP*, é imperativo que após a recepção de um *Datagrama CM*, sejam executadas o menor número de Instruções possíveis de forma a que a *Thread* que executa esta Classe esteja pronta a receber um novo *Datagrama* o mais rapidamente possível. Aqui introduzo um Sistema que denominarei de **Tabuleiro de Datagramas**, isto é, os *Datagramas*, após serem recebidos, necessitam de ser processados para que o Protocolo possa seguir a sua execução normal, mas a *Thread* que os recebe está demasiado ocupada para o fazer no momento da recepção, logo coloca estes num *Buffer* de forma a que possa vir uma segunda *Thread* recolher tais *Datagramas* para que estes sejam processados.

Após ter sido construída uma primeira versão desta Classe, foi observado que esta tinha um mau desempenho devido a alguns pormenores que foram otimizados da seguinte forma:

- Reaproveitamento de Objetos; no decorrer de uma Transferência são emitidos e recebidos múltiplos *Datagramas*, sendo que o método de recepção destes necessita de um Objecto *DatagramPacket* para lá colocar toda a informação relativa a um *Datagrama* recebido. Porém é possível apenas criar 1 destes Objetos, receber um *Datagrama*, eliminar a informação desejada e reutilizar esse Objeto para uma nova recepção.
- Reestruturar código para remoção de *ArrayLists*; inicialmente foram usados *ArrayLists* para construção do **Tabuleiro de Datagramas** pois estes são extremamente úteis para a prova de um conceito mas muito ineficientes. Logo foram substituídos por *Arrays*. Isto levou a que tivesse que ser desenvolvido um **Sistema de Overflow** para o caso em que se quisesse guardar mais *Datagramas* sem que haja espaço disponível no novo **Tabuleiro de Datagramas** que utiliza *Arrays*. Este *Overflow* consiste de um *ArrayList* de *Arrays* que recebe uma nova entrada sempre que o **Tabuleiro de Datagramas** se encontra cheio.
- Decisão do tamanho do *Array* de recepção; com esta reestruturação do **Tabuleiro de Datagramas**, foi necessário basear o tamanho do *Array* na quantidade de *Datagramas* esperados num determinado intervalo de tempo. Para isto foram usadas as medidas de *Round Trip Time* e *Datagramas por Segundo*.

Uma das principais características deste Protocolo é a possibilidade dos Nodos se moverem, e por isso esta Classe tem que conseguir lidar com mudanças no Endereço que está a tentar

ouvir e receber Datagramas. Para isso foi implementado um mecanismo que, quando ativado com um novo Endereço, este irá fechar a antiga *DatagramSocket* utilizada nesta Classe, construir uma nova e fazer *Bind* a este novo par *Endereço:Porta*. Como foi referido na parte da especificação do Protocolo, a alteração de Endereços indica uma nova ligação com um Novo *Router*, fazendo com que possa existir uma nova velocidade de Ligação de primeiro salto, alterando o *DPS* da transferência, logo também foi desenvolvido um método para a alteração do tamanho do *Array* de receção.

Finalmente nesta Classe, foi implementado um mecanismo para que fosse registado um *Timestamp* da chegada de um *Datagrama*, mais especificamente o **Primeiro** *Datagrama* da Transferência e o **Primeiro** *Datagrama* após um *Timeout* para que seja possível a medição de algumas estatísticas.

Mesmo com esta Classe definida é necessário uma que controle os Objetos destas, logo foi construída a Classe *TransferReceiverManager*. Esta tem como objetivo a gestão geral da Receção de *Datagramas* CM, mas em específico é capaz de criar, apagar e gerir Objetos da Classe *FastUnicastListener*, recolher os *Datagramas* CM dos **Tabuleiros de *Datagramas*** destes, registar a Meta Informação destes *Datagramas* no *Sistema de Armazenamento* e ativar a escrita destes mesmos para o Disco do Dispositivo, recolher os Dados estatísticos e calcular estatísticas, detetar *Timeouts* durante a Transferência e, por último, interagir com o Transmissor utilizando os *Datagramas* de gestão *TMRI*, *NSU*, *MCID* e *Over*.

Com tantas responsabilidades, esta foi uma das primeiras Classes a apresentar grandes problemas com a eficiência do código, logo tiveram que ser implementadas algumas otimizações:

- Utilização de lógica booleana; sempre que possível, foram alteradas estruturas de dados por um *Array* de Booleanos, utilizando lógica booleana para que fossem realizadas as mesmas tarefas que eram executadas anteriormente mas de forma muito mais rápida.
- Substituição de *ArrayLists* por *Arrays*; em semelhança à Classe *FastUnicastListener*, foram removidos os *ArrayLists* em favor dos *Arrays*.
- Utilização de Funções *System*; sempre que possível foi alterada a lógica do código para tirar partido das funções da Class *System* disponibilizada pelo *Java*.
- Desenvolvido um Sistema de *Oversleep*; para a implementação de deteção de *Timeouts* foi utilizado um sistema que utiliza um *Thread.sleep(int milli)* que interrompe a execução de código na *Thread* por *milli* milissegundos. O problema aparecia quando a execução de código de receção e registo de *Datagramas* era significativamente grande para ter impacto neste processo de deteção de *Timeouts*, logo é medido o tempo de execução desse código e subtraído ao tempo de *Timeout* calculado.

Ao contrário da Classe anterior, a *TransferReceiverManager* não recebe qualquer *Datagrama* mas emite, logo foi necessário implementar um método para a alteração do Endereço destino dos *Datagramas* que esta emite bem como um sistema para prevenir que estas emissões sejam feitas quando não existe qualquer conexão.

Para o cálculo de Estatísticas foi construída a Classe *ReceiverStats* onde são calculadas as seguintes:

- *Round Trip Time* do *Handshake*, medido entre a emissão do *Datagrama TMRI* e a chegada do **Primeiro** *Datagrama CM*
- *Round Trip Time* de Retransmissão, medido entre a emissão do *Datagrama MCIDs* e a chegada do **Primeiro** *Datagrama CM* após um *Timeout*
- Número de ciclos do Bloco *Transfer*
- Percentagem de Perdas
- Percentagem de Perdas por ciclo do Bloco de *Transfer*
- Velocidade aproximada observada durante o Protocolo
- Velocidade aproximada observada durante os ciclos *Transfer*
Datagramas por segundo
- *Datagrama* por segundo por ciclo do Bloco *Transfer*
- Número de Objetos *FastUnicastListener* tendo em conta velocidades de primeiro salto dos 2 Nodos da Transferência e Cores do *CPU* que cada Nodo diz ter disponível.

Finalmente é necessário uma Classe que possibilite a realização de múltiplas transferências em simultâneo e para tal foi implementada a Classe *ListenerMainUncast*. Para além disto, esta Classe é capaz de criar, gerir e apagar múltiplos Objetos *TransferReceiverManager* e servir de um posto Fixo no espaço de Portas para que o *Transmitter* possa começar uma Transferência. Adicionalmente foi implementada a otimização de agregação de Portas de Gestão das Transferências, logo esta Classe vai ter receber *Datagramas* de múltiplos *Transmitters* e Transferências e ser capaz de reencaminhar cada um dos *Datagramas* que receber para o *TransferReceiverManager* correto. Como esta Classe tem uma *DatagramSocket* ativa, é necessário um método que feche esta e crie uma nova sempre que é detetada uma alteração de Endereço.

6.2.3.2 Transmitter

Nesta Secção serão explicadas as Classes que foram implementadas para realizar a Transmissão de Dados numa Transferência. Começando por descrever a Classe *FastUnicastSender* que tem como objetivo a emissão de *Datagramas* CM.

Tal como a Classe *FastUnicastListener*, a Classe *FastUnicastSender* é uma Classe especializada, neste caso na Emissão de *Datagramas* CM de forma mais eficiente, mas esta apresenta mais alguns desafios.

Para que tal objetivo consiga ser atingido, esta Classe necessita de inicialmente carregar os *Chunks* relativos a uma Transferência de forma a que consiga emitir os mesmos ao ritmo que o Nodo *Transmissor* acordou com o Nodo *Recetor* no *Handshake*. Uma vez que todos os *Datagramas* tenham sido emitidos, o Objeto desta Classe continua a ser válido, pois tem que ser confirmado que todos os *Datagramas* CM chegaram ao Destino, caso contrário alguns terão que ser retransmitidos. Pela existência desta necessidade de retransmissão, foi implementado um Sistema de Retransmissão de *Datagramas* CM que são marcados como em falta pelo Nodo *Recetor*. Também foi desenvolvido um mecanismo de Identificação de *Chunks* já emitidos de forma a possibilitar a Implementação de um Sistema redução de emissões duplicadas desnecessárias de *Datagramas* CM.

Um dos primeiros problemas detetados com a primeira versão desenvolvida desta Classe foi a forma como é controlado o ritmo de envio de *Datagramas*. Inicialmente, a cada envio de um *Datagrama* CM, era utilizado um *Thread.sleep(int milli)* que interrompia a execução de código por *milli* milissegundos, sendo que o menor tempo que este poderia tomar era 1 milissegundo. O problema aparece por 2 motivos, seria impossível ter um Objecto *FastUnicastSender* a emitir mais que 1000 *Datagramas* por segundo, e por causa do método *Thread.sleep(int milli)* ser impreciso, sendo que a *Thread* "adormecerá" **no mínimo** *milli* milissegundos e raramente "adormecerá" esse valor exatamente. Também existe uma versão *Thread.sleep(int milli, long nano)* onde seria teoricamente possível interromper a *Thread* por um tempo menor que 1 milissegundo, mas com estes tempos tão pequenos não é viável utilizar estes métodos visto que estes métodos apresentam uma grande incerteza. Assim foi implementado um sistema de Emissão consecutiva, isto é, de forma a reduzir a utilização de *Thread.sleep(int milli)*, serão agregados um número de *Datagramas* CM para serem emitidos consecutivamente, fazendo com que o tempo acumulado de interrupção da *Thread* seja maior, diminuindo o impacto das imprecisões da mesma. Infelizmente isto torna a medição de *Jitter* muito complicada e fez com que esta estatística não fosse implementada nesta versão do Protótipo.

Uma otimização óbvia nesta primeira implementação é o carregamento de *Chunks* para memória, pois, se estes forem todos carregados de uma só vez, vai haver uma grande exigência de memória RAM. Assim é possível fazer carregamentos parciais em que apenas uma porção dos *Chunks* estão carregados em memória RAM e apenas se carrega mais quando

os primeiros já tiverem sido todos emitidos. Contudo, esta otimização faz com seja necessário carregar dados durante o processo de Transmissão, o que pode ter um grande impacto neste processo. Este carregamento depende totalmente do *Hardware* do dispositivo e é possível que, se o tempo de carregamento for exageradamente alto, o Nodo Recetor detete um *Timeout* erradamente. Assim devem de ser carregados o número de *Chunks* que serão emitidos consecutivamente de forma a reduzir a utilização de memória *RAM* e reduzir a possibilidade de ocorrência deste *Timeout* indevido. Adicionalmente foi implementado um Sistema de *Oversleep* semelhante ao que foi implementado na Classe *TransferReceiverManager* de forma a atenuar os tempos de carregamento de *Chunks*, subtraindo este tempo de carregamento de *Datagramas CM* ao tempo de interrupção de execução acumulado.

Também foi tomado em consideração a reutilização de Objetos nesta Classe, sendo que apenas foi criado 1 Objeto *DatagramPacket* e foi alterado o seu conteúdo e parâmetro do tamanho.

Tal como algumas Classes anteriores, esta Classe tem que lidar com a movimentação de Nodos na Rede e consequente alteração de Endereços, logo foi implementado um método para permitir a alteração do Endereço Destino dos *Datagramas CM*. Ainda neste assunto de movimentação de Nodos, quando o Nodo Transmissor perde conexão ou se apresenta no processo de reconexão, este não deverá de emitir qualquer *Datagrama CM*, pois estes serão automaticamente perdidos na Origem, sendo que as *Threads* que executam o código dos Objetos *FastUnicastSender* terminam a sua execução inesperadamente. Mesmo neste caso é necessário ter em atenção a um pormenor, caso os parâmetros onde são armazenados os *Chunks* que são carregados temporariamente, bem como os identificadores dos *Chunks* que faltam ser carregados e enviados, não forem parâmetros globais da Classe, estes serão permanentemente perdidos quando a *Thread* para a execução levando a uma grande perda na Origem, logo estes necessitam obrigatoriamente de ser parâmetros globais. Adicionalmente, se for pretendido uma maior poupança de memória RAM ao custo de se ter que carregar mais *Chunks*, quando a *Thread* entra neste estado de interrupção de execução por falta de conectividade, pode-se descartar todos os *Chunks* pré carregados desde que os Identificadores destes sejam adicionados ao parâmetro que indica que *Chunks* necessitam de ser carregados e enviados.

Finalmente, devido à possibilidade de alteração de condições na Rede e movimentações de Nodos, foi implementado um método para a alteração do parâmetro *DPS* e a consequente alteração de ritmo de envio de *Datagramas CM*.

Tal como na Capítulo anterior, é necessária a existência de uma Classe que seja capaz de criar, gerir e apagar esta última Classe especializada. Assim foi implementada a Classe *TransferMultiSender* que adicionalmente tem como principais objetivos o processamento dos *Datagramas* de Gestão de Transferências *TMRI*, *NSU*, *MCIDs* e *Over* relativos a uma Transferência, divisão de *Datagramas CM* a transmitir, ou retransmitir, pelos múltiplos Objetos

da Classe *FastUnicastSender* que controla e o Controlo do Sistema de *Timeouts* do Transmissor para a deteção de momentos de desconexão entre os Nodos da Transferência e a consequente libertação de recursos que estão a ser consumidos para a realização desta. É também nesta Classe que se realiza a deteção de possíveis emissões duplicadas desnecessárias de *Datagramas CM*, por isso tal Sistema foi implementado utilizando o mecanismo de detecção de *Datagramas CM* emitidos descrito na Classe *FastUnicastSender*.

Nesta Classe é necessário ter em atenção que, no caso de um Nodo Recetor já possuir alguns *Chunks* antes do início desta Transferência, no Bloco *Handshake* poderão ser emitidos *Datagramas MCIDs* no mesmo instante em que é emitido o *Datagrama TMRI*. Visto que neste Protocolo não existe garantia de entrega de *Datagramas* nem garantia de entrega destes na ordem em que são emitidos, poderá ser possível receber *Datagramas MCIDs* sem que seja recebido o *Datagrama TMRI* ou que estes cheguem fora de ordem, fazendo com que possa ser necessário guardar o conteúdo dos *Datagramas MCIDs* até à chegada do *TMRI*.

Em semelhança às otimizações que foram feitas na Classe *TransferReceiverManager*, também foi utilizada lógica *booleana* onde possível para ganho de eficiência.

Com a maior parte das funcionalidades principais do Protocolo de Transferência de Dados já implementadas, apenas resta implementar uma Classe que tome a iniciativa de iniciar as Transferências e que seja capaz de receber todos os *Datagramas* de Gestão destas provenientes dos Recetores. A Classe implementada que tem estes objetivos denomina-se de *Scheduler*. Esta Classe permite que sejam agendadas Transferências para serem realizadas quando a oportunidade se apresenta. Visto que hoje em dia é comum a existência de múltiplas *Network Interface Card (NIC)* num único Dispositivo, esta Classe tem que ser consciente que podem existir mais que uma conexão entre 2 Nodos e por isso, quando uma Transferência é agendada, esta tem que ser capaz de emitir o *Datagrama TMI* pelos múltiplos *NICs* que o Dispositivo possui. Para tal tarefa conseguir ser realizada para um número arbitrário de *NICs* de forma relativamente simples, foi implementada a Classe *SchedulerIPListener* que tem como objetivo a criação de objetos que representem os múltiplos *Gateways* que o *Scheduler* tem. Para além disto, também servem de recetores de todos os *Datagramas* de Gestão de Transferências, sendo que estes são reencaminhados para o *Scheduler* e este tem que ser capaz de os reencaminhar para o objeto da Classe *TransferMultiSender* correspondente.

Visto que é esta Classe que tem acesso a todos os *Datagramas* de Gestão de Transferências, é também nesta que vai ser feito o controlo e a gestão dos estados atuais de cada uma das Transferências, de forma a ser possível recuperar de casos de falha completa da Rede. Este mecanismo é vital para o recomeço de transferências interrompidas e, devido às necessidades das Redes *DTN*, a Informação relevante para este mecanismo será guardada numa Classe *SchedulerMetaInfo* que implementará a biblioteca *Serializable* para poder ser guardada no Disco do Dispositivo e recuperada quando for necessária.

utilizando o método `getNetworkInterfaces()`, porém este método apenas retorna os *NICs* que têm uma conexão ativa no momento em que o método é executado. Isto é uma barreira pois é impossível saber ao certo quantos *NICs* um Dispositivo possui no total no início da execução do Protótipo, sendo necessário desenvolver código para lidar com a possibilidade de *Hot Swappable NICs*. Inicialmente foi essa a abordagem tomada, mas rapidamente foi abandonada devido ao grande nível de complexidade que seria introduzido no Protótipo a partir deste ponto. Então, de forma a simplificar, foi tomada a decisão de ligar esta detecção de *NICs* ao Sistema Operativo *Linux* recorrendo aos Ficheiros de *Networking* deste e obtendo a informação desses *NICs* através do método `getByName(String name)` do mesmo objeto referido anteriormente, sendo que, se um determinado *NIC* não conseguisse ser detetado pelo *Java*, seria retornado um *null* indicando que este não possui uma conexão. Mesmo conseguindo obter esta informação vital ao bom funcionamento do Protótipo, a decisão de continuar a usar o *Java* para este processo significou que apenas seria possível obter a informação de um *NIC* que possuísse uma conexão explícita com um *Router* ou outro Dispositivo, descartando a possibilidade deste Protótipo funcionar numa Arquitetura com uma estrutura fluida como as Redes *Ad Hoc*.

Com estas decisões feitas foi necessária a implementação de uma Classe *NIC* que possibilitaria a criação de Objetos que obtivessem a informação necessária relativamente aos *NICs* e a transmitissem a todas as outras Classes desenvolvidas. Visto que foi criada esta Classe, também foi implementado nesta um Sistema que regista quantas Transferências estão a decorrer através de um determinado *NIC* e divide igualmente a banda larga disponível entre elas.

Ainda nesta Classe foram encontradas outras barreiras relacionadas com a dificuldade de obtenção de dados relativos aos *NICs*. Uma delas foi obter a velocidade de transmissão de dados no primeiro salto que o dispositivo tinha acordado com o *Router* ou Dispositivo a que se encontrava conectado. Mais uma vez foi necessário recorrer aos Ficheiros de *Networking* do *Linux* para obter esta informação. Apesar disto, foi observado que os Ficheiros relativos a estas velocidades dos *NICs* wireless não possuem qualquer informação e foi necessário recorrer à execução do programa *iwconfig* e realizar *scraping* do *output* deste.

Também foram encontrados problemas na emissão de *Datagramas* de Gestão utilizando múltiplos *NICs*. Isto acontece porque na implementação deste Protótipo foram utilizados *binds* de *DatagramSockets* a pares *Endereço:Porta* com a intuição do *Java* utilizar o *NIC* correspondente para a emissão de tais *Datagramas*. No entanto foi observado que o *Java* emite todos os *Datagramas* por apenas 1 *DatagramSocket* com diferentes Endereços Origem. Isto fez com que não fosse possível utilizar múltiplos *NICs* na execução deste Protótipo.

Por último, foi encontrado um obstáculo com a utilização de Endereços *IPv6 LinkLocal* durante o processo de *bind* de uma *DatagramSocket* a um destes Endereços. Da forma como foi implementada a atualização de Endereços nas múltiplas Classes que necessitam dessa

implementação devido à movimentação de Nodos na Rede, existe um problema com o segundo *bind* realizado em alguns casos(*rebind*). Quando um Nodo possui uma conexão e se encontra a realizar uma Transferência, este possui múltiplas *DatagramSockets* a utilizar vários pares *Endereço:Porta*, mas quando este perde a conexão é necessário libertar recursos e fechar estas *DatagramSockets* para que estes não estejam em constante uso desnecessário. Quando o Dispositivo é capaz de se reconectar ao *Router* antes da Transferência ser terminada por falta de comunicação, voltam a ser alocados Recursos e são feitos novos *binds* para as *DatagramSockets* (*rebinds*). O problema aparece nestes *rebinds* quando são usados Endereços IPv6 *LinkLocal* sendo lançada a exceção *java.net.BindException: Cannot assign requested address (Bind failed)* sem razão aparente. Uma hipótese para este erro devesse à natureza destes Endereços, pois estes fazem parte de uma Rede *interna* ao Dispositivo controlada pelo Sistema Operativo. Quando o método *close()* do objecto *DatagramSocket* é utilizado em *DatagramSockets* que possuem um *bind* a um destes Endereços, suspeita-se que seja lançado um pedido assíncrono ao Sistema Operativo para a realização deste processo, mas que a Exceção *SocketException* seja lançada instantaneamente, abrindo a possibilidade de um *rebind* a um par *Endereço:Porta* que ainda esteja em uso após a utilização do método *close()*. Com esta suspeita foi implementado um Sistema de espera que, quando este erro fosse detetado, seria feito um intervalo de espera antes de se tentar realizar o *rebind* novamente e, após alguns testes, foi verificado que o problema já não era encontrado. Apesar destes resultados, não se recomenda a utilização de tal Sistema pois pode levar a outros problemas bem piores e será necessário encontrar outra solução.

Finalmente temos os problemas que derivam do desenho deste Protocolo e da Implementação. Um problema óbvio do Protocolo é como este determina a velocidade a que uma Transferência deve de ser feita. Se a distância entre os Nodos que se encontram a fazer uma Transferência for de apenas 1 Salto, este Protocolo não encontrará qualquer problema, mas se esta distância for superior, a largura de banda entre esses Nodos pode não ser igual à menor velocidade de primeiro salto desses Nodos, sendo que no mundo Real isto não será verdade na grande maioria dos casos devido às limitações de velocidades que os Provedores de Serviços impõem. Porém a abordagem tomada não é tão descabida como parece, pois a velocidade calculada apenas se trata de uma velocidade máxima possível sendo que, se a largura de banda real for menor que a que se pensa, a transferência sofrerá com *Drops na Rede*, acontecimento que este Protocolo foi construído para conseguir lidar. Um dos maiores problemas de Implementação é a forma como a informação dos *NICs* é tratada. Num mundo sem Redes *NAT* isto não seria um problema, mas com estas é possível que um Dispositivo esteja conectado a um *Router* com o Endereço *10.0.0.100*, desconecte-se deste *Router* e conecte-se a outro sem qualquer relação ao primeiro, e volte a obter o Endereço *10.0.0.100*, fazendo com que este não detete esta mudança de Endereço e não desencadeie a transmissão de Datagramas *NSU*.

6.2.6 *Main*

Visto que não foi possível realizar a Implementação de pequenas aplicações para a demonstração desta *Framework* devido à falta de tempo, foram implementados métodos na Classe *Main* para demonstrar as funcionalidades básicas que esta dispõe.

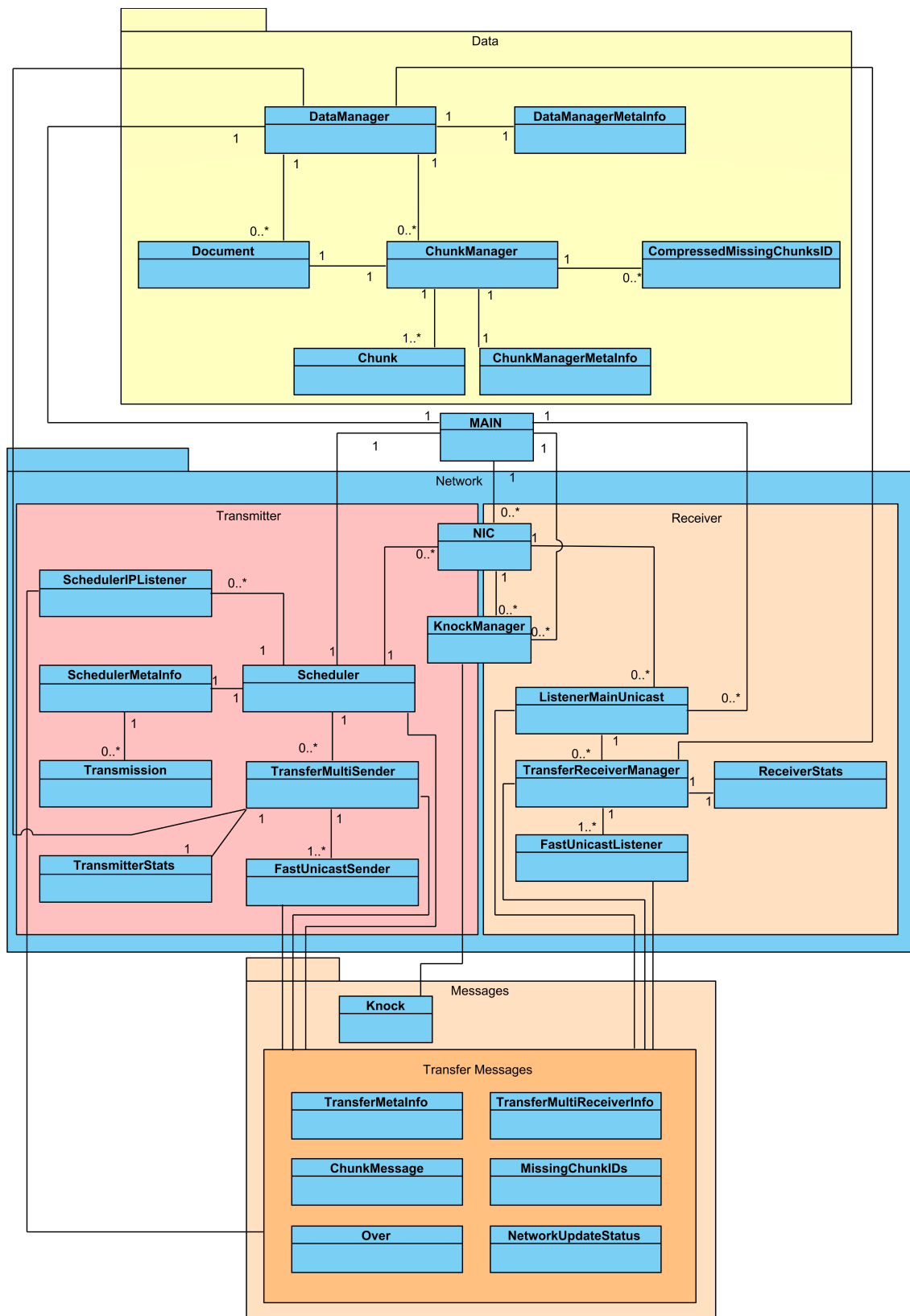


Figura 57: Diagrama de Classes UML simplificado do Protótipo Desenvolvido

TESTES E RESULTADOS

Neste Capítulo serão apresentados os resultados dos múltiplos testes realizados ao Protótipo desenvolvido utilizando diferentes disposições de Redes.

7.1 RESULTADOS

Os seguintes testes foram escolhidos de forma a demonstrar grande parte das funcionalidades deste Protocolo, sendo que funcionalidades como a divisão da banda larga por múltiplas transferências, redução de velocidade devido a Perdas, *Early Over* e outras não são demonstradas pois são consideradas triviais e não são o Objetivo Principal deste Projeto, mas sim consequências do desenho e Implementação realizados.

Assim, de forma que os únicos ponto de falha possíveis na realização deste testes ocorressem na Implementação em si, foram escolhidos 2 Dispositivos com bastantes recursos:

- PC_1 :
 - CPU com 8 Cores, 16 Threads a 3.3GHz/4.4GHz;
 - 16GB memória RAM;
 - SSD NVME;
 - 1 Ethernet NIC;
 - 1 Wireless NIC.
- PC_2 :
 - CPU com 4 Cores, 8 Threads a 4.0GHz/4.2GHz;
 - 16GB memória RAM;
 - SSD SATA;
 - 1 Ethernet NIC;

Em relação aos Routers usados, embora fossem de fabricantes diferentes, as estruturas usadas para estes testes fizeram com que cada Dispositivo tivesse um *bottleneck* de 100 Mbps na comunicação entre eles.

Em relação aos testes, estes serão divididos conforme a estrutura da Rede em que foram realizados e em cada uma destas secções serão explicados os objetivos de cada um dos testes, apresentados e analisados os resultados obtidos e discussão destes.

7.1.1 *Conexão Directa entre Dispositivos*

Nesta secção serão realizados e analisados os testes com uma ligação direta entre os Dispositivos semelhante à da Figura 58.

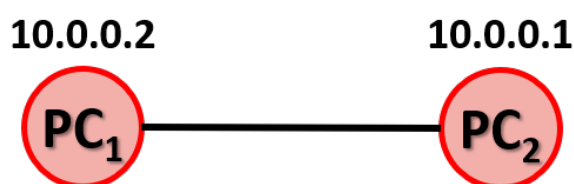
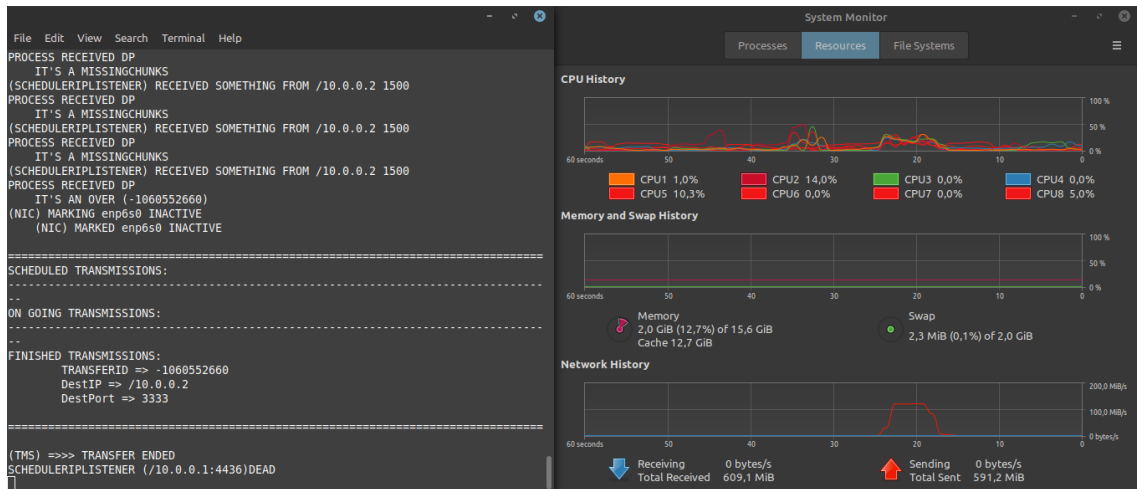
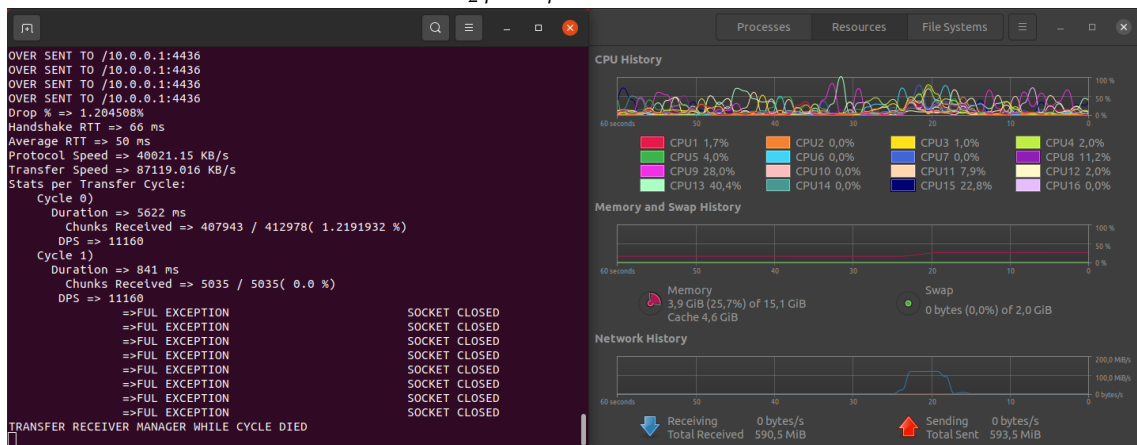


Figura 58: *Conexão direta entre os 2 Dispositivos*

Com esta configuração, a velocidade de conexão entre os Dispositivos é de 1000 *Mbps*. Com esta estrutura é pretendido a realização de 2 testes básicos deste Protocolo e Implementação:

- Realização de uma Transferências sem falhas. Este teste tem o objetivo secundário de observação de velocidade máxima prática desta Implementação;
- Realização de uma Transferência onde um dos Nodos é interrompido durante a realização desta e a consequente retoma quando possível.

O primeiro teste teve como Documento a ser transferido um Ficheiro *Zip* de teste com 512 *MBytes* para que possa ser observada a velocidade máxima sustida na transferência.

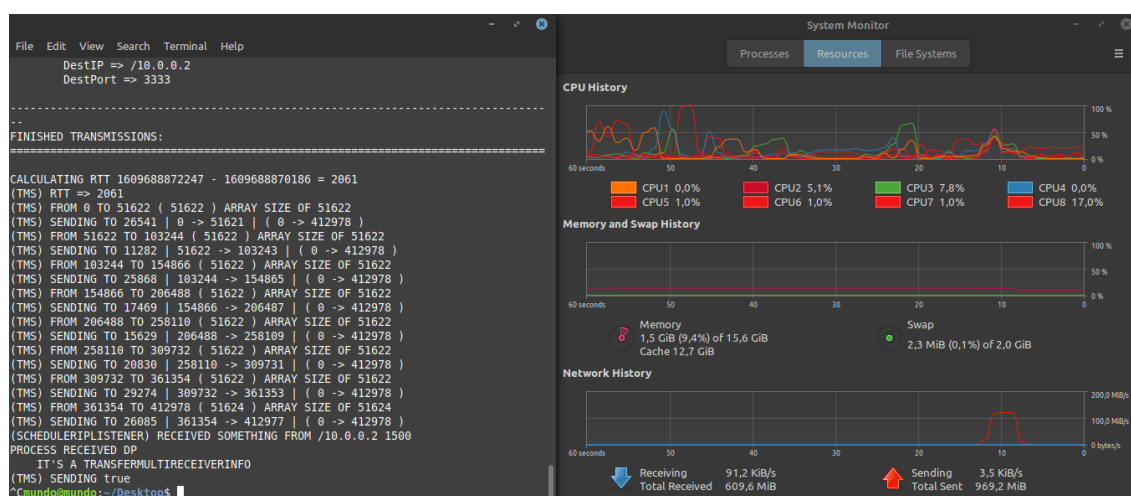
(a) PC₂ participando como Transmissor(b) PC₁ participando como Recetor**Figura 59:** Transferência de um Ficheiro de 512 MBytes numa ligação de 1000 Mbps estável

Neste teste foi possível observar (Figura 59b) uma Transferência que foi concluída com sucesso e realizada a uma média de velocidade observada pelo Recetor de aproximadamente 40000 KBytes/s (≈ 39 MBytes/s). Adicionalmente o Recetor faz a medição desta velocidade durante o Bloco *Transfer* do Protocolo, sendo que a velocidade aproximada observada neste foi de 87000 KBytes/s (≈ 85 MBytes/s). Esta disparidade acontece devido ao tempo que o Recetor demora a criar os Objectos *FastUnicastListener* após a receção do *Datagrama TMI* que dá início à Transferência. Também é possível observar na janela de utilização de recursos da Figura 59b que foi atingida uma velocidade ligeiramente superior a 100 MBytes/s dos 125 MBytes/s que esta ligação de 1000 Mbps teoricamente permite, indicando que os *Datagramas CM* estão a ser emitidos a uma velocidade adequada visto que apenas houve uma perda de $\approx 1,2\%$ destes *Datagramas*. Esta diferença entre a velocidade observada entre a janela de recursos utilizados e a medição feita pelo Recetor no Bloco *Transfer* indica que

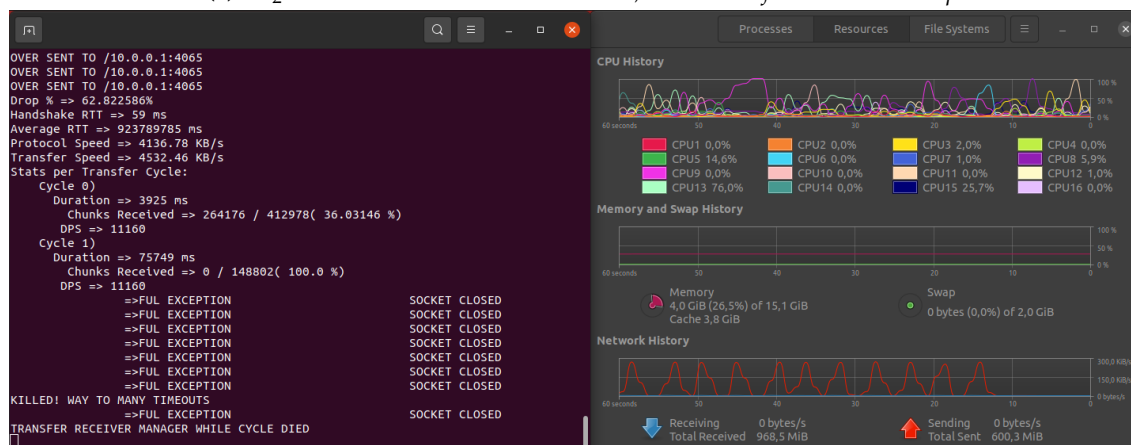
a Implementação da Classe *FastUnicastListener* ainda tem que ser otimizada de forma a aumentar a sua eficiência.

Uma forma de mitigar estas disparidades seria realizar a otimização referida durante a especificação referente à agregação de Portas de receção de *Datagramas* CM. Esta otimização também requer a otimização da Classe *FastUnicastSender* mas, com a sua realização, não seria necessária a criação de novos Objetos desta Classe no início de cada nova Transferência, reduzindo a disparidade entre a velocidade da Transferência e do Bloco *Transfer*.

O segundo teste realizado tem como Objetivo testar a capacidade do Protocolo e Protótipo de reiniciarem uma Transferência que foi interrompida. Também foi utilizado o mesmo Documento Zip de 512 *MBytes* nesta Transferência.



(a) PC₂ inicia a Transmissão do Documento, mas a Transferência é interrompida

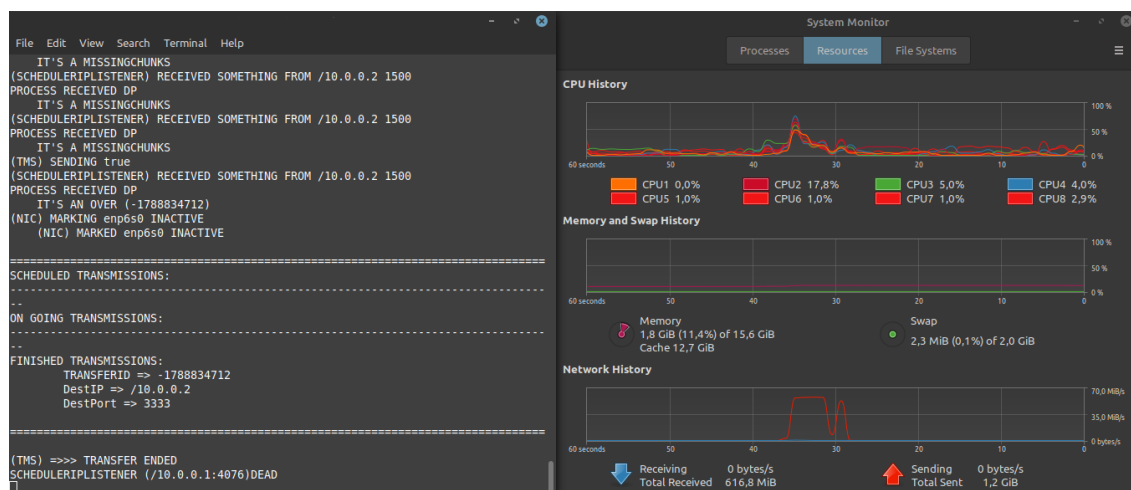


(b) PC₁ inicia a Receção do Documento mas deteta que houve uma interrupção.

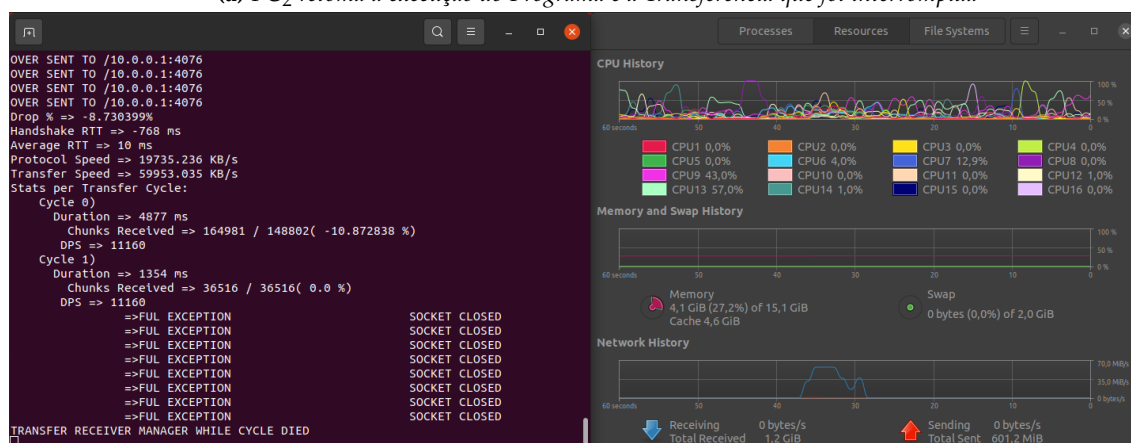
Figura 60: Início de uma Transferência e a sua interrupção.

Nas Imagens da Figura 60 pode ser observado que a troca de Dados entre os Dispositivos foi iniciada, mas o Programa do Transmissor foi interrompido (Figura 60a), o que levou o

Recetor a ativar todas as medidas de recuperação de falhas sem sucesso e consequentemente a interromper a Transferência.



(a) PC₂ retoma a execução do Programa e a Transferência que foi interrompida



(b) PC₁ recebe o pedido de Transmissão de um Documento que já possuía alguns Chunks

Figura 61: Retoma e conclusão de uma Transferência interrompida

Neste teste foi observado o início, interrupção, retoma e conclusão de uma Transferência, provando que os mecanismos do Protocolo e a Implementação permitem a interrupção completa de uma Transferência e a sua retoma quando for possível. É possível observar na Figura 61b que o Recetor detetou uma percentagem negativa de *Drops* no primeiro ciclo após a retoma da transferência, isto é devido à interrupção e necessita de ser observado para se perceber de onde este comportamento é originário.

7.1.2 3 Routers

Nesta Secção será utilizada uma Rede com uma estrutura, semelhante à que se encontra na Figura 62, onde se utilizam 3 *Routers* e onde os Dispositivos se encontram conectados aos *Routers* Folha. De forma a não lidar com as Rede NAT, o *Router_B* e *Router_C* têm a opção DMZ activada e a redireccionar tráfego para os Dispositivos a que cada um deles se encontra conectado. Visto que é pretendido testar a movimentação de Nodos na Rede, o Dispositivo *PC₁* pode-se deslocar entre o *Router_A* e o *Router_B*, sendo que será indicado quando esta alteração ocorre.

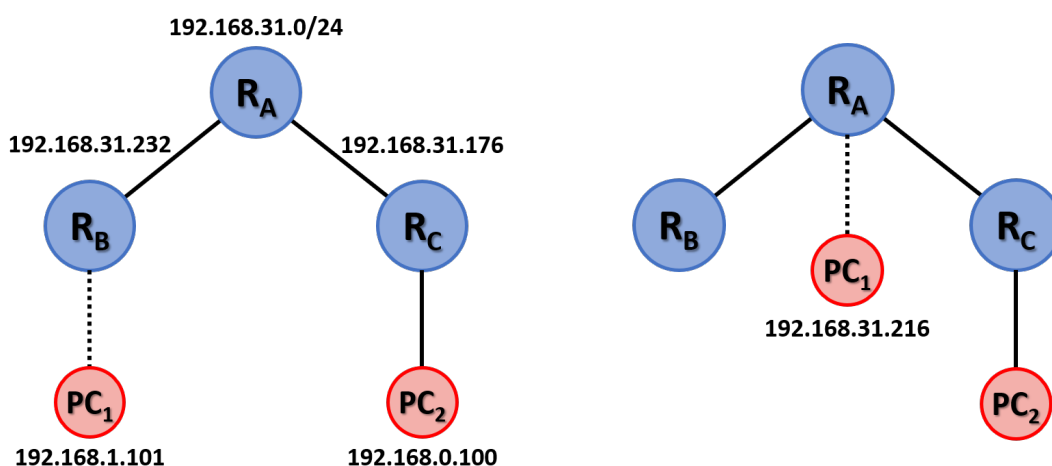
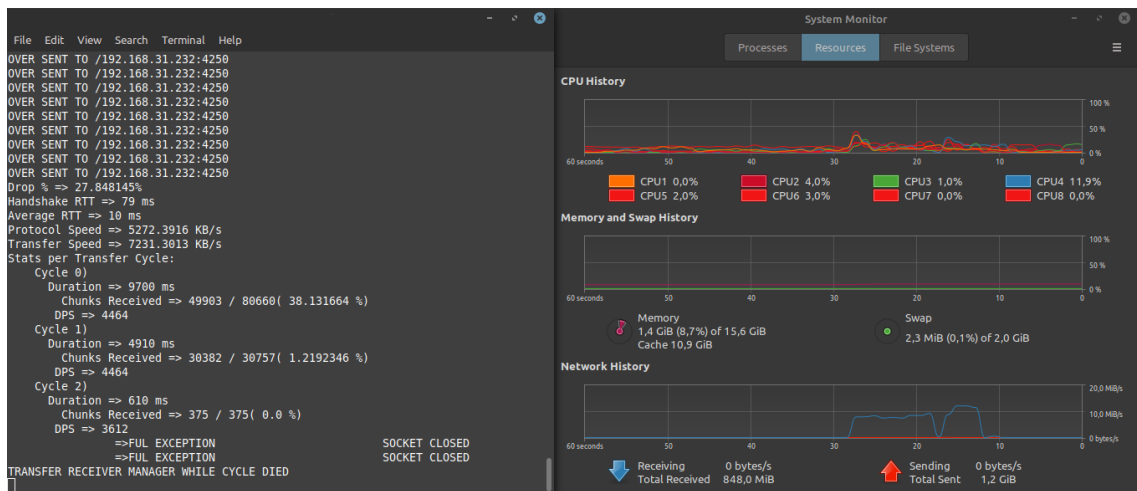
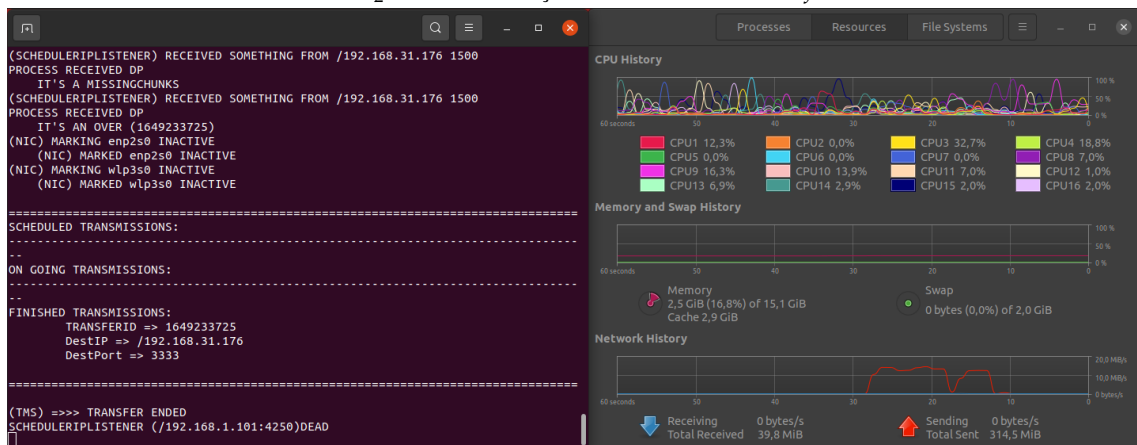


Figura 62: Estrutura da Rede utilizada para os testes que usam 3 Routers

Com esta estrutura é pretendido realizar os seguintes testes:

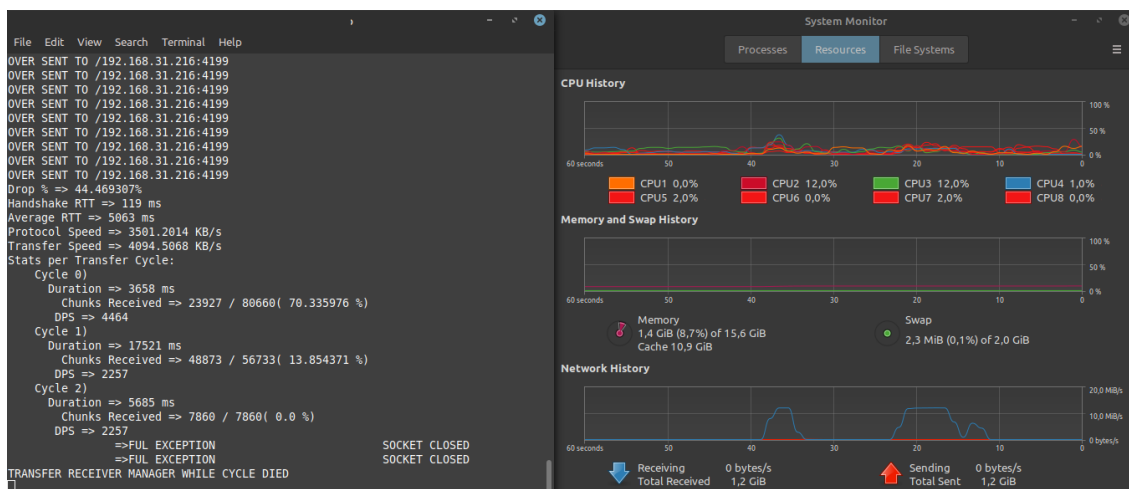
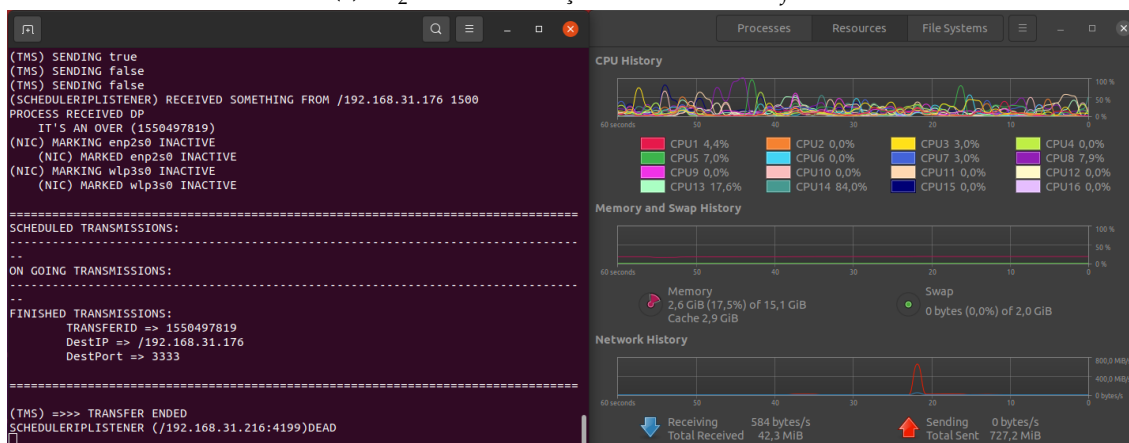
- Maior Perda de Datagramas utilizando *Wi-Fi*, testando os sistemas de recuperação;
- Movimentação do Nodo Recetor na Rede durante a realização de uma Transferência;
- Movimentação do Nodo Transmissor na Rede durante a realização de uma Transferência.

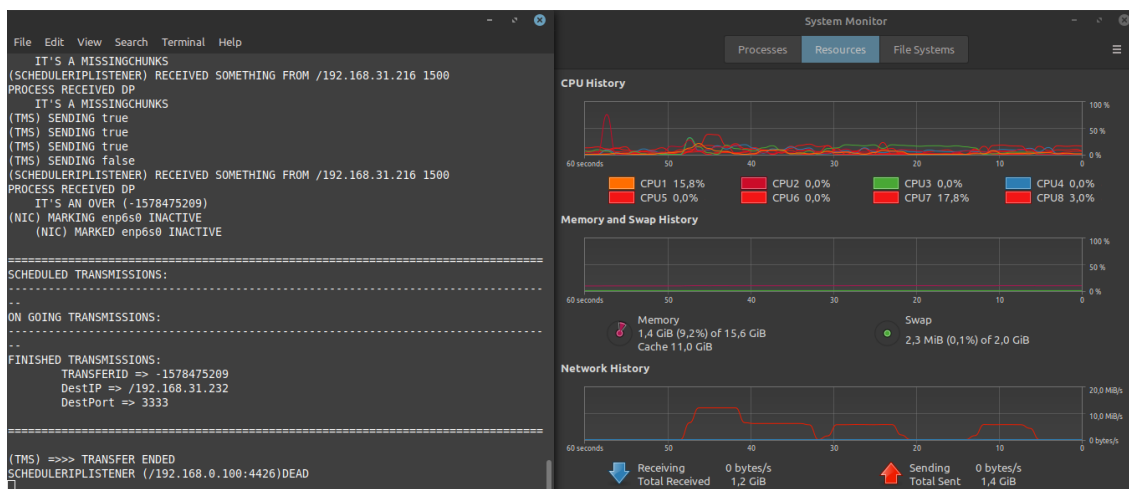
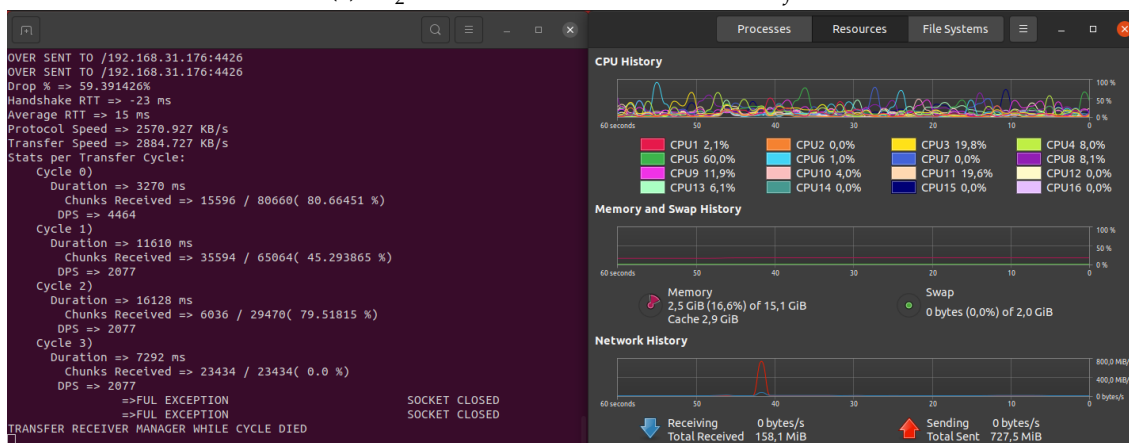
Com o primeiro teste é pretendido que haja uma grande perda de *Datagramas* durante a Transferência de forma a causar *stress* no mecanismo de recuperação de *Drops*, logo foi utilizado o *NIC Wireless* do *PC₁* para a realização deste. Para aumentar a quantidade de *Drops*, durante a realização deste teste foram ativadas todas as antenas dos *Routers* par que causassem interferência. Foi utilizado um Documento de teste *Zip* com o tamanho de 100 *MBytes* neste teste.

(a) PC_2 a realizar a receção de Dados de uma Transferência(b) PC_1 a realizar a transmissão de Dados de uma Transferência através de um NIC Wireless**Figura 63:** Transferência através de um NIC Wireless num ambiente com interferência

Como é possível observar na Figura 63, a Transferência foi concluída com sucesso, tendo havido uma perda média de 27,8% dos *Datagramas CM*, sendo que foram perdidos 38,13% dos *Datagramas CM* no primeiro ciclo do Bloco *Transfer* (Figura 63a).

Os 2 seguintes testes têm como objetivo demonstrar a Movimentação de Nodos durante uma Transferência e todos os mecanismos Implementados para que estas movimentações fossem possíveis. Inicialmente será feita a movimentação do Nodo Transmissor, seguido do teste de movimentação do Nodo Recetor. Para estes testes foi utilizado o Documento *Zip* de 100 *MBytes* para a Transferência.

(a) PC₂ a realizar a receção de Dados da Transferência(b) PC₁ a realizar a emissão de Dados da Transferência e eventual movimentação na Rede**Figura 64:** Transferência onde o Transmissor se move na Rede

(a) PC₂ a realizar a emissão de Dados da Transferência(b) PC₁ a realizar a receção de Dados da Transferência e eventual movimentação na Rede**Figura 65:** Transferência onde o Recetor se move na Rede

Ambos os testes foram concluídos com sucesso, sendo que as Transferências foram iniciadas, modificou-se a origem e o destino dos *Datagramas* respetivamente e foram concluídas antes que fosse atingido um estado em que um dos Nodos desistisse da realização da Transferência. Relativamente aos dados estatísticos recolhidos pelos Recetores destas transferências, é possível observar nas Figuras 64a e 65a que houve uma enorme perda de *Datagramas* CM no primeiro ciclo do Bloco *Transfer*. Isto acontece pois a partir do momento em que um dos Nodos intervenientes perde a conexão, o fluxo de *Datagramas* é 0. É de anotar que a medição de utilização de recursos do Sistema Operativo teve uma anomalia na deteção da velocidade de transmissão de Dados nos Nodos que se movimentaram durante a Transmissão.

7.1.3 1 Router

Com esta configuração da Figura 66 é pretendido realizar 1 teste que apenas requer uma conexão entre 2 Dispositivos. Este teste trata-se da detecção de Nodos na Rede utilizando *Grupos Multicast* e respostas *Unicast*.

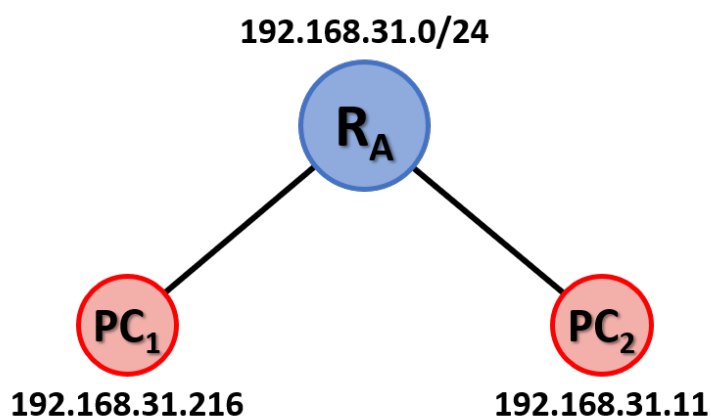


Figura 66: Estrutura da Rede Utilizada para o Teste de detecção de Nodos na Rede

Assim, foi utilizado o mecanismo de detecção de Nodos na Rede especificado e Implementado no Protótipo dando uso aos *Datagramas Knock*. Como foi referido na especificação desta troca de Informação, podem ser colocados Dados nos *Datagramas Knock*, mas neste teste esta característica não foi utilizada.

```

File Edit View Search Terminal Help
=====
1 - LOAD DOCUMENT jar
2 - LOAD DOCUMENT 100MB.zip
3 - LOAD DOCUMENT 512MB.zip
4 - LOAD MESSAGE
5 - START LISTENER
6 - SEND LOADED INFO
7 - START KNOCK MANAGER
8 - SEND KNOCK
9 - MOUNT DOC
0 - PRINT SCHEDULE
=====
8
MULTICAST SENT TO /233.252.18.250:6100
MAIN
  Origem : /192.168.31.216
  Tamanho dos Dados : 1500
=====
1 - LOAD DOCUMENT jar
2 - LOAD DOCUMENT 100MB.zip
3 - LOAD DOCUMENT 512MB.zip
4 - LOAD MESSAGE
5 - START LISTENER
6 - SEND LOADED INFO
7 - START KNOCK MANAGER
8 - SEND KNOCK
9 - MOUNT DOC
0 - PRINT SCHEDULE
=====

```

(a) PC_2 a emitir um Knock Multicast recebendo uma resposta Unicast do PC_1

```

0) enp2s0
  isWireless? false
  Addresses []
1) wlp3s0
  isWireless? true
  Addresses [/fe80:0:0:0:44e8:e583:8d81:4f79%wlp3s0, /192.168.31.216]
1
ESTA A TENTAR LIGAR O CHECKER PORQUE CRIOU UM KNOCKMANAGER NOVO
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
BBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBB
BB
=====
1 - LOAD DOCUMENT jar
2 - LOAD DOCUMENT 100MB.zip
3 - LOAD DOCUMENT 512MB.zip
4 - LOAD MESSAGE
5 - START LISTENER
6 - SEND LOADED INFO
7 - START KNOCK MANAGER
8 - SEND KNOCK
9 - MOUNT DOC
0 - PRINT SCHEDULE
=====
Recevi Knock Multicast
  Origem : /192.168.31.11
  Tamanho dos Dados : 1500

```

(b) PC_1 recebe um Knock Multicast do PC_2 emitindo uma resposta Unicast**Figura 67:** Detecção de Nós na mesma Rede utilizando o Mecanismo de troca de Datagramas Knock

Como é possível observar, os *Datagramas Knock* foram recebidos por ambos os Dispositivos, provando que este mecanismo pode ser usado para detecção de Nodos na Rede.

7.1.4 2 Routers

Nesta secção apenas se pretende provar que a detecção de Nodos na Rede utilizando os *Datagramas Knock* pode ser utilizado através de redes *NAT* e que isto pode ser utilizado como um Mecanismo para "contornar" estas Redes.

Assim foram utilizados 2 *Routers* onde ambos estão a funcionar como *Routers NAT*.

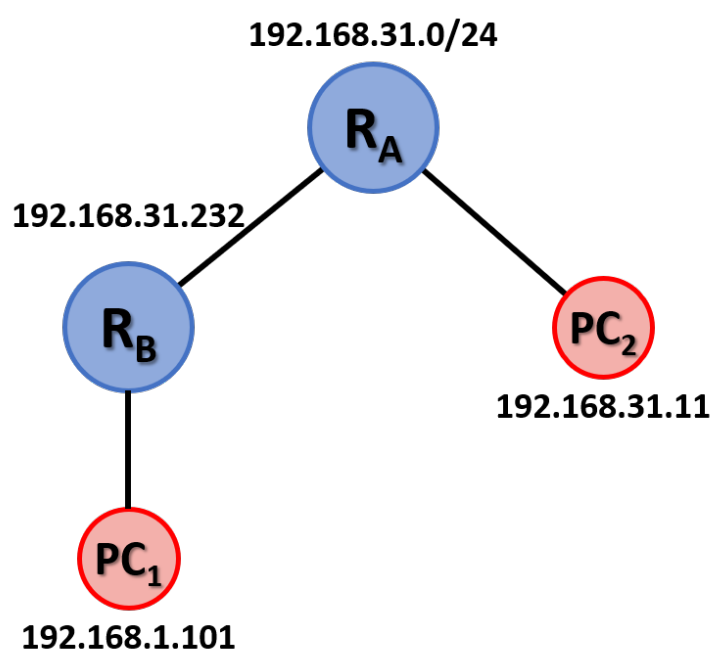


Figura 68: Estrutura da Rede para o Teste de contorno de Redes *NAT*

```

File Edit View Search Terminal Help
=====
1 - LOAD DOCUMENT jar
2 - LOAD DOCUMENT 100MB.zip
3 - LOAD DOCUMENT 512MB.zip
4 - LOAD MESSAGE
5 - START LISTENER
6 - SEND LOADED INFO
7 - START KNOCK MANAGER
8 - SEND KNOCK
9 - MOUNT DOC
0 - PRINT SCHEDULE
=====
8
MULTICAST SENT TO /233.252.18.250:6100
MAIN
  Origem : /192.168.31.232
  Tamanho dos Dados : 1500
=====
1 - LOAD DOCUMENT jar
2 - LOAD DOCUMENT 100MB.zip
3 - LOAD DOCUMENT 512MB.zip
4 - LOAD MESSAGE
5 - START LISTENER
6 - SEND LOADED INFO
7 - START KNOCK MANAGER
8 - SEND KNOCK
9 - MOUNT DOC
0 - PRINT SCHEDULE
=====

```

(a) PC_2 a emitir um Knock Multicast recebendo uma resposta Unicast do PC_1 que possui o Endereço "público" do Router_B

```

0) enp2s0
  isWireless? false
  Addresses []
1) wlp3s0
  isWireless? true
  Addresses [/fe80:0:0:0:1c9e:9a2c:9b5f:ab67%wlp3s0, /192.168.1.101]
1
ESTA A TENTAR LIGAR O CHECKER PORQUE CRIOU UM KNOCKMANAGER NOVO
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
BBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBB
BB
=====
1 - LOAD DOCUMENT jar
2 - LOAD DOCUMENT 100MB.zip
3 - LOAD DOCUMENT 512MB.zip
4 - LOAD MESSAGE
5 - START LISTENER
6 - SEND LOADED INFO
7 - START KNOCK MANAGER
8 - SEND KNOCK
9 - MOUNT DOC
0 - PRINT SCHEDULE
=====
Recevi Knock Multicast
  Origem : /192.168.31.11
  Tamanho dos Dados : 1500

```

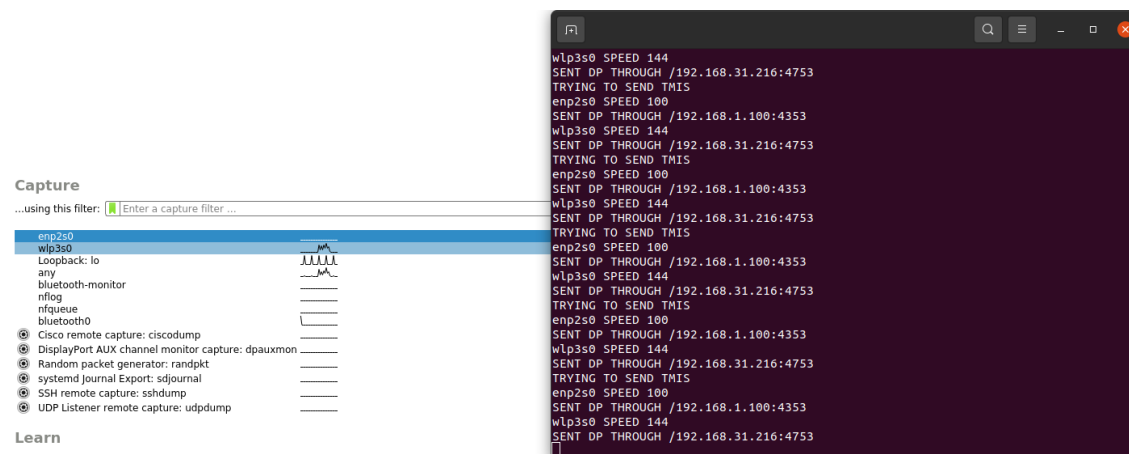
(b) PC_1 recebe um Knock Multicast do PC_2 indicando que estes não são bloqueados pelos Routers NAT. Após a recepção este emite uma resposta Unicast

Figura 69: Detecção de Nodos na Rede utilizando Datagramas Knock através de uma Rede NAT

Como se vê no output do Protótipo, esta detecção de Nós foi concluída com sucesso, abrindo a possibilidade de comunicação através de Redes *NAT*.

Adicionalmente foi realizado um teste para testar a funcionalidade de detecção de diferentes caminhos utilizando múltiplos *NICs*. Por isso o *PC₁* foi ligado a 2 *Routers* independentes de forma a ter 2 conexões a Redes Distintas.

Para melhor observar este teste, foi utilizado o Programa *Wireshark* para verificar se os *Datagramas TMI* estavam a ser emitidos por 2 *NICs* diferentes.



(a) Actividade no *NIC enp2s0* é inexistente enquanto o *NIC wlp3s0* apresenta atividade

*wlp3s0						
No.	Time	Source	Destination	Protocol	Length	Info
36	2.896545148	192.168.31.216	192.168.31.1	TCP	66	57058 → 80 [ACK] Seq=88 Ack=428 Win=64128 Len=0 TSval=1050699...
37	2.896573673	192.168.31.1	192.168.31.216	TCP	66	80 → 57058 [FIN, ACK] Seq=428 Ack=88 Win=14592 Len=0 TSval=42...
38	2.896778956	192.168.31.216	192.168.31.1	TCP	66	57058 → 80 [FIN, ACK] Seq=88 Ack=429 Win=64128 Len=0 TSval=10...
39	2.900616313	192.168.31.1	192.168.31.216	TCP	66	80 → 57058 [ACK] Seq=429 Ack=89 Win=14592 Len=0 TSval=4294949...
40	3.966987992	192.168.31.216	224.0.0.251	MDNS	183	Standard query 0x0000 PTR _nfs._tcp.local. "QM" question PTR ...
41	5.620919563	AzureWav_98:be:89	Broadcast	ARP	42	Who has 192.168.31.111? Tell 192.168.1.100
42	6.645463118	AzureWav_98:be:89	Broadcast	ARP	42	Who has 192.168.31.111? Tell 192.168.31.216
43	7.669461053	AzureWav_98:be:89	Broadcast	ARP	42	Who has 192.168.31.111? Tell 192.168.31.216
44	11.624097763	AzureWav_98:be:89	Broadcast	ARP	42	Who has 192.168.31.111? Tell 192.168.1.100
45	12.629519750	AzureWav_98:be:89	Broadcast	ARP	42	Who has 192.168.31.111? Tell 192.168.31.216
46	13.653460295	AzureWav_98:be:89	Broadcast	ARP	42	Who has 192.168.31.111? Tell 192.168.31.216
47	17.626851776	AzureWav_98:be:89	Broadcast	ARP	42	Who has 192.168.31.111? Tell 192.168.1.100
48	18.645775636	AzureWav_98:be:89	Broadcast	ARP	42	Who has 192.168.31.111? Tell 192.168.31.216
49	19.669479418	AzureWav_98:be:89	Broadcast	ARP	42	Who has 192.168.31.111? Tell 192.168.31.216
50	23.629599419	AzureWav_98:be:89	Broadcast	ARP	42	Who has 192.168.31.111? Tell 192.168.1.100
51	24.661459352	AzureWav_98:be:89	Broadcast	ARP	42	Who has 192.168.31.111? Tell 192.168.31.216
52	25.685470238	AzureWav_98:be:89	Broadcast	ARP	42	Who has 192.168.31.111? Tell 192.168.31.216
53	29.631971851	AzureWav_98:be:89	Broadcast	ARP	42	Who has 192.168.31.111? Tell 192.168.1.100
54	30.645600529	AzureWav_98:be:89	Broadcast	ARP	42	Who has 192.168.31.111? Tell 192.168.31.216
55	31.669489752	AzureWav_98:be:89	Broadcast	ARP	42	Who has 192.168.31.111? Tell 192.168.31.216
56	35.634748840	AzureWav_98:be:89	Broadcast	ARP	42	Who has 192.168.31.111? Tell 192.168.1.100
57	36.661464718	AzureWav_98:be:89	Broadcast	ARP	42	Who has 192.168.31.111? Tell 192.168.31.216
58	37.685460129	AzureWav_98:be:89	Broadcast	ARP	42	Who has 192.168.31.111? Tell 192.168.31.216
59	41.637699685	AzureWav_98:be:89	Broadcast	ARP	42	Who has 192.168.31.111? Tell 192.168.1.100
60	42.645595518	AzureWav_98:be:89	Broadcast	ARP	42	Who has 192.168.31.111? Tell 192.168.31.216
61	43.669463439	AzureWav_98:be:89	Broadcast	ARP	42	Who has 192.168.31.111? Tell 192.168.31.216
62	47.640364597	AzureWav_98:be:89	Broadcast	ARP	42	Who has 192.168.31.111? Tell 192.168.1.100
63	48.661465958	AzureWav_98:be:89	Broadcast	ARP	42	Who has 192.168.31.111? Tell 192.168.31.216
64	49.685476093	AzureWav_98:be:89	Broadcast	ARP	42	Who has 192.168.31.111? Tell 192.168.31.216
65	53.643343276	AzureWav_98:be:89	Broadcast	ARP	42	Who has 192.168.31.111? Tell 192.168.1.100
66	54.645488811	AzureWav_98:be:89	Broadcast	ARP	42	Who has 192.168.31.111? Tell 192.168.31.216
67	55.669485454	AzureWav_98:be:89	Broadcast	ARP	42	Who has 192.168.31.111? Tell 192.168.31.216

(b) O *NIC wlp3s0* está a realizar ambas emissões, mesmo com um Endereço Origem que não possui

Figura 70: Visualização da saída da *Datagramas TMI* usando *Wireshark*

Como se pode ver nas capturas realizadas no *Wireshark* (Figura 70b), esta funcionalidade não ficou bem desenvolvida, fazendo com que apenas se possa iniciar uma Transferência por 1 *NIC*. Este problema está relacionado com a forma como se relaciona *DatagramSockets* com *NICs* neste Protótipo, pois isto foi realizado utilizando o método *bind(InetAddress)* na tentativa de forçar um *Datagrama* emitido por essa *DatagramSocket* a ser emitido pelo *NIC* que possuía o Endereço associado a esta, mas como se pode observar não foi conseguido.

7.2 DISCUSSÃO

Tendo em consideração os testes realizados, considera-se que o desenho do Protocolo e a Implementação do Protótipo foram bem-sucedidos visto que os principais objetivos estabelecidos no início foram atingidos. Apesar disto existem múltiplos pontos que podem ser melhorados, desde funcionalidades que não funcionam como esperado (detecção de múltiplos caminhos entre Nodos usando múltiplos *NICs*) a pequenos detalhes da Implementação. Um destes detalhes é a libertação de memória *RAM*, como se pode ver nas Figuras relativas à Receções de Dados, após a Transferência ser concluída, a memória *RAM* utilizada durante o processo de receção não é libertada. Outro ponto que se pode melhorar mesmo nesta Implementação é o aumento da eficiência e a consequente redução de consumo de recursos. Adicionalmente é possível realizar múltiplas otimizações, algumas delas já detetadas e referidas, que irão auxiliar neste aumento de performance e redução de utilização de recursos.

CONCLUSÃO E PROSPEÇÃO DE TRABALHOS FUTUROS

Ao longo do Trabalho realizado neste Projeto, foi Desenhado um Protocolo de Transferência de Dados que, após 5 iterações, tinha a capacidade de resistir a *Drops* durante transferências, movimentação dos Nodos intervenientes, interrupções do Protocolo por completo e a capacidade de retomar a sua execução. Para além disto, este Protocolo tem a capacidade de cancelar transferências duplicadas desnecessárias e descobrir Nodos na Rede dando uso a Grupos *Multicast*. Ainda se Implementou um Protótipo otimizador deste Protocolo que implementou todas estas características e foi posto à prova em Testes realizados em cenários reais obtendo resultados excelentes. Mesmo não tendo sido integrado no Protótipo, ainda foram abordados temas como Redes *NAT* e mecanismos úteis para o bom funcionamento do Protocolo nestas, a fácil inclusão de Transferências *Multipath* e *Multitransmitter* neste Protocolo, aspetos de Segurança, estatísticas avançadas para o controlo de fluxo de *Datagramas* e a compressão de dados. Este Protocolo de Transferência de Dados apresenta todas estas características e mesmo assim consegue ser apto à transferência de dados em Arquiteturas de Redes *DTN* que são extremamente exigentes.

Adicionalmente foi desenvolvida uma Arquitetura de Redes *AdHoc* que tira partido do Protocolo de Transferência de Dados Tolerante a Falhas e que utiliza a noção de Nodo Central e Nodos Vizinhos distribuídos em Níveis conforme a sua distância ao Nodo Central. Desta forma é possível criar uma Estrutura simples que facilita a troca de Informação entre Nodos destas Redes.

Apesar de todo o trabalho realizado, ainda existem muitos pontos que necessitam de ser mais explorados, desenvolvidos e otimizados. Especificamente, no Protótipo desenvolvido existem pontos óbvios como:

- a deteção de múltiplos caminhos entre Nodos usando múltiplos *NICs*;
- Uma anomalia de chegada de um grande número de *Datagramas* duplicados que foram observados nos testes realizados e que necessitam de ser analisados e, se possível, corrigidos.

Certamente que existirão outras anomalias que não foram detetadas durante o processo de Implementação e a realização de testes que também necessitaram de ser corrigidas.

Adicionalmente é necessário estudar quais serão as estatísticas mais importantes para o controlo correto das Transferências de Dados e como as aplicar nos mecanismos desenvolvidos. A Secção das Estatísticas estudadas (3.3) e Implementadas cumpre a sua função num Protótipo de demonstração mas para um possível produto futuro seria uma obrigatoriedade a melhoria desta área devido ao seu grande impacto no funcionamento do Protocolo, desde tempos de *Timeouts*, velocidades de Transferências até à deteção de congestão da Rede.

Em relação a mecanismos que poderiam ter sido desenvolvidos e Implementados neste Protótipo, são de destacar os seguintes:

- Compressão de dados para a redução do número de *Chunks* relativos a um conjunto de Dados;
- Introdução de uma Camada de Segurança;
- Introdução de um mecanismo de "contorno" de Redes *NAT*;
- Implementação de Transferências *Multipath* e *Multitransmitter*.

Das muitas que seriam possíveis desenvolver e Implementar, estas são as que considero mais críticas, pois tornam possível a utilização deste Protocolo e Protótipo em Redes que de momento não seria possível, aumentariam a eficiência das Transferências e seria introduzida a capacidade de transmissão de Dados entre Nodos de forma segura.

Por fim, encontram-se todas as otimizações possíveis que, para além de poderem aumentar a velocidade a que uma transferência é realizada, também permitem a redução de recursos necessários para a realização de Transferências, permitindo maiores velocidades em Transferências entre Dispositivos que possam não ser tão atuais como os que foram utilizados nos testes. Dentro destas otimizações destacam-se as seguintes:

- Analisar o Protocolo de forma a reconstruí-lo de uma forma mais simplificada;
- Analisar a lógica Implementada de forma a agilizar o código desenvolvido, remover mecanismos não utilizados e remover lógica desnecessária;
- Remover o máximo de *Timers* existentes de forma a construir um código mais sólido e reduzir o consumo de recursos desnecessários;
- Melhorar a Implementação da Classe *FastUnicastListener* de forma a melhorar a sua eficiência, permitindo a mudança de filosofia de criação de Objectos, fixando o número de Objetos desta Classe à capacidade de receção de *Datagramas* dos Dispositivos. Esta melhoria também permitiria uma medição de estatísticas mais "relaxada" visto que haveria uma maior margem de eficiência do código.

Uma mudança que poderia ser feita é a linguagem de Programação a utilizar na futura Implementação deste produto, sendo procurada uma linguagem que sacrificasse a comodidade oferecida pelo *Java* por uma melhor eficiência.

Visto que não foi possível desenvolver e Implementar todos os Protótipos propostos no início deste Documento, é natural que como trabalho futuro se inclua:

- Implementação da Arquitetura *Ad Hoc*;
- Especificação e Implementação da Arquitetura *DTN*

Finalmente, visto que este tipo de comunicação entre Dispositivos é útil em mais situações que as inicialmente pensadas, seria interessante no futuro construir uma versão deste Protocolo que realizasse a troca de Dados entre Dispositivos utilizando o conceito de Sessão, mudando o foco do Protocolo da Transferência de um Documento ou Mensagem para a Transferência de múltiplos destes, facilitando a Troca de dados de forma prolongada entre Nodos. Adicionalmente seria interessante a Implementação de um Protocolo para troca de Dados em Tempo Real, ou seja, num estilo de uma *Stream* de Dados.

BIBLIOGRAFIA

- [1] Yung-Wei Kao, ChiaFeng Lin, Kuei-An Yang, and Shyan-Ming Yuan. A web-based, offline-able, and personalized runtime environment for executing applications on mobile devices. *Computer Standards & Interfaces*, 34(1):212–224, 2012. ISSN 0920-5489. doi: <https://doi.org/10.1016/j.csi.2011.08.006>. URL <https://www.sciencedirect.com/science/article/pii/S0920548911000869>.
- [2] Wolfgang Kiess and Martin Mauve. A survey on real-world implementations of mobile ad-hoc networks. *Ad Hoc Networks*, 5(3):324–339, 2007. ISSN 1570-8705. doi: <https://doi.org/10.1016/j.adhoc.2005.12.003>. URL <https://www.sciencedirect.com/science/article/pii/S1570870505001149>.
- [3] M. Al-Kuwaiti, N. Kyriakopoulos, and S. Hussein. A comparative analysis of network dependability, fault-tolerance, reliability, security, and survivability. *IEEE Communications Surveys Tutorials*, 11(2):106–124, 2009. doi: 10.1109/SURV.2009.090208.
- [4] Narjess Ayari, Denis Barbaron, Laurent Lefevre, and Pascale Primet. Fault tolerance for highly available internet services: concepts, approaches, and issues. *IEEE Communications Surveys Tutorials*, 10(2):34–46, 2008. doi: 10.1109/COMST.2008.4564478.
- [5] Collaboration Intelligence company Vyopta. Troubleshooting packet loss: How much is an acceptable amount?, December 2018. URL <https://www.vyopta.com/blog/video-conferencing/understanding-packet-loss/>. Accessed: 2020-04-04.
- [6] Robin Ödling. Data transfer over high latency and high packet loss networks, 2016. URL <https://www.diva-portal.org/smash/get/diva2:1051983/FULLTEXT01.pdf>.
- [7] Leigh Torgerson, Scott C. Burleigh, Howard Weiss, Adrian J. Hooke, Kevin Fall, Dr. Vinton G. Cerf, Keith Scott, and Robert C. Durst. Delay-Tolerant Networking Architecture. RFC 4838, April 2007. URL <https://rfc-editor.org/rfc/rfc4838.txt>.
- [8] InterPlanetary Networking Special Interest Group (IPNSIG). Introduction to delay & disruption tolerant networking (dtn), 2020. URL <http://ipnsig.org/introducing-delay-disruption-tolerant-networking-dtn/>. Accessed: 2020-01-20.
- [9] Artemios G Voyiatzis. A survey of delay- and disruption-tolerant networking applications, 01 2012.

- [10] Keith Scott and Scott C. Burleigh. Bundle Protocol Specification. RFC 5050, November 2007. URL <https://rfc-editor.org/rfc/rfc5050.txt>.
- [11] Wesley Eddy and Elwyn B. Davies. Using Self-Delimiting Numeric Values in Protocols. RFC 6256, May 2011. URL <https://rfc-editor.org/rfc/rfc6256.txt>.
- [12] Mark Nottingham. URI Design and Ownership. RFC 8820, June 2020. URL <https://rfc-editor.org/rfc/rfc8820.txt>.
- [13] Tim Berners-Lee, Roy T. Fielding, and Larry M. Masinter. Uniform Resource Identifier (URI): Generic Syntax. RFC 3986, January 2005. URL <https://rfc-editor.org/rfc/rfc3986.txt>.
- [14] Internet Assigned Numbers Authority (IANA). Uniform resource identifier (uri) schemes, January 2020. URL <https://www.iana.org/assignments/uri-schemes/uri-schemes.xhtml>. Accessed: 2020-01-04.
- [15] Dave Thaler, Tony Hansen, and Ted Hardie. Guidelines and Registration Procedures for URI Schemes. RFC 7595, June 2015. URL <https://rfc-editor.org/rfc/rfc7595.txt>.
- [16] Mark Nottingham. Well-Known Uniform Resource Identifiers (URIs). RFC 8615, May 2019. URL <https://rfc-editor.org/rfc/rfc8615.txt>.
- [17] Michele Polese, Federico Chiariotti, Elia Bonetto, Filippo Rigotto, Andrea Zanella, and Michele Zorzi. A survey on recent advances in transport layer protocols. *IEEE Communications Surveys Tutorials*, 21(4):3584–3608, 2019. doi: 10.1109/COMST.2019.2932905.
- [18] Transmission Control Protocol. RFC 793, September 1981. URL <https://rfc-editor.org/rfc/rfc793.txt>.
- [19] Ethan Blanton, Dr. Vern Paxson, and Mark Allman. TCP Congestion Control. RFC 5681, September 2009. URL <https://rfc-editor.org/rfc/rfc5681.txt>.
- [20] Andrei Gurtov, Tom Henderson, Sally Floyd, and Yoshifumi Nishida. The NewReno Modification to TCP's Fast Recovery Algorithm. RFC 6582, April 2012. URL <https://rfc-editor.org/rfc/rfc6582.txt>.
- [21] User Datagram Protocol. RFC 768, August 1980. URL <https://rfc-editor.org/rfc/rfc768.txt>.
- [22] Lars Eggert, Gorrry Fairhurst, and Greg Shepherd. UDP Usage Guidelines. RFC 8085, March 2017. URL <https://rfc-editor.org/rfc/rfc8085.txt>.

- [23] Internet Assigned Numbers Authority (IANA). Ipv4 multicast address space registry, January 2020. URL <https://www.iana.org/assignments/multicast-addresses/multicast-addresses.xhtml>. Accessed: 2020-01-04.
- [24] David Meyer, Michelle Cotton, and Leo Vegoda. IANA Guidelines for IPv4 Multicast Address Assignments. RFC 5771, March 2010. URL <https://rfc-editor.org/rfc/rfc5771.txt>.
- [25] Jana Iyengar and Martin Thomson. QUIC: A UDP-Based Multiplexed and Secure Transport. RFC 9000, May 2021. URL <https://rfc-editor.org/rfc/rfc9000.txt>.
- [26] Matt Holdrege and Pyda Srisuresh. IP Network Address Translator (NAT) Terminology and Considerations. RFC 2663, August 1999. URL <https://rfc-editor.org/rfc/rfc2663.txt>.
- [27] Kjeld Borch Egevang and Pyda Srisuresh. Traditional IP Network Address Translator (Traditional NAT). RFC 3022, January 2001. URL <https://rfc-editor.org/rfc/rfc3022.txt>.
- [28] Daniel Maier, Oliver Haase, Jürgen Wäsch, and Marcel Waldvogel. A comparative analysis of nat hole punching. *HTWG FORUM - Das Forschungsmagazin der HTWG Konstanz*, pages 40–48, 2011.
- [29] Daniel Maier, Oliver Haase, Jürgen Wäsch, and Marcel Waldvogel. Nat hole punching revisited. In *2011 IEEE 36th Conference on Local Computer Networks*, pages 147–150, 2011. doi: 10.1109/LCN.2011.6115173.
- [30] Ha Tran Thi Thu, Jaehyung Park, Yonggwon Won, and Jinsul Kim. Combining stun protocol and udp hole punching technique for peer-to-peer communication across network address translation. In *2014 International Conference on IT Convergence and Security (ICITCS)*, pages 1–4, 2014. doi: 10.1109/ICITCS.2014.7021753.
- [31] Toerless Eckert and Dan Wing. IP Multicast Requirements for a Network Address Translator (NAT) and a Network Address Port Translator (NAPT). RFC 5135, February 2008. URL <https://rfc-editor.org/rfc/rfc5135.txt>.
- [32] Dr. Steve E. Deering and Bob Hinden. Internet Protocol, Version 6 (IPv6) Specification. RFC 8200, July 2017. URL <https://rfc-editor.org/rfc/rfc8200.txt>.
- [33] Dr. Thomas Narten, Tatsuya Jinmei, and Dr. Susan Thomson. IPv6 Stateless Address Autoconfiguration. RFC 4862, September 2007. URL <https://rfc-editor.org/rfc/rfc4862.txt>.
- [34] Eric Rescorla and Brian Korver. Guidelines for Writing RFC Text on Security Considerations. RFC 3552, July 2003. URL <https://rfc-editor.org/rfc/rfc3552.txt>.

- [35] Hal Finney, Lutz Donnerhacke, Jon Callas, Rodney L. Thayer, and David Shaw. OpenPGP Message Format. RFC 4880, November 2007. URL <https://rfc-editor.org/rfc/rfc4880.txt>.
- [36] Collaboration Intelligence company Vyopta. Openpgp software: Openpgp is a non-proprietary format for authenticating or encrypting data, using public key cryptography, June 2020. URL <https://www.openpgp.org/>. Accessed: 2020-06-01.
- [37] K. Robert Glenn and Cheryl R. Madson. The Use of HMAC-SHA-1-96 within ESP and AH. RFC 2404, November 1998. URL <https://rfc-editor.org/rfc/rfc2404.txt>.
- [38] Mihir Bellare, Ran Canetti, and Hugo Krawczyk. Message authentication using hash functions: The hmac construction. *RSA Laboratories' CryptoBytes*, 2(1):12–15, 1996.
- [39] Mauro Conti, Nicola Dragoni, and Viktor Lesyk. A survey of man in the middle attacks. *IEEE Communications Surveys Tutorials*, 18(3):2027–2051, 2016. doi: 10.1109/COMST.2016.2548426.
- [40] Meriem Bettayeb, Sara Ghunaim, Nour Mohamed, and Qassim Nasir. Error correction codes in wireless sensor networks: A systematic literature review. In *2019 International Conference on Communications, Signal Processing, and their Applications (ICCSPA)*, pages 1–6, 2019. doi: 10.1109/ICCSPA.2019.8713725.

