



UNIVERSIDADE FEDERAL RURAL DO SEMI-ÁRIDO
CAMPUS PAU DOS FERROS
DEPARTAMENTO DE ENGENHARIAS E TECNOLOGIA
CURSO DE GRADUAÇÃO EM TECNOLOGIA DA INFORMAÇÃO

JOSAIAS DE MOURA SILVA

**UTILIZANDO O PROTOCOLO BITTORRENT DHT PARA VIABILIZAR
CONECTIVIDADE FIM-A-FIM DE PROPÓSITO GERAL EM REDES COM
SERVIDORES NAT**

PAU DOS FERROS – RN

2018

JOSAIAS DE MOURA SILVA

**UTILIZANDO O PROTOCOLO BITTORRENT DHT PARA VIABILIZAR
CONECTIVIDADE FIM-A-FIM DE PROPÓSITO GERAL EM REDES COM
SERVIDORES NAT**

Trabalho de Conclusão de Curso apresentado ao Curso de Graduação em Tecnologia da Informação do Departamento de Engenharias e Tecnologia da Universidade Federal Rural Do Semi-Árido, como requisito parcial à obtenção do grau de bacharel em Tecnologia da Informação.

Orientador: Prof. Me. Marco Diego Aurélio Mesquita

Co-Orientador: Prof. Dr. Vinícius Samuel Valério de Souza

PAU DOS FERROS – RN

2018

© Todos os direitos estão reservados a Universidade Federal Rural do Semi-Árido. O conteúdo desta obra é de inteira responsabilidade do (a) autor (a), sendo o mesmo, passível de sanções administrativas ou penais, caso sejam infringidas as leis que regulamentam a Propriedade Intelectual, respectivamente, Patentes: Lei nº 9.279/1996 e Direitos Autorais: Lei nº 9.610/1998. O conteúdo desta obra tomar-se-á de domínio público após a data de defesa e homologação da sua respectiva ata. A mesma poderá servir de base literária para novas pesquisas, desde que a obra e seu (a) respectivo (a) autor (a) sejam devidamente citados e mencionados os seus créditos bibliográficos.

S586u Silva, Josaias de Moura.
UTILIZANDO O PROTOCOLO BITTORRENT DHT PARA
VIABILIZAR CONECTIVIDADE FIM-A-FIM DE PROPÓSITO
GERAL EM REDES COM SERVIDORES NAT / Josaias de
Moura Silva. - 2018.
51 f. : il.

Orientador: Marco Diego Aurélio Mesquita.
Coorientador: Vinícius Samuel Valério de Souza.
Monografia (graduação) - Universidade Federal
Rural do Semi-árido, Curso de Tecnologia da
Informação, 2018.

1. BitTorrent DHT. 2. UDP Hole Punching. 3.
Peer-to-Peer. 4. Node.js. I. Mesquita, Marco
Diego Aurélio, orient. II. Souza, Vinícius Samuel
Valério de, co-orient. III. Título.

O serviço de Geração Automática de Ficha Catalográfica para Trabalhos de Conclusão de Curso (TCC's) foi desenvolvido pelo Instituto de Ciências Matemáticas e de Computação da Universidade de São Paulo (USP) e gentilmente cedido para o Sistema de Bibliotecas da Universidade Federal Rural do Semi-Árido (SISBI-UFERSA), sendo customizado pela Superintendência de Tecnologia da Informação e Comunicação (SUTIC) sob orientação dos bibliotecários da instituição para ser adaptado às necessidades dos alunos dos Cursos de Graduação e Programas de Pós-Graduação da Universidade.

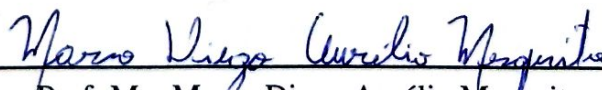
JOSAIAS DE MOURA SILVA

**UTILIZANDO O PROTOCOLO BITTORRENT DHT PARA VIABILIZAR
CONECTIVIDADE FIM-A-FIM DE PROPÓSITO GERAL EM REDES COM
SERVIDORES NAT**

Trabalho de Conclusão de Curso apresentado
ao Curso de Graduação em Tecnologia da
Informação do Departamento de Engenharias
e Tecnologia da Universidade Federal Rural
Do Semi-Árido, como requisito parcial à
obtenção do grau de bacharel em Tecnologia da
Informação.

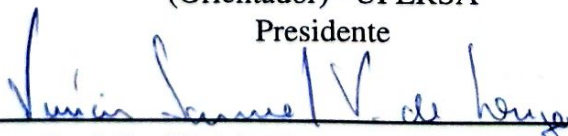
Aprovada em: 14 / 09 / 2018

BANCA EXAMINADORA



Prof. Me. Marco Diego Aurélio Mesquita
(Orientador)– UFERSA

Presidente



Prof. Dr. Vinícius Samuel Valério de Souza
(Co-Orientador)– UFERSA

Membro Examinador



Profa. Dra. Laysa Mabel de Oliveira Fontes –
UFERSA

Membro Examinador

Dedico este trabalho a minha família, colegas,
amigos e professores.

AGRADECIMENTOS

Agradeço primeiramente a meus pais, **Jacinto** e **Jacinta**, por todo apoio, carinho, amor e preocupação durante toda minha vida. Meus pilares.

A minha irmã **Jesana**, por todo apoio, carinho e confiança, mesmo estando distante sempre posso contar contigo.

A minha esposa **Patrícia**, por sempre acreditar em mim quando até eu duvido, por fazer parte da minha vida e estar do meu lado, por todo carinho e amor que você transparece. Sim, desculpe pelas luzes e barulhos na cadeira tarde da noite :P

A minha sogros **Célia** e **Cesar**, que me abraçaram na vida como mais um filho, obrigado pelo apoio e carinho.

Ao meu orientador **Marco** e co-orientador **Vinícius**, obrigado pela paciência, dedicação e contribuições que foram fundamentais para esta realização.

A minha banca de defesa, pelas contribuições e críticas que me proporcionaram um grande aprendizado.

As meus familiares, tios e primos.

Aos meus amigos da UFERSA Caraúbas, que guardo boas lembranças.

A todos amigos e colegas que acompanharam minha trajetória ou que participaram dela desde minha infância até hoje.

A todos os professores da UFERSA que eu tive o privilégio de ser aluno.

Bem, agradeço a todos que de alguma forma ajudaram ou comemoraram o meu êxito.

“História, nossas histórias. Dias de luta, dias de glória.”

(Charlie Brown Jr.)

RESUMO

Como resultado da utilização da técnica *Network Address Translation* (NAT) como forma de solução para o problema de escassez de endereços IP do protocolo IPv4, os computadores que se conectam à Internet por meio desta técnica perderam a capacidade de se comunicarem diretamente. Diante deste cenário, este trabalho propõe uma biblioteca JavaScript em Node.js que proporciona conectividade *peer-to-peer* (P2P) entre *peers* que se conectam à Internet por meio de NAT's não simétricos. A abordagem adotada para solucionar o problema da conectividade foi a utilização da técnica de UDP *Hole Punching* utilizando a rede BitTorrent DHT (*Distributed Hash Table*) para atuar como *rendezvous server*. Foram utilizadas simulações de redes para verificar o funcionamento da biblioteca e foi observado que ela obteve sucesso em proporcionar a conectividade desejada.

Palavras-chave: BitTorrent DHT. UDP Hole Punching. Ponto-a-Ponto. NodeJS.

ABSTRACT

As a result of using the, Network Address Translation (NAT) technique as a solution to the problem of IPv4 address limit, computers that connect to the Internet using this technique have lost the ability to communicate directly. In this scenario this work proposes a Node.js JavaScript library development that provides peer-to-peer (P2P) connectivity between peers with Internet connection through non-symmetrical NAT's. The approach adopted to solve the connectivity problem was the use of the UDP Hole Punching technique using the BitTorrent DHT (Distributed Hash Table) as a rendezvous server role. Network simulations were used to verify the operation of the library and it was observed that it succeeded in providing the desired connectivity.

Keywords: BitTorrent DHT. UDP Hole Punching. Peer-to-Peer. NodeJS.

LISTA DE ILUSTRAÇÕES

Figura 1 – Diferença entre arquitetura cliente/servidor e par-a-par	20
Figura 2 – Arquivo sendo distribuído utilizando BitTorrent	21
Figura 3 – Exemplo de mensagens de consulta (protocolo KRPC)	24
Figura 4 – Exemplo de mensagens de resposta, se encontrou algum <i>peer</i> (protocolo KRPC)	24
Figura 5 – Posicionamento e operação de um NAT	26
Figura 6 – O cabeçalho UDP	27
Figura 7 – UDP Hole Punching por trás de NAT's distintas	28
Figura 8 – Diagrama de atividades (visão geral)	36
Figura 9 – Estabelecimento de comunicação entre os <i>peers</i>	37
Figura 10 – Cabeçalho da biblioteca	39
Figura 11 – Exemplo de código JavaScript que utiliza a biblioteca	40
Figura 12 – Tela de chat do computador de Alice	41
Figura 13 – Tela de chat do computador de Bob	42
Figura 14 – Arquitetura de rede para testes com Mininet-VM	44
Figura 15 – Arquitetura de rede para testes com NetworkEmulatorCore	45
Figura 16 – Resultado da saída do mininet	47
Figura 17 – Alice, Bob e Charlie mostrando seus IP's privados	48
Figura 18 – Alice, Bob e Charlie tentando contato direto	49
Figura 19 – Alice, Bob e Charlie trocando mensagens via <i>Simple Chat</i>	49

LISTA DE TABELAS

Tabela 2 – Intervalos de endereços privados reservados	25
Tabela 3 – Resultado pesquisa de dispositivos que suportam UDP Hole Punching e TCP Hole Punching	29
Tabela 4 – Máquinas virtuais utilizadas nos testes	44

LISTA DE ABREVIATURAS E SIGLAS

AES	<i>Advanced Encryption Standard</i>
BEP	<i>BitTorrent Enhancement Proposal</i>
CD	<i>Compact Disk</i>
CLI	<i>Command Line Interface</i>
CORE	<i>Common Open Research Emulator</i>
DHT	<i>Distributed Hash Table</i>
I/O	<i>Input/Output</i>
ICE	<i>Interactive Connectivity Establishment</i>
ICMP	<i>Internet Control Message Protocol</i>
IoT	<i>Internet of Things</i>
IP	<i>Internet Protocol</i>
IPv4	<i>Internet Protocol versão 4</i>
IPv6	<i>Internet Protocol versão 6</i>
ISP	<i>Internet Service Provider</i>
IV	<i>Initialization vector</i>
MVP	<i>Minimum Viable Product</i>
NAT	<i>Network Address Translation</i>
NATs	<i>Network Address Translators</i>
NPM	<i>Node Package Manager</i>
P2P	<i>Peer-to-Peer</i>
STUN	<i>Simple Traversal of User Datagram Protocol (UDP) Through Network Address Translators (NATs)</i>
TLS	<i>Transport Layer Security</i>
TURN	<i>Traversal Using Relays around Network Address Translation (NAT)</i>
UDP	<i>User Datagram Protocol</i>

SUMÁRIO

1	INTRODUÇÃO	14
1.1	O PROBLEMA	15
1.2	SOLUÇÕES	15
1.2.1	Abordagem adotada	16
1.3	MOTIVAÇÃO	17
1.4	OBJETIVOS	17
1.4.1	Objetivo Geral	17
1.4.2	Objetivos Específicos	18
1.5	ESTRUTURA DO TRABALHO	18
2	FUNDAMENTAÇÃO TEÓRICA	19
2.1	BITTORRENT	19
2.1.1	Funcionamento do BitTorrent	20
2.1.2	BitTorrent DHT - Distributed Hash Table	22
2.1.2.1	Protocolo KRPC	23
2.2	NAT - NETWORK ADDRESS TRANSLATION	24
2.3	PROTOCOLO UDP	26
2.3.1	UDP Hole Punching	27
2.4	NODE.JS	30
2.4.1	NPM - Node Package Manager	31
3	TRABALHOS RELACIONADOS	32
3.1	PWNAT	32
3.2	TOMP2P	32
3.3	GITTORRENT	33
4	DESENVOLVIMENTO DA BIBLIOTECA	34
4.1	METODOLOGIA	34
4.2	FERRAMENTAS UTILIZADAS	35
4.3	DESCRIÇÃO DO PROJETO E ABORDAGEM	35
4.4	A API DESENVOLVIDA	39
4.4.1	Projeto exemplo	41
5	ANÁLISE E RESULTADOS	43
5.1	MATERIAIS E MÉTODOS	43

5.1.1	Experimentos no Mininet-VM	44
5.1.2	Experimentos no NetworkEmulatorCore	45
5.2	RESULTADOS E DISCUSSÃO	46
5.2.1	Resultados do experimento no Mininet-VM	46
5.2.2	Resultados do experimento no NetworkEmulatorCore	48
5.2.3	Discussão	50
6	CONSIDERAÇÕES FINAIS	51
6.1	LIMITAÇÕES	51
6.2	TRABALHOS FUTUROS	52
	REFERÊNCIAS	53

1 INTRODUÇÃO

A Internet se tornou extremamente popular desde a sua criação. Atualmente, são mais de 4 bilhões de indivíduos conectados à rede mundial de computadores segundo a wearesocial.com (2018). Se considerarmos que um usuário se conecta à Internet por meio de diversos dispositivos, podemos inferir que o número de dispositivos conectados à Internet é superior a quantidade de usuários a utilizando. Essa popularidade foi alavancada pelo surgimento e popularização dos *smartphones* que trouxeram o poder de um computador doméstico para dentro do bolso dos usuários, e também levaram a Internet a qualquer lugar.

No entanto, com esse crescimento e popularização vieram alguns problemas. Um deles é que o protocolo *Internet Protocol* versão 4 (IPv4) limita a quantidade de endereços da Internet em aproximadamente 4,29 bilhões de dispositivos (WETHERALL; TANENBAUM, 2011). Sendo que, para pertencer a Internet, é necessário possuir um endereço *Internet Protocol* (IP) único na rede, ou seja, como o número de endereços da Internet é limitado e para se conectar a ela é necessário um, com o crescimento da demanda, em algum instante temporal a Internet não suportaria novos usuários.

Foram pensadas algumas soluções para este problema e, dentre elas, a que se tornou popular e que é utilizada nos dias atuais é um conjunto de técnicas chamadas de NAT (SRI-SURESH, 2001). A técnica consiste basicamente de mapear um endereço público na rede em endereços locais (privados) e com isso proporcionar o compartilhamento de um endereço externo da Internet com várias máquinas em uma rede local. Um endereço público na rede é um endereço visível por todos os pertencentes da Internet, enquanto um endereço privado é acessível apenas em rede local. Para que um computador de uma rede local consiga se conectar à Internet, é necessário que esse IP local seja mapeado em um IP público na rede por um roteador de borda de área (WETHERALL; TANENBAUM, 2011).

No entanto, esta limitação de endereçamento é característica do IPv4. Estes endereços foram consideravelmente expandidos pelo protocolo *Internet Protocol* versão 6 (IPv6), de forma que, utilizar esse novo protocolo, praticamente, elimina o problema de endereçamento. Mas, somente o protocolo não elimina o problema da conexão direta, pois será necessário que todos *Internet Service Provider* (ISP) concordem em não utilizar o NAT no IPv6 (JOHANSSON, 2018). Bem como, os roteadores domésticos também deverão ser configurados adequadamente para permitir a alocação fixa de endereços IPv6's para os dispositivos conectados, como por exemplo, dispositivos móveis ou embarcados conectados em um roteador *wireless*.

1.1 O PROBLEMA

O NAT além de proporcionar esse compartilhamento de endereços da Internet também mantém essas redes internas isoladas da rede mundial de computadores. Isso acontece pelo funcionamento do NAT, pois como é feita tradução dos endereços privados em um endereço público, apenas ele (o tradutor NAT) sabe quem realmente é a origem de uma mensagem. Sendo assim, se um computador *A* pertencente a uma rede privada enviar uma mensagem através dos NATs para um computador *B* público na Internet, o computador *B* recebe como origem da mensagem o endereço mapeado pelo NAT e não o endereço privado do computador *A*.

Sendo assim, o NAT não permite (a menos que esteja especialmente configurado) que um computador pertencente a Internet se comunique diretamente com outro computador que pertença a uma rede privada. Essa característica, segundo Carpenter (1996), faz com que o NAT viole a comunicação ponta-a-ponta que a Internet foi projetada. Em outras palavras, com a utilização do NAT os computadores por trás de NAT's diferentes perdem a capacidade de se conectarem diretamente, ficando os pares isolados em redes privadas, e toda comunicação com a Internet é intermediada pelo NAT.

1.2 SOLUÇÕES

Foram desenvolvidas algumas soluções para este problema, denominadas de técnicas de travessia de NAT. Essas técnicas foram criadas para contornar o problema da falta de conectividade ponta-a-ponta perdida com a utilização do NAT.

Uma destas soluções é chamada de *UDP Hole Punching* comentada na RFC 5128 por Srisuresh, Ford e Kegel (2008). Essa técnica tenta contornar alguns tipos de NAT permitindo que dispositivos em redes privadas disjuntas possam se comunicar. A técnica basicamente consiste em dois pares descobrirem seus IP's e portas (que os NATs estão mapeando), trocarem essa informação entre si e, em seguida, ambos os pares tentarem enviar mensagens UDP um para o outro, usando as informações trocadas, criando uma perfuração UDP em ambos os NATs e permitindo a comunicação direta entre os computadores (FORD; SRISURESH; KEGEL, 2005).

No entanto, para que essa técnica funcione, ela precisa do auxílio de uma máquina externa (pública na Internet) para descoberta e compartilhamento dos IP's e portas que estão sendo mascarados pelo NAT. Esse servidor é denominado de *Rendezvous Server* e fará a ponte entre os dois *peers* atuando apenas no início da conexão (FORD; SRISURESH; KEGEL, 2005).

Outra técnica é chamada *Simple Traversal of UDP Through NATs* (STUN), especificada na RFC 5389 por Matthews *et al.* (2008). O STUN é um protocolo que permite que um computador (por trás de um NAT) consiga descobrir qual o tipo do NAT que faz a ponte entre ele e a Internet, qual o endereço IP e porta que o NAT está utilizando para mapear a conexão, fazer testes de conectividade e manter ativa conectividades abertas no NAT. O STUN sozinho não realiza travessia de NAT, no entanto pode ser utilizado como ferramenta em outras aplicações, como por exemplo o *Hole Punching*.

Mahy, Matthews e Rosenberg (2010) especificaram um outro protocolo chamado *Traversal Using Relays around NAT* (TURN) para estender o STUN e proporcionar conectividade independente do tipo do NAT envolvido. A técnica consiste basicamente em fazer a retransmissão de todas as mensagens trocadas entre dois *peers* que estão por trás de NATs. Essa retransmissão é feita em forma de serviço por um servidor público na Internet. Esta técnica não é a melhor opção, pois, apesar de funcionar em todos os tipos de NAT, depende de um servidor público na Internet em todo tempo de duração da conexão (MAHY; MATTHEWS; ROSENBERG, 2010).

Rosenberg (2010) especificaram outra técnica chamada de *Interactive Connectivity Establishment* (ICE). A técnica faz uso de um conjunto de protocolos, dentre eles o STUN e TURN citados anteriormente, para entregar uma solução robusta para o problema da travessia de NAT. O funcionamento do ICE consiste em inicialmente encontrar a melhor opção para comunicação direta entre dois agentes, em seguida compartilhar essa informação entre os agentes e testar se há conectividade entre eles, e por fim estabelecimento e manutenção de uma sessão entre os agentes.

1.2.1 Abordagem adotada

A abordagem adotada por este trabalho para superar a travessia de NAT é a utilização da técnica de UDP *Hole Punching* e a rede BitTorrent¹. Com esta união será possível obter conectividade *Peer-to-Peer* (P2P) utilizando a rede BitTorrent para ignição da conexão e também para descoberta de novos *peers* interessados em se comunicarem diretamente. Essa abordagem foi inspirada em Moore *et al.* (2015) e foi escolhida para proporcionar um UDP *Hole Punching* descentralizado, pois utiliza os inúmeros nós da rede BitTorrent para auxiliar na técnica.

Este trabalho irá fazer uso da rede e protocolo BitTorrent *Distributed Hash Table* (DHT),

¹ Neste trabalho o termo BitTorrent é utilizado para referenciar a arquitetura, o protocolo, a empresa e tecnologia BitTorrent. De forma alternada, dependendo do contexto.

a qual, segundo Loewenstern e Norberg (2008), basicamente permite armazenar arbitrariamente dados em nós da rede BitTorrent. A utilização desta rede irá suprir a dependência do UDP *Hole Punching* de um servidor público na rede para fazer a descoberta e troca das informações de conexão, pois o BitTorrent DHT tem milhares de nós públicos distribuídos na Internet que, segundo Moore *et al.* (2015), podem atuar como *Rendezvous Server*.

Para isso será desenvolvida uma biblioteca que reúne todas essas tecnologias, utilizando JavaScript e Node.js, e poderá ser incluída por desenvolvedores em seus projetos. A biblioteca também será leve o suficiente para ser usada em computadores de baixo custo e embarcados, como, por exemplo, uma Raspberry Pi².

1.3 MOTIVAÇÃO

A motivação para esta pesquisa se manifesta no benefício de proporcionar comunicação *peer-to-peer* entre dispositivos que pertencem a redes privadas disjuntas utilizando uma infraestrutura de servidores sem custo, anônima e descentralizada para auxiliar no processo de abertura do buraco UDP e troca de informações de conexão. Também, uma vez que a comunicação depende de um terceiro apenas para ser inicializada, podem-se ter mais garantias sobre ela como: disponibilidade, segurança e privacidade.

Outra motivação está em possíveis utilizações da biblioteca desenvolvida neste trabalho em projetos em *Internet of Things* (IoT). Um exemplo a destacar poderia ser a criação de um sistema de vigilância residencial onde a aplicação se comunica diretamente com a infraestrutura de câmeras de uma casa, sem necessidade de contratação de serviços de terceiros. Outro exemplo seria proporcionar criação de uma rede distribuída de monitoramento de animais ou plantas.

1.4 OBJETIVOS

1.4.1 Objetivo Geral

Desenvolver uma biblioteca JavaScript para realizar conexão direta (*peer-to-peer*) entre computadores de baixo custo que fazem parte de redes privadas distintas e se conectam à Internet por meio de NATs, e, para isso, utilizar a técnica do UDP *Hole Punching* junto com a rede e o protocolo BitTorrent DHT no suporte inicial à comunicação entre os *peers*.

² <https://www.raspberrypi.org/>

1.4.2 Objetivos Específicos

- (a) Revisar a literatura em busca de trabalhos similares a fim de identificar contribuições para o projeto;
- (b) Investigar o protocolo BitTorrent DHT e as implementações existentes em JavaScript;
- (c) Identificar as limitações dos computadores de baixo custo a fim de definir os requisitos mínimos para funcionamento do projeto;
- (d) Desenvolver o projeto em Node.js e acrescentá-lo ao *Node Package Manager* (NPM);
- (e) Realizar testes da ferramenta a fim de minimizar a existência de falhas e validar o seu funcionamento.

1.5 ESTRUTURA DO TRABALHO

O presente trabalho foi organizado da seguinte forma: no capítulo 2, será feita uma revisão sobre os referenciais teóricos que oferecem suporte para a realização do trabalho; o capítulo 3 será destinado a apresentar alguns trabalhos relacionados ou semelhantes a este; o capítulo 4 será destinado a apresentar o desenvolvimento da biblioteca proposta por este trabalho, descrevendo a metodologia, as ferramentas e a abordagem adotada; no capítulo 5, serão realizados testes na biblioteca desenvolvida e apresentados os resultados dos experimentos; e, o capítulo 6 fará uma abordagem geral do que foi feito neste trabalho, fazendo alusão aos resultados e se os mesmos estão em conformidade com o esperado, apontando limitações e sugestões de trabalhos futuros.

2 FUNDAMENTAÇÃO TEÓRICA

O presente trabalho se apoia em um conjunto previamente desenvolvido de tecnologias e conceitos, tais como alguns protocolos e padrões de rede. A compreensão de todo o suporte para o projeto é necessária para um melhor entendimento daquilo que se pretende alcançar. As próximas seções irão prover mais detalhes sobre os referenciais teóricos que oferecem suporte para a realização do trabalho.

2.1 BITTORRENT

O BitTorrent é um protocolo de compartilhamento de arquivos distribuídos em uma rede *peer-to-peer* (par-a-par) (COHEN, 2008). Com ele é possível distribuir arquivos entre milhares de *peers* de uma forma cooperativa, onde todos os *peers* compartilham e recebem pequenos pedaços do arquivo completo. Sendo que ao final do *download*, todas as peças são montadas formando o arquivo completo.

O BitTorrent se tornou tão popular que segundo pesquisa realizada por Barbera *et al.* (2005) o tráfego de dados em redes *peer-to-peer* pelos *backbones* no ano de 2005 era de aproximadamente 80% e uma grande parcela deste tráfego era de clientes utilizando o protocolo BitTorrent, se tornado o mais famoso protocolo de compartilhamento de arquivos P2P. E recentemente, a BitTorrent foi adquirida pela Tron¹ por valores estimados de aproximadamente 126 milhões de dólares (HEATER, 2018).

O protocolo BitTorrent pode ser utilizado por corporações para distribuir arquivos sem que seja necessário arcar com infraestrutura de servidor para hospedar estes arquivos. A possibilidade de distribuição de arquivos sem a necessidade de uma grande infraestrutura é bastante atraente para a redução de custos.

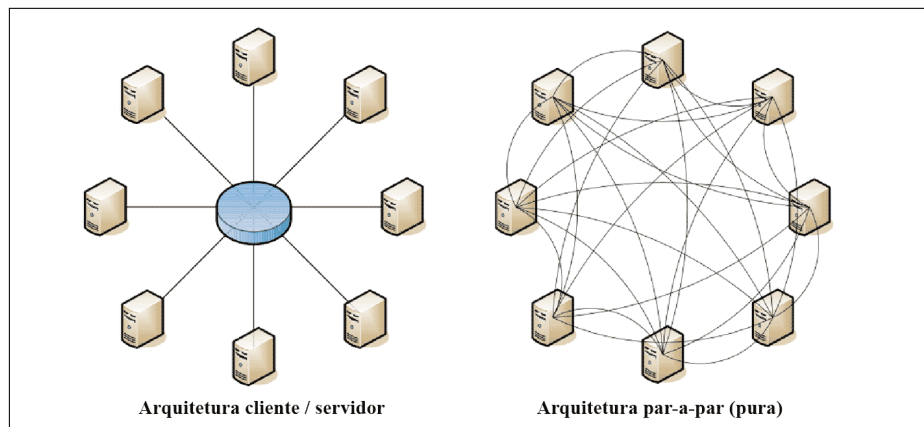
Quando se pretende disponibilizar um arquivo grande para um grande número de clientes, o BitTorrent permite que os interessados compartilhem automaticamente o arquivo entre eles sem que um servidor centralizado fique sobrecarregado. Por exemplo, o *Compact Disk* (CD) de instalação da distribuição Debian, disponível em debian.org (2018), é fornecido via *.torrent* (arquivo de metadados). Outro exemplo seria utilizar o protocolo para distribuir arquivos de atualização entre uma grande rede de servidores.

¹ <https://tron.network>

2.1.1 Funcionamento do BitTorrent

O paradigma de conexão entre pares, conhecido como P2P, surgiu para mudar a antiga arquitetura cliente-servidor existente na Internet, onde os clientes se conectam a servidores para que esses realizem alguma tarefa e enviem uma resposta. A Figura 1 exemplifica o contraste entre os dois paradigmas, na arquitetura cliente-servidor todos os *peers* se conectam a um servidor central para consumir um serviço, enquanto que na arquitetura par-a-par os *peers* se comunicam entre si de forma direta, ambos assumindo comportamento de cliente e servidor (KAMIENSKI *et al.*, 2005).

Figura 1 – Diferença entre arquitetura cliente/servidor e par-a-par



Fonte: Johnsen, Karlsen e Birkeland (2005)

O protocolo BitTorrent faz uso de ambas as arquiteturas, de forma híbrida. Ele utiliza a arquitetura cliente/servidor para uma comunicação inicial com um *tracker*, que é um servidor na web responsável por guardar informações de quais *peers* tem partes de um determinado arquivo e também qual a localização deste *peer* (endereço IP e porta). Os *peers*, que na rede Torrent podem ser *seed* (aquele que tem um arquivo completo e está semeando) ou *leecher* (aquele que está baixando o arquivo e/ou está semeando partes dele) se comunicam entre si diretamente em uma rede par-a-par (JOHNSEN; KARLSEN; BIRKELAND, 2005).

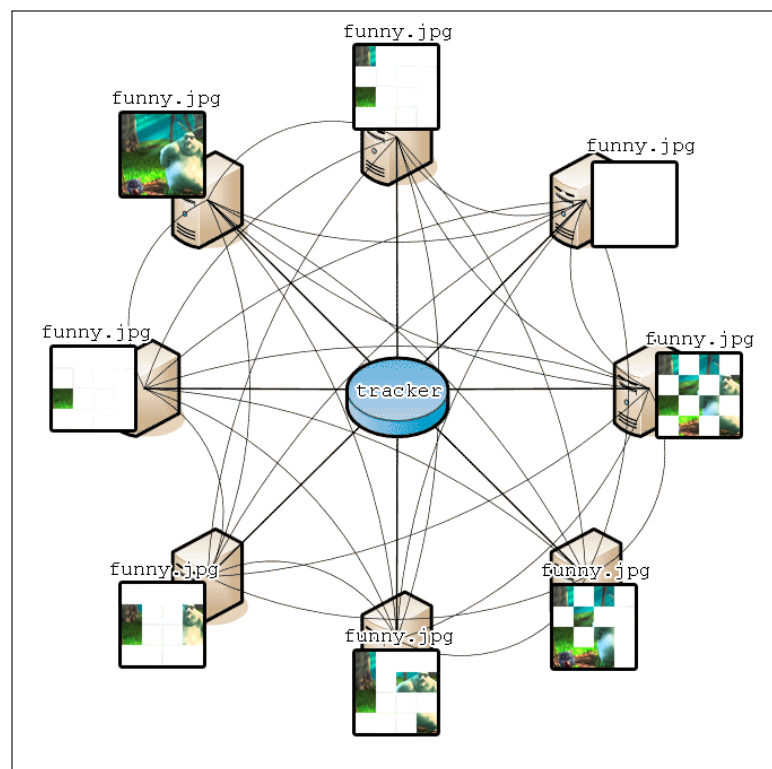
O funcionamento do protocolo consiste inicialmente em um *peer* que deseja semear um arquivo na rede BitTorrent escolher um *tracker* e então criar um arquivo *.torrent* a partir do conteúdo que deseja compartilhar. Para gerar esse arquivo, o BitTorrent divide o conteúdo que se deseja compartilhar em diversas pequenos pedaços, para cada pedaço é gerado um *hash* que a identifica. Também é gerado um *hash* que identifica o *.torrent* como um todo, esse é chamado de *info hash*. O *peer* então comunica ao *tracker* que ele deseja semear um arquivo, informando a

ele o *info hash* do *.torrent*. O *tracker* cria uma lista de *seeds* no seu banco de dados e inclui o *peer* nessa lista.

Em seguida, o *peer* precisa compartilhar esse arquivo *.torrent* para outros *peers* por meios convencionais (e-mail, sites, transferência de arquivos, etc.). Um *peer* que deseja obter o arquivo semeado, baixa o *.torrent* recebido, se comunica com o *tracker* e solicita a lista de *peers* que tem partes do arquivo. O *tracker* então fornece a lista e o *peer* que se comunica com todos na tentativa de baixar pedaços do arquivo. A partir do momento que o *peer* faz *download* de pedaços do arquivo, ele também pode semear partes do arquivo.

Neste ponto, outro *peer* que entrar na rede, agora poderá baixar partes do arquivo de dois lugares diferentes. À medida que cresce a quantidade de integrantes da rede, cresce também o número de fontes do arquivo. Dessa forma, em algum momento, o *peer* que introduziu o arquivo pode desconectar-se da rede e, ainda assim, o arquivo continuará disponível.

Figura 2 – Arquivo sendo distribuído utilizando BitTorrent



Fonte: Extraída de Johnsen, Karlsen e Birkeland (2005) e adaptada pelo autor

A Figura 2 ilustra esse processo, nela tem-se vários *peers* interagindo entre si e com um *tracker* para compartilhar pedaços de um arquivo chamado *funny.jpg* utilizando protocolo BitTorrent. A figura também mostra uma representação visual do estado da imagem em cada *peer*,

sendo possível observar que o compartilhamento dos pedaços não segue uma ordem predefinida, o BitTorrent equilibra o compartilhamento das partes de modo a maximizar a quantidade de *peers* compartilhando pedaços do arquivo.

Apesar de funcionar perfeitamente, um cenário problemático é se o *tracker* deixar a rede. Neste cenário, todos os *peers* perderiam a comunicação. Por esse motivo, ao criar um arquivo *.torrent*, é comum adicioná-lo a mais de um *tracker*, assim minimizando as chances de um arquivo parar de ser distribuído.

2.1.2 BitTorrent DHT - Distributed Hash Table

O protocolo BitTorrent teve algumas extensões, chamadas de BitTorrent *Enhancement Proposal* (BEP), elas acrescentam funcionalidades ao protocolo. Segundo Harrison (2008) existem doze extensões aceitas pela comunidade. Uma delas, proposta por Loewenstern e Norberg (2008), é a extensão BEP 5, chamada de BitTorrent DHT. Sendo a DHT a extensão mais presente nos clientes BitTorrent, pois permite que um *torrent* seja compartilhado sem que seja necessário a especificar um *tracker*. Essa capacidade torna a rede mais descentralizada, pois elimina o uso de um servidor central (LOEWENSTERN; NORBERG, 2008).

O DHT ou *TrackerLess* é uma rede onde cada *peers* age como uma tabela *hash*, podendo ser armazenados e recuperados dados de *torrents*. Desse modo, cada *peer* da rede pode atuar como um *tracker*. Os nós da rede DHT armazenam pares (chave, valor) e qualquer nó da rede pode consultar um dado associado a uma chave.

Cada nó da rede recebe um identificador (*node id*) de 160 bits, escolhido aleatoriamente. Para saber quais nós estão responsáveis por um determinado *hash*, é aplicada uma "métrica de proximidade" que compara o identificador do nó com o valor do *hash* afim de determinar qual nó está mais "próximo" do *hash*. Segundo (LOEWENSTERN; NORBERG, 2008), o DHT foi baseado no Kademlia², e a métrica de distancia utilizada é o XOR e o resultado é interpretado como um inteiro sem sinal resultante da função: $distancia(A, B) = |A \oplus B|$.

Cada nó também mantém uma tabela de roteamento internamente, esta que é utilizada na realização de consultas para saber qual nó está mais próximo de determinado *hash*. Quando um nó quer encontrar quais *peers* possuem um determinado *.torrent*, o nó procura em sua tabela de roteamento os *peers* com id mais próximo do *info hash* do *.torrent*. Em seguida, ele pergunta a estes *peers* se eles têm algum valor com o *info hash* como chave. Os *peers* que não tiverem

² <http://xlattice.sourceforge.net/components/protocol/kademlia/specs.html>

nenhuma informação irão responder com uma lista de *peers* com id mais próximos do *info hash* encontrados nas suas tabelas. O processo é recursivamente repetido até que seja encontrado os *peers* com ID's mais próximos de toda a rede (LOEWENSTERN; NORBERG, 2008).

2.1.2.1 Protocolo KRPC

A comunicação entre os nós da rede DHT do BitTorrent é feita utilizando um protocolo de chamadas de processamento remoto denominado KRPC, descrito por Loewenstern e Norberg (2008). Os nós da rede utilizam esse protocolo para trocar mensagens de consultas e respostas. As mensagens deste protocolo são enviadas utilizando protocolo UDP e codificadas no formato Bencode (codificação B), uma forma compacta de codificação binária de metadados de *torrents* para transmissão em rede (COHEN, 2008).

Segundo Loewenstern e Norberg (2008) o protocolo KRPC possui apenas quatro mensagens de consulta, são elas: *ping* - um simples *ping*; *find_node* - busca nós próximos a um *hash*; *get_peers* - busca nós associados a um *info hash* de um *.torrent*; *announce_peer* - se anuncia como associado a um *info hash* de um *.torrent*. Estas consultas podem retornar mensagens de resposta ou de erros. Sendo que os possíveis códigos de erros são: 201 - um erro genérico, não especificado; 202 - erro de servidor; 203 - pacote malformado, argumentos inválidos ou *token* inválido; 204 - comando não encontrado.

No entanto, Norberg e Siloti (2014) expandiram o BitTorrent DHT com o BEP 44 adicionando duas mensagens de consulta, são elas: *get* - recuperar um dado associado a uma chave de 160-bits; *put* - armazenar dados arbitrários associados a uma chave de 160-bits. Estas consultas foram adicionadas à BEP 5 para permitir que seja armazenados dados arbitrários associados a uma chave (NORBERG; SILOTI, 2014).

A Figura 3 mostra um exemplo de uma consulta de busca de *peers* que possuem associado determinado *info hash*. A chave "a" são os argumentos enviados na consulta *get_peers*, são dois argumentos, o "id" é o valor do *node id* e o "info_hash" se refere ao *info hash* do *.torrent* que se deseja encontrar os *peers* associados.

A Figura 4 mostra a resposta para a consulta exemplificada na Figura 3. Nesta resposta, a chave "r" representa dados de retorno da consulta, neles pode-se observar as chaves: "id" - identificador do nó que respondeu a consulta; "token" - um valor aleatório para ser utilizado em consultas futuras; "values" - uma lista com os *peers* encontrados que estão associados o *info hash*, os itens da lista vem codificados em binário, onde os 4 primeiros bytes representam o

Figura 3 – Exemplo de mensagens de consulta (protocolo KRPC)

```
consulta =
{
  "t": "aa",
  "y": "q",
  "q": "get_peers",
  "a": {
    "id": "abcdefghij0123456789",
    "info_hash": "mnopqrstuvwxyz123456"
  }
}

codificada =
d1:ad2:id20:abcdefghij01234567899:info_hash20:mnopqrstuvwxyz12345
6e1:q9:get_peers1:t2:aa1:y1:qe
```

Fonte: Adaptação extraída de Loewenstern e Norberg (2008)

endereço IP e os 2 últimos bytes representam a porta do *peer*.

Figura 4 – Exemplo de mensagens de resposta, se encontrou algum *peer* (protocolo KRPC)

```
resposta =
{
  "t": "aa",
  "y": "r",
  "r": {
    "id": "abcdefghij0123456789",
    "token": "aoeusnth",
    "values": [
      "axje.u",
      "idhtnm"
    ]
  }
}

codificada =
d1:rd2:id20:abcdefghij01234567895:token8:aoeusnth6:values16:axje.
u6:idhtnmee1:t2:aa1:y1:re
```

Fonte: Adaptação extraída de Loewenstern e Norberg (2008)

Loewenstern e Norberg (2008) descreve algumas chaves importantes que o protocolo KRPC traz. O "t" se refere a um valor de 2 *bytes* aleatórios atribuídos pelos clientes DHT no momento da consulta, e que funciona como um identificador da mensagem, é com ele que o cliente DHT consegue identificar qual é a consulta associada a uma resposta recebida. Cada mensagem também possui uma chave "y", que tem tamanho de 1 byte e representa o tipo da mensagem, podendo assumir os valores de: "q" - consulta; "r" - resposta; "e" - erro. Esta estrutura pode ser observada nas Figuras 3 e 4.

2.2 NAT - NETWORK ADDRESS TRANSLATION

O protocolo da camada de rede do modelo TCP/IP amplamente utilizado é o IPv4, o qual possui um cabeçalho com uma parte fixa de 20 *bytes*. Deste cabeçalho, tem-se 4 *bytes* utilizados para endereço de origem e 4 *bytes* para endereço de destino na rede. Esses endereços IPv4 são compostos por 32 bits (4 *bytes*) e são escritos normalmente em forma decimal com pontuação, sendo que, a cada *byte* (intervalo de 0-255 em decimal), é acrescentado um ponto. Um exemplo seria o endereço em hexadecimal C8890651, que na forma decimal seria 200.137.6.81 (WETHERALL; TANENBAUM, 2011).

Os endereços IPv4 possuem valores limitados a aproximadamente 4,29 bilhões de *hosts*. No início da Internet, quando era utilizada principalmente em universidades, eram atribuídos um a cada cliente. No entanto, os projetistas não esperavam tamanha popularização da Internet, e com sua chegada veio também o problema da escassez de endereços IP. Algumas soluções tiveram de ser pensadas para resolver esta situação.

Uma saída para este problema era atribuir endereços IP's de forma dinâmica para clientes, sendo assim, quando um cliente se conectava à Internet ele recebia um endereço IP aleatório. Quando ele desconectava, esse endereço ficava livre para que outro usuário pudesse se conectar à Internet e recebê-lo (WETHERALL; TANENBAUM, 2011).

Outra solução, globalmente utilizada, foi a *IP Network Address Translator* (NAT) descrita na RFC 2663 por Srisuresh e Holdrege (1999). Com o NAT, os computadores de uma rede local (interna de uma empresa) passaram a utilizar endereços de IP's privados, que não são publicamente visíveis na Internet, ou seja, se um serviço na Internet enviar uma mensagem para um endereço IP privado, esta é descartada pela rede. Os endereços privados foram definidos na RFC 1918 por Rekhter *et al.* (1996), como mostrado na Tabela 2.

Tabela 2 – Intervalos de endereços privados reservados

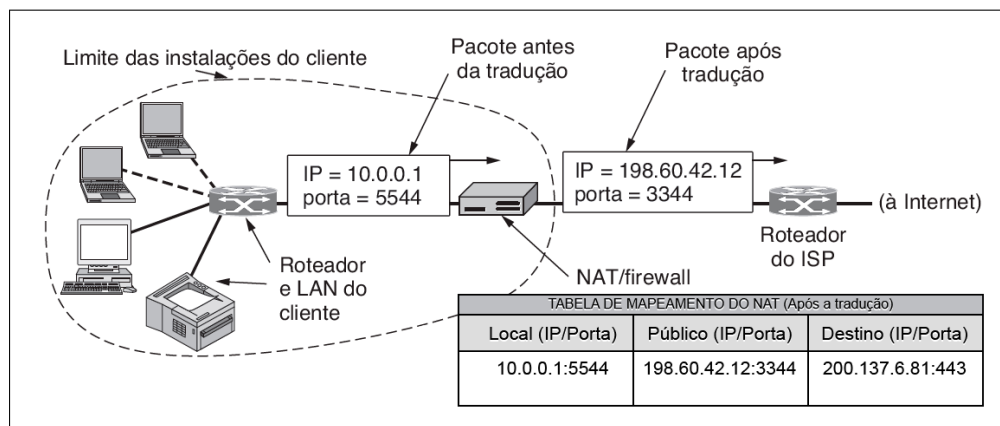
Intervalo		Quantidade de hosts
Limite inferior	Limite superior	
10.0.0.0	10.255.255.255	16.777.216 hosts
172.16.0.0	172.31.255.255	1.048.576 hosts
192.168.0.0	192.168.255.255	65.536 hosts

Fonte: Adaptação extraída de Rekhter *et al.* (1996)

O funcionamento do NAT consiste basicamente em fazer as traduções dos endereços privados, que não são enxergados pela Internet, para endereços públicos visíveis na Internet. Para

exemplificar esse processo, um cliente 10.0.0.1 (privado) tenta enviar uma mensagem pela sua porta 5544 para o servidor da Internet 200.137.6.81 na porta 443. Como seu endereço é privado, este é traduzido pelo NAT para o endereço 198.60.42.12 (visível pela Internet). A porta também é traduzida para uma porta 3344 aberta pelo NAT no endereço traduzido e é por essa porta que as mensagens de resposta serão encaminhadas. Quando uma resposta chegar, o processo do NAT acontece de forma inversa e a mensagem é entregue a 10.0.0.1, conforme ilustrado na Figura 5.

Figura 5 – Posicionamento e operação de um NAT



Fonte: Extraída de Wetherall e Tanenbaum (2011) alterado pelo autor

Ainda no recebimento de uma mensagem, para que o NAT consiga saber para qual endereço privado deve entregar uma mensagem que foi recebida de um endereço da Internet, ele mantém uma tabela chamada "Tabela de mapeamento do NAT". É nesta tabela que são registrados temporariamente os endereços IP's e portas da origem e do destino das mensagens como também o endereço IP e porta, públicos na Internet, utilizados na tradução, como ilustrado na Figura 5. O NAT atualiza esta tabela de acordo com o fluxo de mensagens e remove da tabela entradas não utilizadas por um determinado período de tempo.

Existem, segundo Rosenberg *et al.* (2003), quatro variações do NAT, são elas: *Full Cone NAT*, *Restricted Cone NAT*, *Port Restricted Cone NAT*, *Symmetric NAT*. Estas variações diferenciam-se uma das outras principalmente na escolha dos endereços e portas para tradução. Nos NAT's do tipo Cone NAT o mesmo endereço privado e porta são traduzidos para o mesmo endereço público, mesmo se o endereço de destino mudar. Isso não acontece nos NAT's simétricos. Nesses, se o endereço de destino mudar, uma nova porta pública é atribuída para a sessão.

A grande vantagem de se usar um NAT é permitir que computadores de uma rede privada possam acessar a Internet através de um único ponto de saída. Essa configuração é bastante

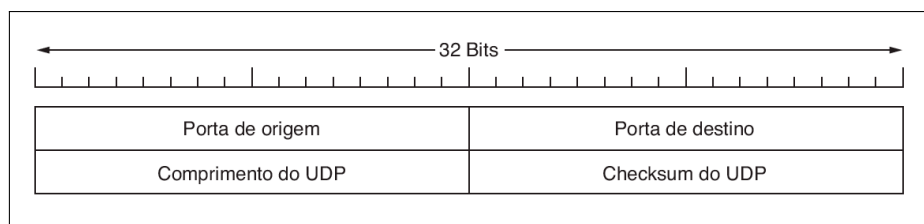
comum hoje em dia com roteadores *wireless*, por exemplo. Porém, existem diversas desvantagens, uma delas é que os computadores por trás de uma rede NAT perderam a capacidade de serem visíveis diretamente pela Internet, pois seus endereços privados só os identificam unicamente na rede local (privada).

2.3 PROTOCOLO UDP

O UDP é um protocolo da camada de transporte do modelo TCP/IP. Este protocolo fornece meios para que aplicações se comuniquem sem que seja necessário estabelecer uma conexão entre elas. Entretanto, o UDP, por sua simplicidade, não realiza controle de fluxo, controle de congestionamento, retransmissão de segmento que foi perdido pela rede ou ordenação de segmentos que chegarem desordenados. Ou seja, o UDP não faz garantias, não verifica entrega ou sequenciamento dos segmentos. Fica a cargo da aplicação a responsabilidade de implementar estas garantias, caso alguma seja um requisito para funcionamento da mesma (WETHERALL; TANENBAUM, 2011).

O UDP fornece apenas o necessário para que a camada de transporte consiga fazer a comunicação entre processos comunicantes em máquinas distintas. Seu funcionamento consiste em transmitir segmentos que são formados pela carga útil (fluxo de *bytes* fornecido pela aplicação) e um cabeçalho de 8 *bytes* de tamanho.

Figura 6 – O cabeçalho UDP



Fonte: Wetherall e Tanenbaum (2011)

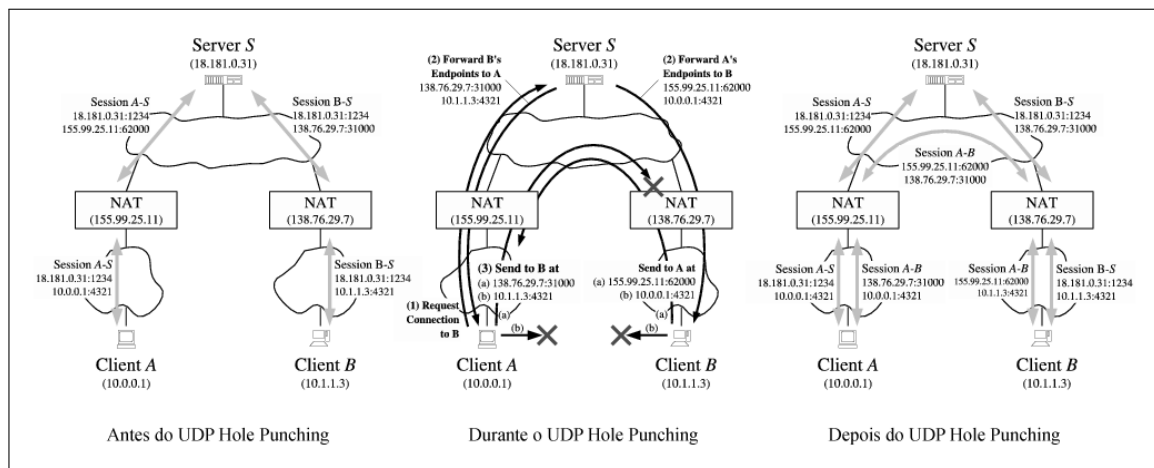
Neste cabeçalho, como ilustrado na Figura 6, os campos "Comprimento UDP" e "Checksum do UDP" servem, respectivamente, para informar o tamanho do segmento e para verificação de integridade dos dados. Já os campos "Porta de origem" e "Porta de destino" servem para identificar os processos de origem e destino do segmento. Sendo assim, quando um segmento UDP chega à camada de transporte do destino, o cabeçalho é interpretado e a carga útil é entregue ao processo que estiver associado à porta de destino (WETHERALL; TANENBAUM, 2011).

2.3.1 UDP Hole Punching

O *UDP Hole Punching* é uma técnica que possibilita comunicação UDP direta entre dois *hosts* que estão localizados por trás de NAT's. Para que a técnica funcione os dois *hosts* precisam da ajuda de um terceiro que esteja localizado na Internet, fora de um NAT. Ou seja, antes que os *peers*, que estão por trás da NAT, iniciem a troca de mensagens UDP entre si, é necessário que se comuniquem com um terceiro *host* que irá informar aos roteadores entre os *peers* que estes irão trocar informações. Este terceiro *host* desempenha o papel de informar os roteadores entre os *peers* seus endereços IP e portas para que a comunicação possa ocorrer diretamente. Assim, a comunicação pode ocorrer de um *peer* para o outro (uma vez iniciada) sem que pacote algum passe pelo terceiro *host*. A técnica é bastante utilizada por diversas empresas, inclusive, segundo Baset e Schulzrinne (2004), o Skype (aplicação de chamadas VoIP - Voz sobre IP) utiliza uma adaptação para realização de chamadas de áudio/vídeo entre seus usuários.

De forma geral, o funcionamento da técnica é explicado por Ford, Srisuresh e Kegel (2005) e pode ser observado na Figura 7, em que temos o Cliente A e o Cliente B, que estão atrás de diferentes NAT's, querem trocar mensagens UDP. Inicialmente ambos os clientes se comunicam com o Servidor S, nesse momento o NAT do Cliente A irá criar uma sessão A-S na sua tabela de mapeamento, o mesmo acontece com o NAT do Cliente B, com a sessão B-S. Após essas sessões iniciadas, um dos clientes pede ajuda ao Servidor S para se comunicar com o outro cliente, e, nesse momento, o Servidor S envia para o Cliente B as informações de endereço IP/Porta do Cliente A, e faz o mesmo com o Cliente A, enviando as informações do Cliente B.

Figura 7 – UDP Hole Punching por trás de NAT's distintas



Fonte: Ford, Srisuresh e Kegel (2005)

De posse das informações enviadas pelo Servidor S, o Cliente A inicia um processo de enviar pacotes UDP's para todos os IP's do Cliente B. Esses pacotes serão descartados pelo NAT que o Cliente B está por trás, no entanto, este processo cria uma sessão entre o Cliente A e B no NAT do Cliente A.

Em algum instante temporal, o Cliente B também recebe as informações do Cliente A vindas do Servidor S e também inicia um processo de enviar pacotes UDP's para todos os IP'S do Cliente A. No entanto, estes pacotes irão chegar até o Cliente A, pois há uma sessão entre Cliente A-B na tabela de mapeamento do NAT do Cliente A, criada pelos disparos UDP feitos por ele. Ao receber a mensagem vinda do Cliente B, os Clientes A e B já conseguem se comunicar diretamente, pois no momento que o Cliente B fez o envio da mensagem UDP foi criada uma sessão na tabela de mapeamento do NAT do Cliente B. A partir deste ponto ambos podem trocar mensagens UDP livremente, isso é ilustrado na Figura 7 pelo estabelecimento da Sessão A-B, após a técnica.

A técnica não irá funcionar se um dos *hosts* estiver por trás de um NAT simétrico, isso acontece pela característica deste tipo de NAT de trocar as informações da tabela de mapeamento quando o *peer* tentar enviar um pacote UDP para um par endereço IP/Porta diferentes dos que estão registrados na tabela. Sendo assim, quando um dos *hosts* tentar enviar uma mensagem UDP o NAT irá atualizar as informações da tabela de roteamento trocando a porta, e essa nova porta não é conhecida pelo *host* de destino, impossibilitando que seja aberto um buraco UDP.

Contudo, Ford, Srisuresh e Kegel (2005) realizaram uma pesquisa que alcançou como resultado o percentual de funcionamento da técnica *UDP Hole Punching*. A técnica foi aplicada em 380 equipamentos de diferentes marcas, o resultado pode ser conferido na Tabela 3. Foi concluído que, felizmente para a técnica do *UDP Hole Punching*, os NAT's simétricos são minoria e a taxa geral de funcionamento do *UDP Hole Punching* é de 82%. Se considerado apenas os dispositivos da Belkin, Cisco, SMC e 3Com a taxa de funcionamento da técnica alcança os 100% (FORD; SRISURESH; KEGEL, 2005).

Existe também uma técnica similar chamada *TCP Hole Punching*, esta, no entanto, não consegue uma taxa de sucesso tão atraente, funcionando de forma geral em 64% dos equipamentos pesquisados por Ford, Srisuresh e Kegel (2005), conforme Tabela 3. Diante disso, foi escolhido para este projeto utilizar a técnica *UDP Hole Punching* como técnica principal, devido sua maior taxa de sucesso.

Tabela 3 – Resultado pesquisa de dispositivos que suportam UDP Hole Punching e TCP Hole Punching

	UDP Hole Punching		TCP Hole Punching	
NAT em Hardware				
Linksys	45/46	(98%)	33/38	(87%)
Netgear	31/37	(84%)	19/30	(63%)
D-Link	16/21	(76%)	9/19	(47%)
Draytek	2/17	(12%)	2/7	(29%)
Belkin	14/14	(100%)	11/11	(100%)
Cisco	12/12	(100%)	6/7	(86%)
SMC	12/12	(100%)	8/9	(89%)
ZyXEL	7/9	(78%)	0/7	(0%)
3Com	7/7	(100%)	5/6	(83%)
NAT baseado em Sistema Operacional				
Windows	31/33	(94%)	16/31	(52%)
Linux	26/32	(81%)	16/24	(67%)
FreeBSD	7/9	(78%)	2/3	(67%)
Todos os fornecedores	310/380	(82%)	184/286	(64%)

Fonte: Adaptado de Ford, Srisuresh e Kegel (2005)

2.4 NODE.JS

O Node.js ou NodeJS é um interpretador de JavaScript de código aberto e multiplataforma que proporciona a execução de *scripts* fora dos *browsers*. Esse interpretador foi desenvolvido baseado no motor de JavaScript utilizado pelo Google Chrome (navegador de Internet da empresa norte-americana Google), o Google's V8 (NODEJS.ORG, 2018).

Segundo Flanagan (2007), o JavaScript (ou ECMAScript, que é como foi padronizada pela *European Computer Manufacturer's Association*) é uma linguagem de programação que surgiu no início da Internet, desenvolvida para o antigo Netscape, mas que hoje pode ser considerada uma linguagem de propósito geral.

O surgimento do Node.js veio do desejo de solucionar um problema comum entre algumas linguagens de programação, como: Java, .NET, Ruby ou PHP. Estas linguagens bloqueiam o processo em execução quando o servidor precisa realizar uma operação de *Input/Output* (I/O), também conhecida como operação bloqueante (PEREIRA, 2014).

Nesse sentido, vamos considerar que a requisição de um usuário a um serviço web representa a iniciação de um novo processo no servidor para tratar esta requisição. Com o passar do tempo, a medida que o número de usuários crescem, vão sendo criados novos processos no servidor. Em um sistema bloqueante, essas requisições são enfileiradas e vão sendo processadas uma-a-uma, sem que múltiplos processamentos ocorram. Quando uma requisição é processada,

as demais ficam bloqueadas aguardando em uma fila de requisições ociosas (PEREIRA, 2014).

Com o Node.js isso não acontece, a aplicação não bloqueia quando precisa fazer I/O, ela continua tratando as requisições de outros clientes. Isso acontece porque toda sua arquitetura é *non-blocking thread* (não bloqueante). Além dessa característica, o Node.js também é *single-thread* (uma thread por processo) e *event-driven* (orientado a eventos) (PEREIRA, 2014).

Tudo isso em conjunto com a rapidez e leveza proporcionado pelo Google's V8, que entrega desempenho sem consumir muitos recursos. Isso permite inclusive que o Node.js seja executado em computadores com baixo poder computacional.

Estas características, aliadas ao fato que, segundo Stack Overflow (2018), o JavaScript é a linguagem de programação mais popular no ano de 2018, motivaram a escolha do Node.js para o trabalho. Um destaque extra para o benefício da tecnologia não ser bloqueante, pois para este trabalho que o canal é a Internet, é fundamental não ocorrerem bloqueios na aplicação, pois isso iria atrasar a resposta a um cliente.

2.4.1 NPM - Node Package Manager

O NPM, uma abreviação de *Node Package Manager*, é o maior registro de software do mundo. Possui mais de 600 mil pacotes em seu registro e um volume de 3 bilhões de downloads por semana (NPMJS.COM, 2018). Desenvolvedores individuais e organizações do mundo todo publicam seus projetos JavaScript no NPM e o usam para gerenciar e compartilhar suas aplicações em desenvolvimento. O NPM já vem acompanhado com o Node.js, trabalhando em conjunto para melhorar a experiência do desenvolvedor.

O NPM é formado por 3 componentes principais: o site, local onde é possível buscar os pacotes, gerenciar suas aplicações criar perfis ou organizações; o registro de pacotes, é uma enorme base de dados de softwares JavaScript; uma ferramenta *Command Line Interface* (CLI) que é executada via terminal do sistema operacional, sendo esta a forma como os desenvolvedores interagem com o registro de pacotes (NPMJS.COM, 2018). Um dos grandes benefícios do NPM é a capacidade de incluir pacotes ou softwares de terceiros dentro de projetos e utilizá-los. Assim promovendo um reuso de código JavaScript em escala global.

3 TRABALHOS RELACIONADOS

Este capítulo tem o objetivo de mostrar trabalhos que também se objetivam em proporcionar conectividade entre *peers* que se conectam à Internet por meio de um serviço de tradução de endereços. Os trabalhos escolhidos para compor este capítulo foram selecionados baseado na relação dos temas dos artigos com o UDP *Hole Punching* e BitTorrent DHT.

3.1 PWNAT

O PWNAT¹ é uma ferramenta que permite usuários finais estabelecerem comunicação direta entre eles. Esta ferramenta é uma implementação da técnica de travessia de NAT de forma anônima (*Autonomous NAT Traversal*) descrita por Muller *et al.* (2010). A técnica se baseia em como os NAT's fazem a entrega de pacotes *Internet Control Message Protocol* (ICMP) para *hosts* que estão em suas redes privadas de modo a fazer com que um *host* que está por trás de um NAT consiga "fingir" ser público na Internet.

Apesar de eficiente, esta ferramenta depende que a passagem de pacotes ICMP seja permitida e tratada pelo NAT. A ferramenta age em baixo nível, não sendo fácil implementar uma rede onde vários *peers* possam trocar mensagens. A ferramenta proposta neste trabalho utiliza o protocolo UDP, evitando bloqueios dos NAT's por motivo do protocolo, também permite que facilmente vários *peers* se comuniquem diretamente.

3.2 TOMP2P

O TomP2P² é uma implementação de uma rede P2P de *Distributed Hash Table* (DHT) que proporciona uma forma descentralizada de armazenamento de dados em uma estrutura chave-valor. Essa aplicação também trás algumas funcionalidades extras como, por exemplo, a possibilidade de adição de mais de um valor para uma chave. A biblioteca também permite que *peers* se inscrevam para receberem alterações do valor de uma determinada chave. Para estender as funcionalidades do TomP2P, Wagner e Stiller (2015) propõem a utilização da técnica de UDP *Hole Punching* no intuito de proporcionar comunicação direta entre os nós da rede.

Apesar da biblioteca TomP2P proporcionar conectividade P2P e de seu funcionamento

¹ <https://samy.pl/pwnat/>

² <https://github.com/tomp2p/Tomp2P>

ser semelhante do BitTorrent DHT, para utilizar a biblioteca é necessário ter um IP público para servir de ponto de encontro para a rede. Diferentemente da biblioteca proposta por este trabalho, que utiliza a rede do BitTorrent DHT para servir de ponto de encontro para o UDP *Hole Punching*. Dispensando a necessidade do cliente possuir um IP público.

3.3 GITTORRENT

O GitTorrent³ é uma ferramenta de linha de comando que permite armazenar repositórios Git de forma descentralizada. Para isto, a biblioteca faz uso da rede DHT do BitTorrent para armazenar os repositórios de forma distribuída entre os nós da rede. Sendo assim, tornando desnecessário a utilização de um serviço central para sincronização dos repositórios Git em um projeto.

Apesar da biblioteca GitTorrent proporcionar conectividade P2P, este não é o foco principal dela. Tornando inviável para um desenvolvedor utilizar a biblioteca GitTorrent para fins diferentes do foco principal. Diferente disto, a ferramenta proposta por esse trabalho permite que facilmente seja realizada troca de mensagens de propósito genérico entre *peers*.

³ <https://github.com/cjb/GitTorrent>

4 DESENVOLVIMENTO DA BIBLIOTECA

Este trabalho tem como objetivo desenvolver uma biblioteca JavaScript que possibilite comunicação direta entre dispositivos separados por redes privadas distintas. Para alcançar este objetivo foram seguidos alguns métodos e utilizadas distintas ferramentas que auxiliaram no desenvolvimento. Este capítulo objetiva explicar esta metodologia e as ferramentas utilizadas, também apresentar e descrever a biblioteca desenvolvida e mostrar exemplos de uso.

4.1 METODOLOGIA

Para alcançar os objetivos deste trabalho, inicialmente, foi feita uma revisão bibliográfica com intuito de levantar trabalhos relacionados ao tema e também as tecnologias utilizadas neles. Estes trabalhos serviram de fundamentação teórica que deram o suporte adequado para desenvolvimento do projeto.

Em paralelo com esta revisão da literatura, também foi feito um estudo aprofundado sobre o funcionamento do protocolo BitTorrent DHT e do funcionamento do UDP *Hole Punching*. Estas tecnologias juntas formam o núcleo base para o desenvolvimento do projeto, então estas precisavam ser bem compreendidas. Este estudo foi feito, em duas etapas, primeiro com base na interpretação da documentação de cada tecnologia disponível na literatura. Em seguida, foram feitas análises de tráfego de rede, a fim de acompanhar como o protocolo BitTorrent DHT estava sendo implementado pelos nós da rede BitTorrent.

Para a elaboração da biblioteca, inicialmente, foram levantados os requisitos necessários, e, em seguida, foi elaborada uma prova de conceito do UDP *Hole Punching* a fim de validar se a técnica poderia ser aplicada utilizando a linguagem JavaScript e NodeJS. Feito isso, foi desenvolvido um *Minimum Viable Product* (MVP) que faz uso da prova de conceito do UDP *Hole Punching* e do protocolo BitTorrent DHT. Este MVP foi recebendo incrementos e melhorias até contemplar os objetivos propostos.

Para acompanhar a evolução do desenvolvimento, foi utilizado sistema de versionamento Git¹. E para assegurar o funcionamento da mesma, foram criados testes de unidade durante todo o desenvolvimento. Também foram criados testes funcionais automatizados em ambiente de rede simulado.

¹ <https://git-scm.com/>

4.2 FERRAMENTAS UTILIZADAS

A biblioteca foi criada utilizando o ambiente Node.js, instalado e devidamente configurado junto com o NPM. Sendo o NPM fundamental para gestão das dependências e também para publicação da biblioteca tornando-a disponível publicamente para ser importada e utilizada por terceiros em seus projetos.

Para o desenvolvimento, foi utilizada a linguagem JavaScript e o editor de texto de código aberto Atom². No Atom, foram adicionados alguns *plugins* auxiliares, como o *plugin* de auto-completar código JavaScript e também de validação em tempo real se o código está dentro de padrões definidos pela comunidade da linguagem (ESLint³). Foram também realizados testes de unidade no decorrer de todo desenvolvimento utilizando a biblioteca de testes Tape⁴.

Para estudo do protocolo BitTorrent DHT em funcionamento, foi utilizada a ferramenta Wireshark⁵. O Wireshark age como uma escuta em uma interface de rede, possibilitando acompanhar o tráfego nesta entrada. Além disso, a ferramenta permite filtrar e decodificar o resultado da escuta. Isto foi de grande auxílio no estudo do BitTorrent, pois facilitou encontrar e compreender os pacotes deste protocolo.

4.3 DESCRIÇÃO DO PROJETO E ABORDAGEM

Este trabalho tem como base um conjunto de técnicas predefinidas na literatura, e estas serviram de subsídio na abordagem escolhida: a construção de uma biblioteca JavaScript que faz uso da rede BitTorrent DHT para servir como *Rendezvous Server* na técnica de *UDP Hole Punching*. De modo a usufruir dos milhares de nós da rede para ajudar no estabelecimento de comunicação entre *peers*.

A biblioteca simula a participação na rede BitTorrent DHT e se comporta de maneira similar a qualquer *peer* normal pertencente a rede e que deseja baixar algum arquivo *.torrent*. No entanto, a biblioteca não se comporta igual a um membro da rede. Como visto na Subseção 2.1.2, o protocolo pode atender a várias consultas, porém a biblioteca responde apenas a consultas do tipo "ping". Essa estratégia foi adotada para evitar que recursos de hardware dos dispositivos que utilizam a biblioteca sejam usados pela rede BitTorrent DHT. Ou seja, se a biblioteca permitisse

² <https://atom.io/>

³ <https://www.npmjs.com/package/eslint>

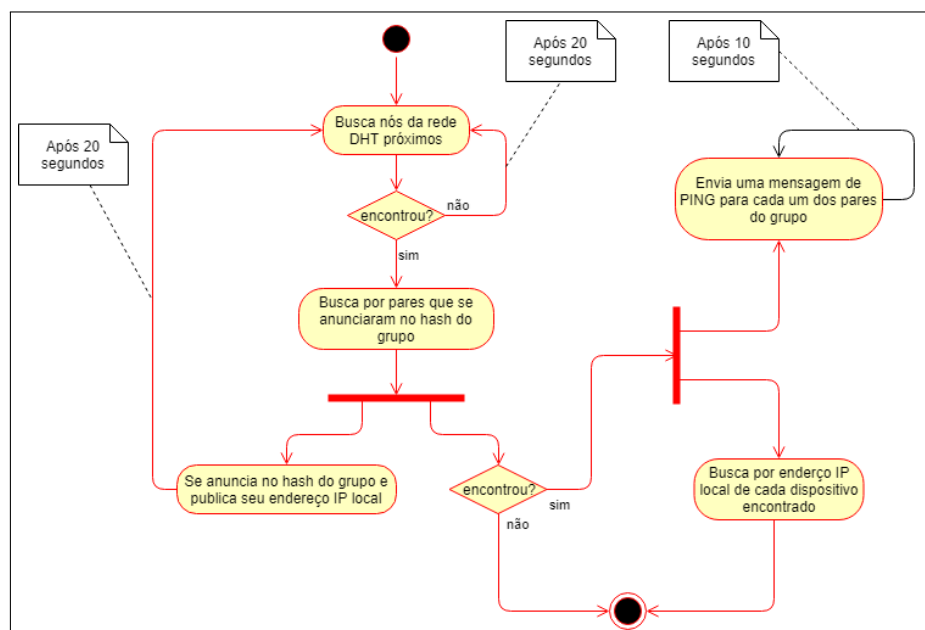
⁴ <https://www.npmjs.com/package/tape>

⁵ <https://www.wireshark.org/>

que consultas do tipo "put" fossem respondidas, isso poderia comprometer o funcionamento de equipamentos de baixo custo que possuem pouca memória.

O funcionamento geral da biblioteca pode ser observado no diagrama de atividades ilustrado na Figura 8. Nele, se pode ver que, inicialmente, a biblioteca faz uma busca pelos nós mais próximos a um *hash*. Esse *hash* é formado por combinações de um "NomeDeRede" e uma "senha", estas informações são fornecidas pelo usuário da aplicação. Esse *hash* é chamado de *hash* do grupo e é o endereço principal da rede.

Figura 8 – Diagrama de atividades (visão geral)



Fonte: O Autor, 2018

Após a busca pelos nós mais próximos do *hash* do grupo, a biblioteca terá a sua disposição uma lista com vários nós da rede BitTorrent DHT. Estes nós inicialmente serão consultados individualmente para verificar se há algum *peer* que já tenha se anunciado e possua o *hash* do grupo. Em paralelo a essa consulta, a biblioteca se anuncia para os nós encontrados. Adicionando assim seu endereço IP e porta na lista de *peers* dos nós da rede BitTorrent DHT.

A biblioteca também faz a adição do seu endereço IP local, por meio de uma consulta de "put" nos nós. Ou seja, outros *peers* agora terão acesso à informação do IP e porta público (mapeado pelo NAT) e também do IP e porta local da rede privada. Esse par de informações é importante, pois permite que computadores que estejam em uma mesma rede possam se comunicar diretamente, sem que o tráfego saia da rede local. Apesar deste não ser o propósito central do trabalho, esta funcionalidade foi adicionada, pois amplia a qualidade da conexão em

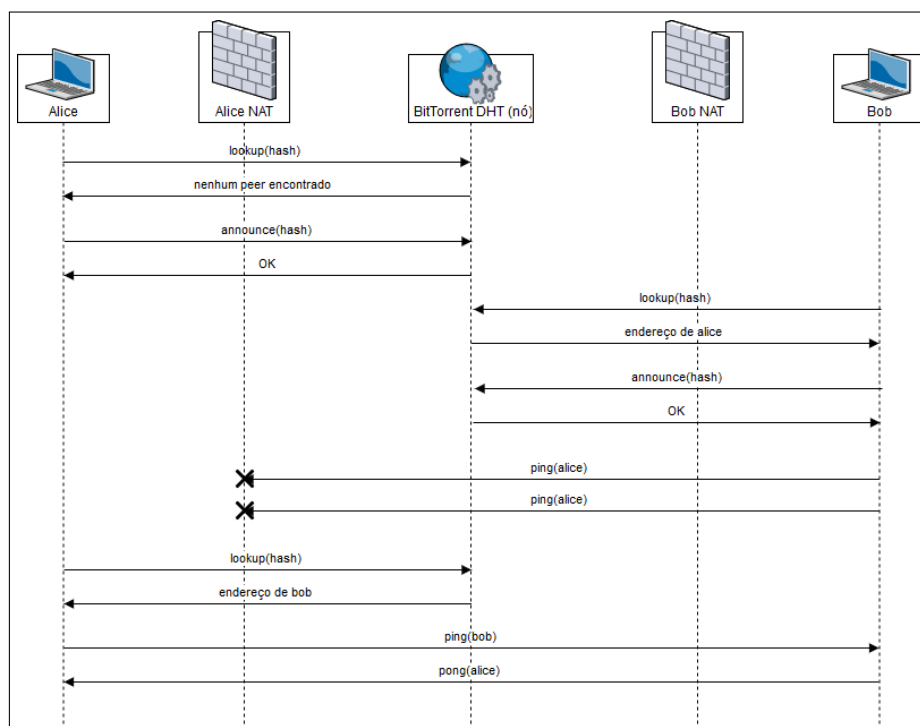
cenários específicos.

Ao final da etapa de anúncio, a biblioteca aguarda 20 segundos e, em seguida, repete o processo de busca de novos nós e pares. Este *loop* é importante para que a rede consiga encontrar novos membros à medida que estes vão se anunciando em um hash do grupo. Quando um novo *peer* é descoberto, inicia-se um processo de *ping* periódico nesse novo *peer*.

Além do *loop* de descoberta de *peers* e *nodes*, também existe outro loop que se encarrega de verificar se um determinado *peer* ainda está vivo. Esse processo consiste em enviar mensagens regulares a cada 10 segundos para os *peers* encontrados na rede. Cada nó irá receber a mensagem e a responder, avisando que ainda está vivo na rede. Tal processo tem duas finalidades: saber quando um *peer* saiu da rede e, dado que NAT's retiram entradas de suas tabelas caso não haja comunicação por um determinado período, manter o buraco UDP aberto.

Os passos para estabelecimento de uma comunicação entre *peers* é ilustrado pela figura Figura 9. Nela Alice e Bob, dois clientes que acessam a internet por meio de NATs pretendem estabelecer comunicação direta utilizando o BitTorrent DHT como nó de encontro para o UDP *Hole Punching*. Inicialmente Alice faz uma busca por nós associados a o *hash* do grupo por meio da função *lookup*, em seguida, após receber a resposta do Nó DHT, se anuncia como associado ao *hash* do grupo com a função *announce*.

Figura 9 – Estabelecimento de comunicação entre os *peers*



Fonte: O Autor, 2018

Em seguida, em algum instante temporal, Bob faz o mesmo processo que Alice, só que agora Bob recebe como resposta do *lookup* o endereço IP e porta de Alice, pois ela já tinha se anunciado. Com as informações de Alice, Bob inicia um processo de *ping*, enviando mensagens diretas para Alice. Estas mensagens são interceptadas pelo NAT de Alice, no entanto, o NAT de Bob adiciona em sua tabela de mapeamento o endereço de Alice.

Após 20 segundos (tempo de espera da biblioteca), Alice novamente repete o processo de *lookup* e *announce*. Desta vez, Alice recebe as informações de endereço IP e Porta de Bob, pois Bob se anunciou para o *hash* do grupo no nó DHT. Com essas informações em mãos, Alice envia mensagem de *ping* para Bob, e como Bob já tem em seu NAT o endereço de Alice mapeado, então o NAT de Bob entrega corretamente a mensagem. Em seguida, Bob responde o *ping* de Alice com uma mensagem de *pong* e desta vez o NAT Alice já tem o endereço de Bob mapeado, então entrega a mensagem de resposta a Alice.

Ao final dos passos ilustrados pela Figura 9 já é possível enviar e receber mensagens de forma direta por meio do UDP *Hole Punching*. No entanto, pela mesma porta que será feita a perfuração UDP também irão trafegar os dados de resposta dos protocolos KRPC (ver Subseção 2.1.2).

Para diferenciar os pacotes que utilizam o protocolo KRPC foi criado um cabeçalho da biblioteca simples que é acrescentado às mensagens que são trocadas entre os *peers*. Este cabeçalho, conforme ilustrado na Figura 10, possui apenas dois campos de 2 *bytes* e um campo de 16 *bytes* (20 *bytes* no total).

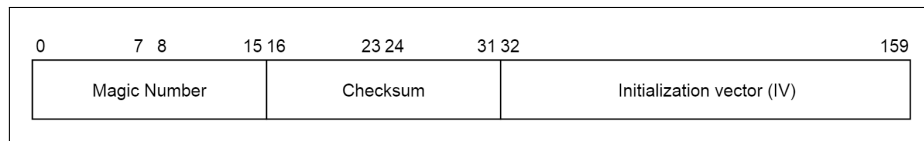
O primeiro campo é um inteiro denominado de *Magic Number*, este é um valor fixo em todas as mensagens, ele serve apenas como identificação. Ele foi inspirado em arquivos e protocolos que utilizam um conjunto de bytes fixos para identificação, como é o caso dos arquivos de imagem JPEG que sempre iniciam com os bytes 0xFFD8. Outro exemplo é o protocolo SMB que inicia com o inteiro 0xFF534D42.

O segundo campo é o campo de *Checksum*, este campo é inspirado no campo *checksum* do protocolo UDP e serve para identificar se uma mensagem foi corrompida, danificada ou modificada proposital ou acidentalmente durante o percurso.

O terceiro campo é o campo *Initialization vector* (IV), inspirado no campo homônimo do protocolo *Transport Layer Security* (TLS), este campo é utilizado para criptografia simétrica do conteúdo da mensagem utilizando algoritmo *Advanced Encryption Standard* (AES). Esta criptografia já vem implementada nativamente pelo NodeJS na biblioteca *crypto*⁶ e foi utilizada neste

⁶ <https://nodejs.org/api/crypto.html>

Figura 10 – Cabeçalho da biblioteca



Fonte: O Autor, 2018

projeto para acrescentar uma camada de segurança. A chave utilizada para encriptar e decriptar os dados é gerada internamente pela biblioteca baseada nas informações de "NomeDeRede" e "senha".

A biblioteca apesar de possuir uma base teórica e técnica avançada, não exige do utilizador nenhum conhecimento prévio sobre os protocolos ou tecnologias utilizadas, bastando apenas utilizar a API pública disponibilizada e criar aplicações que interajam com os eventos disparados pela mesma. Essa transparência para o usuário é importante porque torna a biblioteca bastante tolerante a falhas, pode-se destacar um exemplo, se um nó da rede DHT deixar de funcionar, a aplicação irá se reajustar e um novo nó assumirá o lugar. O usuário também não necessita de conhecimento sobre a localização física dos *peers*, a biblioteca gera nomes para cada *peer*, alcançando uma transparência de localização.

4.4 A API DESENVOLVIDA

A biblioteca foi publicada no repositório⁷ de códigos NPM, e para utilizá-la em um novo projeto Node.js basta criar uma pasta vazia e, em seguida, configurar o projeto com o comando: `npm init`. O comando irá criar um novo arquivo chamado *package.json*, este arquivo é responsável por guardar os metadados do projeto NodeJS. Em seguida, com o projeto criado, é preciso instalar a biblioteca no mesmo, utilizando o comando:

```
npm install @josaiasmoura/peer-network --save
```

Este comando irá automaticamente fazer a instalação da biblioteca e também irá configurar o *package.json* (criado anteriormente) adicionando a biblioteca como dependência no novo projeto.

Após a instalação da biblioteca ela poderá ser importada dentro dos códigos do projeto. A Figura 11 apresenta um exemplo de código JavaScript que utiliza a biblioteca desenvolvida neste projeto para criar uma simples aplicação que cria um grupo secreto de rede chamado

⁷ <https://www.npmjs.com/package/@josaiasmoura/peer-network>

"NomeDaRede" e com senha "senhadarede". Em seguida, após execução do método *start*, a aplicação inicia a escuta dos eventos de *ready* (linha 5 da Figura 11) que corresponde a quando a rede estiver pronta. O evento *message* (linha 7 da Figura 11) é disparado quando o *peer* recebe uma mensagem de um outro *peer* da rede. Também é possível enviar mensagem para outro *peer*, basta executar o método *send*, passando como argumento a mensagem e o nome do *peer* de destino.

Figura 11 – Exemplo de código JavaScript que utiliza a biblioteca

```

1  const PeerNetwork = require("@josaiasmoura/peer-network");
2
3  let peer = new PeerNetwork({ group:"NomeDaRede", password:"
    senhadarede" });
4
5  peer.on("ready", () => {
6      console.log("Estou online!");
7  }).on("message", (msg, from) => {
8      console.log("Voce recebeu mensagem! ", from, msg.
        toString());
9  }).on("online", (newPeer) => {
10     console.log("Novo peer online! ", newPeer);
11     peer.send(Buffer.from("Ola novo peer"), newPeer);
12 }).on("offline", (offPeer) => {
13     console.log("Peer esta offline agora! ", offPeer);
14 }).on("warning", (err) => {
15     console.log("Alerta! " + err.message);
16 });
17
18 peer.start();

```

Fonte: O Autor, 2018

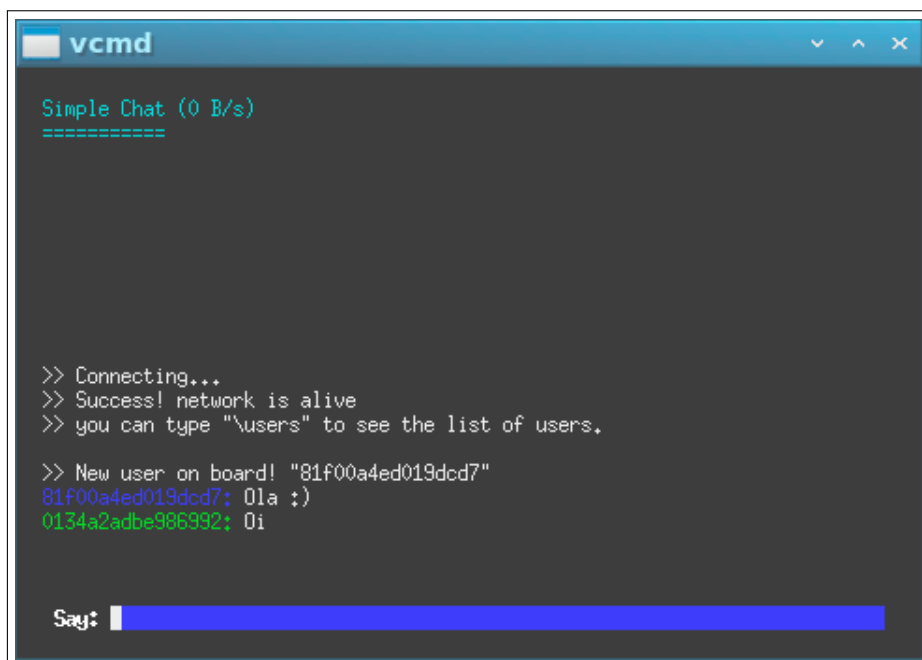
Outros eventos também estão sendo escutados pelo código ilustrado pela Figura 11. Dentre eles, tem-se o evento *online* (linha 9 da Figura 11), que é disparado quando um novo *peer* é encontrado na rede e a comunicação com ele é estabelecida com sucesso. O evento *offline* (linha 12 da Figura 11) é o oposto ao evento *online*, ele é disparado quando um dos *peers* que estava *online* deixa de responder à rede por mais de 20 segundos. Já o evento *warning* (linha 14 da Figura 11) é disparado quando ocorrer algum comportamento inesperado ou não suportado pela biblioteca. Um exemplo de comportamento não suportado é quando a biblioteca acessar a internet por meio de um NAT simétrico.

4.4.1 Projeto exemplo

Para melhor exemplificar a utilização da biblioteca, foi criado um projeto denominado *Simple Chat*. Este projeto consiste em um chat em terminal onde todos os *peers* pertencem a uma mesma sala e conseguem trocar mensagens entre si. A aplicação escuta todos os eventos descritos na subseção anterior, a fim de identificar quando um novo *peer* está *online*, quando recebeu mensagem, quando enviou mensagem, quando um *peer* ficou *offline*.

Na aplicação, quando um *peer* estiver *online* será mostrada uma mensagem *New user on board*, seguido de um número identificador do *peer*. A partir deste momento este novo *peer* participará da conversa. Quando um *peer* envia uma mensagem, esta mensagem aparece na sua tela com a cor verde, já quando ele recebe uma mensagem, esta aparece na tela com uma cor azul. As mensagens são compostas pelo identificador do *peer* mais dois pontos e, em seguida, o texto da mensagem.

Figura 12 – Tela de chat do computador de Alice



```
vcmd
Simple Chat (0 B/s)
=====

>> Connecting...
>> Success! network is alive
>> you can type "\users" to see the list of users.

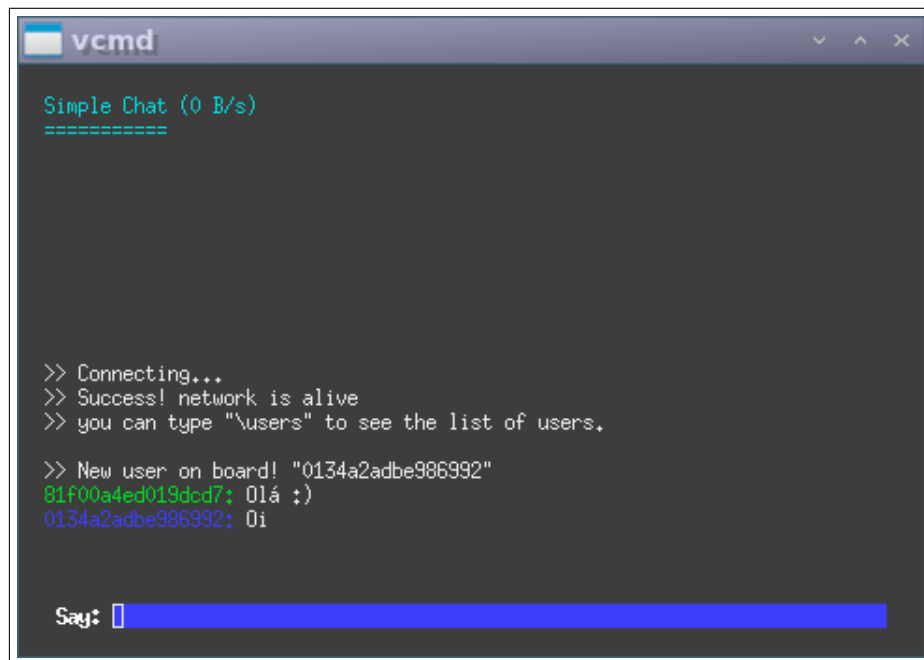
>> New user on board! "81f00a4ed019dcd7"
81f00a4ed019dcd7: Ola :)
0134a2adbe986992: Oi

Say: 
```

Fonte: O Autor, 2018

Alice e Bob são nomes fictícios para representar duas pessoas que estão em redes distintas e acessam à Internet por meio de NAT. Na Figura 12, temos Alice que se conectou a rede e enviou mensagem para Bob. Na Figura 13 vemos que Bob recebeu a mensagem e respondeu para Alice. Ambos puderam conversar livremente de forma direta através do NAT.

Figura 13 – Tela de chat do computador de Bob



Fonte: O Autor, 2018

5 ANÁLISE E RESULTADOS

5.1 MATERIAIS E MÉTODOS

A solução desenvolvida neste projeto foi submetida a alguns testes com finalidade de verificar o seu funcionamento em alguns cenários específicos, ou seja, verificar se a biblioteca consegue permitir conectividade direta entre *peers* que acessam a Internet por meio de NAT's. Para reproduzir os cenários, foram utilizadas duas redes simuladas, no intuito de reconstruir cenários inspirados nos apresentados por Ford, Srisuresh e Kegel (2005).

Os testes para validação da biblioteca foram feitos utilizando duas ferramentas de simulação de redes, o Mininet¹ e o *Common Open Research Emulator* (CORE)². Ambas executadas a partir de máquinas virtuais criadas utilizando a ferramenta VirtualBox³. A virtualização foi necessária pois o desenvolvimento da biblioteca foi feito em ambiente Windows, no entanto, as ferramentas de simulação são baseadas em Linux. Essa alternância entre sistemas operacionais foi possível devido a natureza multiplataforma do Node.js.

A ferramenta Mininet foi escolhida pela sua facilidade em criar *scripts* automatizados de testes, facilitando que outros pesquisadores possam reproduzir os testes e verificar os resultados obtidos. Enquanto que a ferramenta CORE foi escolhida para compor o ambiente de testes devido sua facilidade em simular ambientes de rede mais complexos. Também por sua fácil utilização, podendo inclusive, simular um terminal de um cliente de forma gráfica e intuitiva.

Os testes foram realizados em duas máquinas virtuais, descritas na Tabela 4). As configurações de *hardware* delas foram definidas de acordo com as necessidades de cada sistema, de modo que os recursos nunca fossem usados por completo. No caso da primeira máquina, a "Mininet-VM"⁴, a máquina virtual não tem servidor gráfico instalado, utiliza um sistema operacional Ubuntu 64-bit, para ela foi alocado 1024 MB de memória e 1 núcleo de processamento. Enquanto a "NetworkEmulatorCore"⁵ possui interface gráfica, utiliza um sistema operacional Ubuntu 32-bit, foi alocado para ele 2048 MB de memória e 2 núcleos de processamento.

Estes testes funcionais foram desenvolvidos a partir da primeira versão estável da bibli-

¹ <http://mininet.org/>

² <https://www.nrl.navy.mil/itd/ncs/products/core>

³ <https://www.virtualbox.org/>

⁴ <https://github.com/mininet/mininet/releases/download/2.2.2/mininet-2.2.2-170321-ubuntu-14.04.4-server-amd64.zip>

⁵ <https://downloads.pf.itd.nrl.navy.mil/core/vmware-image/vcore-4.7.zip>

Tabela 4 – Máquinas virtuais utilizadas nos testes

Nome	Sistema Operacional	Memória alocada	Cores alocados
Mininet-VM	Ubuntu (64-bit)	1024MB	1 core
NetworkEmulatorCore	Ubuntu (32-bit)	2048MB	2 cores

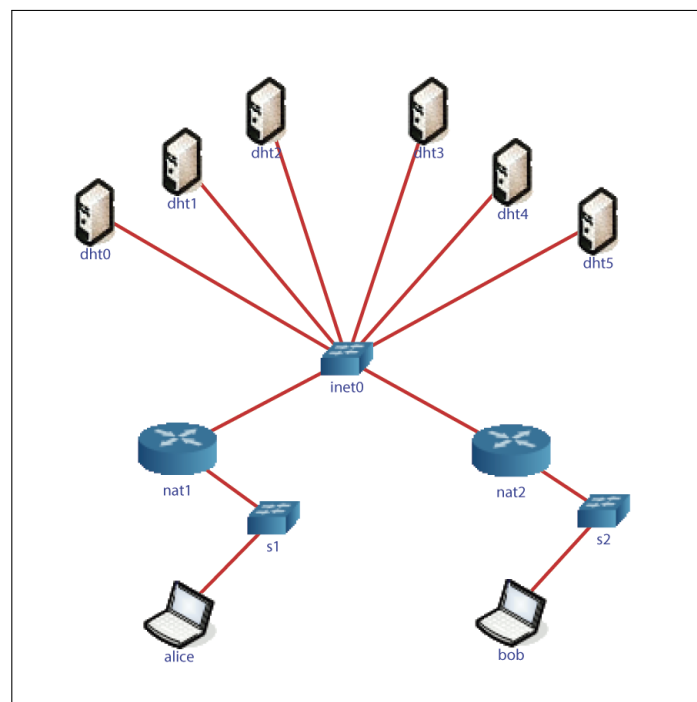
Fonte: O Autor, 2018

oteca publicada no registro de códigos NPM. Na publicação também estão contidos todos os scripts necessários para reproduzir os testes aqui descritos.

5.1.1 Experimentos no Mininet-VM

Na máquina virtual Mininet-VM foram realizados testes funcionais por meio da execução de script automatizado que utiliza a biblioteca produzida para realizar comunicação entre dois atores: Alice e Bob. A topologia

Figura 14 – Arquitetura de rede para testes com Mininet-VM



Fonte: O Autor, 2018

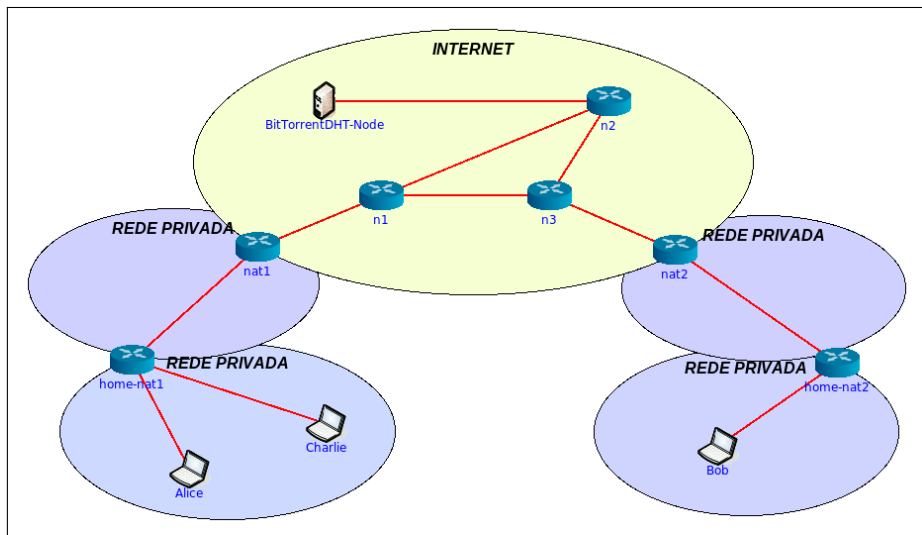
Como ambiente destes atores, foi utilizada uma arquitetura de rede composta por 2 NAT's, 3 switch's e 6 nós da rede BitTorrent DHT. Essa arquitetura está ilustrada na Figura 14, sendo que o primeiro nó da rede, "dht0", será o nó de *bootstrap* e os dois últimos nós são os computadores de Alice e Bob, o switch "inet0" será uma simulação de uma internet entre os

NATs e os nós DHT.

5.1.2 Experimentos no NetworkEmulatorCore

Na máquina virtual NetworkEmulatorCore, foi replicado um teste similar ao efetuado na máquina Mininet-VM utilizando a aplicação de exemplo (*Simple Chat*). No entanto, foi acrescentado alguns roteadores a mais e também duas camadas de NAT's, ou seja, neste teste expandimos o cenário em busca de validar a biblioteca em uma situação mais complexa. Também foi acrescentado um novo ator Charlie que pertence a mesma rede de Alice.

Figura 15 – Arquitetura de rede para testes com NetworkEmulatorCore



Fonte: O Autor, 2018

A nova arquitetura agora possui 4 NAT's, 3 *peers*, 3 roteadores e 1 nó da rede BitTorrent DHT. Essa arquitetura pode ser vista na Figura 15, onde pode-se ver os NATs de borda (nat1 e nat2) e os NATs domésticos (home-nat1 e home-nat2). Além dos *peers* Alice e Bob, separados por NATs, neste cenário há também o *peer* Charlie, que faz parte da mesma rede local de Alice. O intuito deste é validar se a biblioteca consegue identificar um *peer* que pertence a mesma rede privada. Foi escolhido adicionar apenas um nó DHT neste teste porque no teste anterior já foi contemplado o caso de ter vários nós da rede DHT disponíveis.

5.2 RESULTADOS E DISCUSSÃO

Nesta seção apresenta-se os resultados dos experimentos (testes) realizados na biblioteca. Os testes são divididos em subseções, sendo que a primeira se destina aos testes realizados em ambiente virtual Mininet, a segunda se destina aos realizados em ambiente virtual CORE e, por fim, uma subseção de discussão sobre os experimentos.

5.2.1 Resultados do experimento no Mininet-VM

A Figura 16 representa a saída do terminal ao executar o comando: `npm run mininet`. Este comando irá executar as rotinas de teste em linha de comando presentes no arquivo `start.py` que se encontra dentro da pasta `"/tests/mininet"`. Este script se encarrega de criar toda a topologia descrita na subseção 5.1.1.

Analizando a saída do terminal, percebe-se que inicialmente é preparada toda a estrutura de rede virtualizada, os NAT's são configurados e em seguida é iniciada uma tarefa de "*Ping all*" que tem intuito de detectar se os nós da rede virtualizada conseguem se encontrar normalmente.

O resultado da tarefa de *ping* mostrou que: nenhum dos nós DHT conseguem encontrar Alice ou Bob na rede; Alice ou Bob conseguem encontrar qualquer nó na rede, menos um ao outro; O NAT que faz a tradução do endereço de Alice consegue encontrar todos os participantes da rede menos o Bob; O NAT que faz a tradução de endereço de Bob consegue encontrar todos os *peers* da rede, menos Alice.

O que conclui-se a partir desta tarefa de "*Ping all*" é que na rede simulada Alice e Bob não conseguem se comunicar normalmente, no entanto, Alice consegue se comunicar com o NAT que Bob faz sua tradução de endereços. O mesmo acontece com Bob, que não consegue se comunicar normalmente com Alice, mas consegue se comunicar com o NAT que ela faz tradução de endereços.

Em seguida, o teste prossegue, agora tentando, por meio da biblioteca desenvolvida neste trabalho, estabelecer comunicação com Alice por meio do UDP *Hole Punching* e com a ajuda dos nós BitTorrent DHT. Esta comunicação é alcançada e demonstrada quando Bob consegue enviar uma mensagem de "*hello*" para Alice e ela então recebe e responde a mensagem. E com isso o experimento chega ao seu final.

Figura 16 – Resultado da saída do mininet

```

root@mininet-vm:/tcc# npm run mininet

> @josaiasmoura/peer-network@0.0.2 mininet /tcc
> mn -c 2> /dev/null && cd test/mininet && python start.py

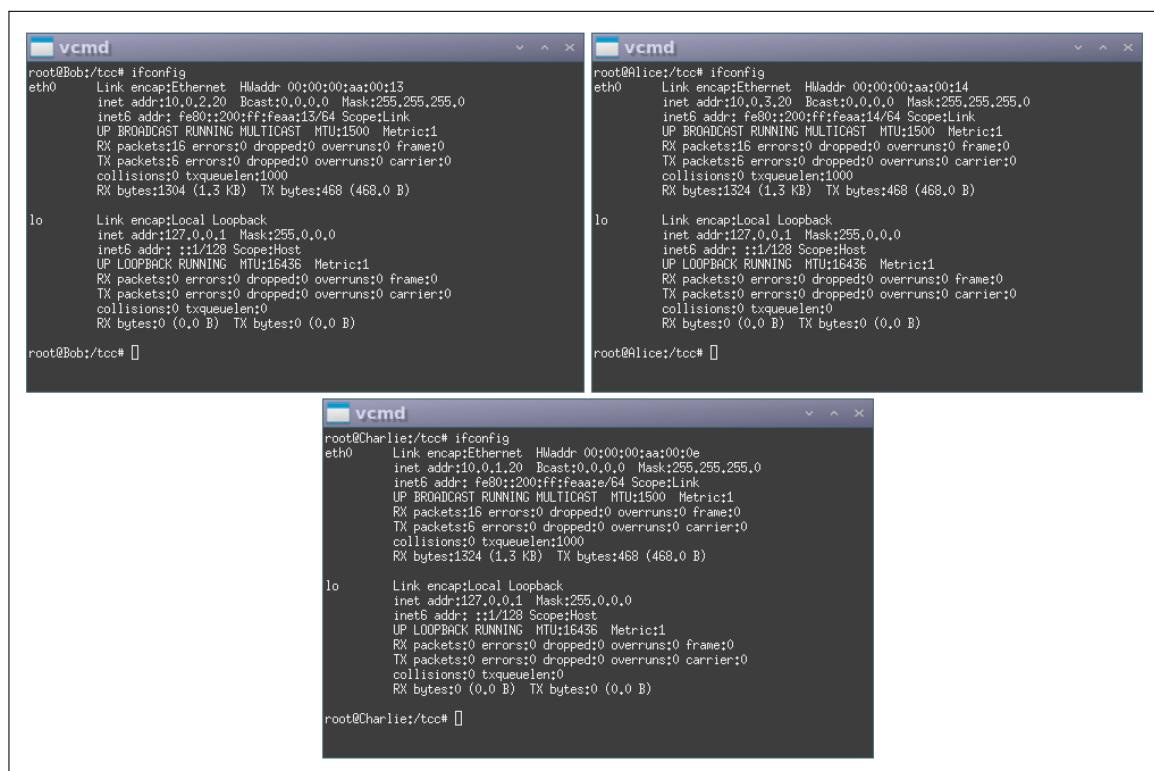
*** Creating network
*** Adding controller
*** Adding hosts:
alice bob dht0 dht1 dht2 dht3 dht4 dht5 nat1 nat2
*** Adding switches:
inet0 s1 s2
*** Adding links:
(alice, s1) (bob, s2) (dht0, inet0) (dht1, inet0) (dht2, inet0) (dht3, inet0)
(dht4, inet0) (dht5, inet0) (nat1, inet0) (nat1, s1) (nat2, inet0) (nat2, s2)
*** Configuring hosts
alice bob dht0 dht1 dht2 dht3 dht4 dht5 nat1 nat2
*** Starting controller
c0
*** Starting 3 switches
inet0 s1 s2 ...
*** Config NAT
*** Ping all
*** Ping: testing ping reachability
alice -> X dht0 dht1 dht2 dht3 dht4 dht5 nat1 nat2
bob -> X dht0 dht1 dht2 dht3 dht4 dht5 nat1 nat2
dht0 -> X X dht1 dht2 dht3 dht4 dht5 nat1 nat2
dht1 -> X X dht0 dht2 dht3 dht4 dht5 nat1 nat2
dht2 -> X X dht0 dht1 dht3 dht4 dht5 nat1 nat2
dht3 -> X X dht0 dht1 dht2 dht4 dht5 nat1 nat2
dht4 -> X X dht0 dht1 dht2 dht3 dht5 nat1 nat2
dht5 -> X X dht0 dht1 dht2 dht3 dht4 nat1 nat2
nat1 -> alice X dht0 dht1 dht2 dht3 dht4 dht5 nat2
nat2 -> X bob dht0 dht1 dht2 dht3 dht4 dht5 nat1
*** Results: 17% dropped (74/90 received)
***** PoC using Mininet
*** Start DhtNode
*** Bob and Alice start communicate with DhtNode
Bob is alive on port 21201
Bob now can talk with Alice
Bob send hello message to Alice
Bob receive hello from Alice
Bob close socket
***** End of PoC
*** Stopping 1 controllers
c0
*** Stopping 12 links
.....
*** Stopping 3 switches
inet0 s1 s2
*** Stopping 10 hosts
alice bob dht0 dht1 dht2 dht3 dht4 dht5 nat1 nat2
*** Done

```


5.2.2 Resultados do experimento no NetworkEmulatorCore

No experimento do CORE, o teste não foi realizado de forma automática como no Mininet. Foi utilizado a aplicação de exemplo a *Sample Chat* para tentar conectividade entre 3 *peers*. Sendo que 2 deles já possuíam conectividade antes. Assim possibilitando demonstrar que a biblioteca também funciona caso os *peers* pertençam a mesma rede interna ou tenham conectividade direta.

Figura 17 – Alice, Bob e Charlie mostrando seus IP's privados



The figure displays three terminal windows, each showing the output of the 'ifconfig' command for a different peer (Alice, Bob, and Charlie) in a NetworkEmulatorCore environment. Each window shows the configuration for the 'eth0' (Ethernet) and 'lo' (Loopback) interfaces.

Bob's configuration (top left):

```

root@Bob:/tcc# ifconfig
eth0      Link encap:Ethernet  HWaddr 00:00:00:aa:00:13
          inet addr:10.0.2.20  Bcast:0.0.0.0  Mask:255.255.255.0
          inet6 addr: fe80::200:ff:feaa:13/64 Scope:Link
          UP BROADCAST RUNNING MULTICAST  MTU:1500  Metric:1
          RX packets:16 errors:0 dropped:0 overruns:0 frame:0
          TX packets:6 errors:0 dropped:0 overruns:0 carrier:0
          collisions:0 txqueuelen:1000
          RX bytes:1304 (1.3 KB)  TX bytes:468 (468.0 B)

lo        Link encap:Local Loopback
          inet addr:127.0.0.1  Mask:255.0.0.0
          inet6 addr: ::1/128 Scope:Host
          UP LOOPBACK RUNNING  MTU:16436  Metric:1
          RX packets:0 errors:0 dropped:0 overruns:0 frame:0
          TX packets:0 errors:0 dropped:0 overruns:0 carrier:0
          collisions:0 txqueuelen:0
          RX bytes:0 (0.0 B)  TX bytes:0 (0.0 B)

root@Bob:/tcc#

```

Alice's configuration (top right):

```

root@Alice:/tcc# ifconfig
eth0      Link encap:Ethernet  HWaddr 00:00:00:aa:00:14
          inet addr:10.0.3.20  Bcast:0.0.0.0  Mask:255.255.255.0
          inet6 addr: fe80::200:ff:feaa:14/64 Scope:Link
          UP BROADCAST RUNNING MULTICAST  MTU:1500  Metric:1
          RX packets:16 errors:0 dropped:0 overruns:0 frame:0
          TX packets:6 errors:0 dropped:0 overruns:0 carrier:0
          collisions:0 txqueuelen:1000
          RX bytes:1324 (1.3 KB)  TX bytes:468 (468.0 B)

lo        Link encap:Local Loopback
          inet addr:127.0.0.1  Mask:255.0.0.0
          inet6 addr: ::1/128 Scope:Host
          UP LOOPBACK RUNNING  MTU:16436  Metric:1
          RX packets:0 errors:0 dropped:0 overruns:0 frame:0
          TX packets:0 errors:0 dropped:0 overruns:0 carrier:0
          collisions:0 txqueuelen:0
          RX bytes:0 (0.0 B)  TX bytes:0 (0.0 B)

root@Alice:/tcc#

```

Charlie's configuration (bottom):

```

root@Charlie:/tcc# ifconfig
eth0      Link encap:Ethernet  HWaddr 00:00:00:aa:00:0e
          inet addr:10.0.1.20  Bcast:0.0.0.0  Mask:255.255.255.0
          inet6 addr: fe80::200:ff:feaa:e/64 Scope:Link
          UP BROADCAST RUNNING MULTICAST  MTU:1500  Metric:1
          RX packets:16 errors:0 dropped:0 overruns:0 frame:0
          TX packets:6 errors:0 dropped:0 overruns:0 carrier:0
          collisions:0 txqueuelen:1000
          RX bytes:1324 (1.3 KB)  TX bytes:468 (468.0 B)

lo        Link encap:Local Loopback
          inet addr:127.0.0.1  Mask:255.0.0.0
          inet6 addr: ::1/128 Scope:Host
          UP LOOPBACK RUNNING  MTU:16436  Metric:1
          RX packets:0 errors:0 dropped:0 overruns:0 frame:0
          TX packets:0 errors:0 dropped:0 overruns:0 carrier:0
          collisions:0 txqueuelen:0
          RX bytes:0 (0.0 B)  TX bytes:0 (0.0 B)

root@Charlie:/tcc#

```

Fonte: O Autor, 2018

Inicialmente, foi descoberto o endereço IP privado dos *peers*, para isso foi executado o comando `ifconfig` em todos os nós (Figura 17). Em seguida, foi feita uma tentativa de conectividade em todos os *peers*, no entanto, apenas houve sucesso de conectividade entre Alice e Charlie (Figura 18).

Por fim, em cada *peer*, foi aberto a aplicação *Simple Chat* (Figura 19). Após alguns segundos, a aplicação já começou a encontrar os *peers* da rede. Para finalizar o experimento, foi enviando uma mensagem partindo de cada nó, na tentativa de visualizar se a mensagem seria enxergada por todos, ou seja, se existia conectividade entre Alice, Bob e Charlie.

Figura 18 – Alice, Bob e Charlie tentando contato direto

```

vcmd
root@Bob:/tcc# ping 10.0.3.20 -c 4
PING 10.0.3.20 (10.0.3.20) 56(84) bytes of data:
From 120.0.0.1 icmp_seq=1 Destination Net Unreachable
From 120.0.0.1 icmp_seq=2 Destination Net Unreachable
From 120.0.0.1 icmp_seq=3 Destination Net Unreachable
From 120.0.0.1 icmp_seq=4 Destination Net Unreachable

--- 10.0.3.20 ping statistics ---
4 packets transmitted, 0 received, +4 errors, 100% packet loss, time 2998ms

root@Bob:/tcc# ping 10.0.1.20 -c 4
PING 10.0.1.20 (10.0.1.20) 56(84) bytes of data:
From 120.0.0.1 icmp_seq=1 Destination Net Unreachable
From 120.0.0.1 icmp_seq=2 Destination Net Unreachable
From 120.0.0.1 icmp_seq=3 Destination Net Unreachable
From 120.0.0.1 icmp_seq=4 Destination Net Unreachable

--- 10.0.1.20 ping statistics ---
4 packets transmitted, 0 received, +4 errors, 100% packet loss, time 3003ms
root@Bob:/tcc#

vcmd
root@Alice:/tcc# ping 10.0.2.20 -c 4
PING 10.0.2.20 (10.0.2.20) 56(84) bytes of data:
From 110.0.0.1 icmp_seq=1 Destination Net Unreachable
From 110.0.0.1 icmp_seq=2 Destination Net Unreachable
From 110.0.0.1 icmp_seq=3 Destination Net Unreachable
From 110.0.0.1 icmp_seq=4 Destination Net Unreachable

--- 10.0.2.20 ping statistics ---
4 packets transmitted, 0 received, +4 errors, 100% packet loss, time 2998ms

root@Alice:/tcc# ping 10.0.1.20 -c 4
PING 10.0.1.20 (10.0.1.20) 56(84) bytes of data:
64 bytes from 10.0.1.20: icmp_req=1 ttl=63 time=0.087 ms
64 bytes from 10.0.1.20: icmp_req=2 ttl=63 time=0.359 ms
64 bytes from 10.0.1.20: icmp_req=3 ttl=63 time=0.365 ms
64 bytes from 10.0.1.20: icmp_req=4 ttl=63 time=0.130 ms

--- 10.0.1.20 ping statistics ---
4 packets transmitted, 4 received, 0% packet loss, time 2998ms
rtt min/avg/max/mdev = 0.087/0.235/0.365/0.128 ms
root@Alice:/tcc#

vcmd
root@Charlie:/tcc# ping 10.0.2.20 -c 4
PING 10.0.2.20 (10.0.2.20) 56(84) bytes of data:
From 110.0.0.1 icmp_seq=1 Destination Net Unreachable
From 110.0.0.1 icmp_seq=2 Destination Net Unreachable
From 110.0.0.1 icmp_seq=3 Destination Net Unreachable
From 110.0.0.1 icmp_seq=4 Destination Net Unreachable

--- 10.0.2.20 ping statistics ---
4 packets transmitted, 0 received, +4 errors, 100% packet loss, time 3007ms

root@Charlie:/tcc# ping 10.0.3.20 -c 4
PING 10.0.3.20 (10.0.3.20) 56(84) bytes of data:
64 bytes from 10.0.3.20: icmp_req=1 ttl=63 time=0.099 ms
64 bytes from 10.0.3.20: icmp_req=2 ttl=63 time=0.411 ms
64 bytes from 10.0.3.20: icmp_req=3 ttl=63 time=0.293 ms
64 bytes from 10.0.3.20: icmp_req=4 ttl=63 time=0.197 ms

--- 10.0.3.20 ping statistics ---
4 packets transmitted, 4 received, 0% packet loss, time 2997ms
rtt min/avg/max/mdev = 0.099/0.250/0.411/0.115 ms
root@Charlie:/tcc#

```

Fonte: O Autor, 2018

Figura 19 – Alice, Bob e Charlie trocando mensagens via *Simple Chat*

```

vcmd
Simple Chat (0 B/s)
=====

>> Connecting...
>> Success! network is alive
>> you can type "users" to see the list of users.

>> New user on board! "e3218b55fe05b8bf"
>> New user on board! "0134a2adbe986392"
81f00a4ed019dcd7: hello Alice and Charlie
e3218b55fe05b8bf: hello Bob and Charlie
0134a2adbe986392: hello Alice and Bob

Say:

vcmd
Simple Chat (32 B/s)
=====

>> Connecting...
>> Success! network is alive
>> you can type "users" to see the list of users.

>> New user on board! "81f00a4ed019dcd7"
>> New user on board! "0134a2adbe986392"
81f00a4ed019dcd7: hello Alice and Charlie
e3218b55fe05b8bf: hello Bob and Charlie
0134a2adbe986392: hello Alice and Bob

Say:

vcmd
Simple Chat (151 B/s)
=====

>> Connecting...
>> Success! network is alive
>> you can type "users" to see the list of users.

>> New user on board! "81f00a4ed019dcd7"
>> New user on board! "e3218b55fe05b8bf"
81f00a4ed019dcd7: hello Alice and Charlie
e3218b55fe05b8bf: hello Bob and Charlie
0134a2adbe986392: hello Alice and Bob

Say:

```

Fonte: O Autor, 2018

5.2.3 Discussão

Analisando os resultados, pode-se observar que em ambos os cenários foi possível obter conectividade entre os *peers*. No primeiro experimento, a conectividade foi testada com utilização da ferramenta Mininet de uma forma automatizada. Contudo, no experimento que utilizou o CORE, apesar da não utilização de scripts automatizados, proporcionou uma experimentação mais próxima de uma utilização real da biblioteca por usuários reais.

6 CONSIDERAÇÕES FINAIS

Com a demanda exigida pelo crescimento da Internet, foi necessária utilização de técnicas de tradução de endereços para garantir que os novos usuários pudessem ter acesso a rede mundial de computadores. No entanto, essa tradução de endereços dificulta a comunicação direta entre *peers* em uma rede P2P.

Visto isso, este trabalho realizou uma pesquisa bibliográfica no intuito de construir uma base teórica com objetivo de desenvolver uma biblioteca JavaScript que solucionasse o problema da conectividade direta entre os *peers* sem alterar a infraestrutura atual da Internet. A biblioteca foi desenvolvida para rodar em ambiente Node.js e utiliza o protocolo e rede BitTorrent DHT. Utilizando os nós DHT como ponto de encontro e troca de informações dos *peers*, que estão por trás, de NAT's e desejam se comunicar, como sugerido por Moore *et al.* (2015). Também foi feita publicação da biblioteca no repositório de códigos NPM, facilitando assim que outros desenvolvedores possam modificar e utilizar a mesma em seus projetos.

Foram realizados testes afim de comprovar a eficácia da biblioteca desenvolvida, sendo que, a aplicação foi submetida a testes de unidade e a testes de sistema em ambiente de rede simulado. Utilizando para isso duas ferramentas de simulação distintas. Ao final dos testes foi observado que os mesmos obtiveram sucesso em realizar comunicação entre *peers* em redes privadas disjuntas que se conectavam através de NAT's, alcançando resultados satisfatórios e semelhantes.

Visto posto, os objetivos deste trabalho foram alcançados com a revisão bibliográfica e o desenvolvimento da biblioteca elaborada neste trabalho. Também com a realização de testes e a disponibilização da mesma publicamente, contribuindo com ecossistema Node.js.

6.1 LIMITAÇÕES

Apesar de o método de travessia de NAT escolhido para compor este projeto ter um alto grau de funcionamento (82% vide Tabela 3 na Subseção 2.3.1), ele não se mostra eficaz em NATs simétricos (FORD; SRISURESH; KEGEL, 2005). Neste ponto, a biblioteca desenvolvida neste trabalho se limita a funcionar apenas se os *peers* componentes do grupo acessarem a Internet diretamente ou por meio de NAT's não simétricos.

6.2 TRABALHOS FUTUROS

É sugerido para trabalhos futuros a implementação do protocolo UTP. Este protocolo estende o protocolo UDP adicionando a ele as garantias existentes no protocolo TCP. Com isso, pode-se destacar o fato que os *peers* poderão se comunicar em forma de fluxo de dados, e não apenas em troca de mensagens separadas.

Uma outra sugestão de trabalho futuro é fazer uma revisão na literatura sobre técnicas de travessia de NAT's simétricos e acrescentar uma ou mais técnicas na biblioteca desenvolvida por este trabalho. Tendo como motivação o aumento na cobertura de eficácia de utilização da biblioteca. Ficando limitados apenas a NAT's que a literatura não disponibilizar meios para travessia.

Outro trabalho importante sugerido é a realização de uma análise de desempenho apurada da biblioteca desenvolvida. Mensurando diferentes pontos da biblioteca no intuito de verificar se a mesma pode ser utilizada em projetos que exigem determinadas características da rede. Um exemplo a destacar seria a utilização da biblioteca em projetos de telefonia VoIP (*Voice over IP*) que necessita de uma baixa latência, pois atrasos não são aceitáveis nesse tipo de aplicação (WETHERALL; TANENBAUM, 2011).

REFERÊNCIAS

- BARBERA, M.; LOMBARDO, A.; SCHEMBRA, G.; TRIBASTONE, M. A markov model of a freerider in a bittorrent p2p network. In: IEEE. **Global Telecommunications Conference, 2005. GLOBECOM'05. IEEE**. [S.l.], 2005. v. 2, p. 5–pp.
- BASET, S. A.; SCHULZRINNE, H. An analysis of the skype peer-to-peer internet telephony protocol. **arXiv preprint cs/0412017**, 2004.
- CARPENTER, B. **Architectural principles of the Internet**. [S.l.], 1996.
- COHEN, B. **The BitTorrent Protocol Specification**. 2008. Disponível em: <http://www.bittorrent.org/beps/bep_0003.html>. Acesso em: 12 jun. 2018.
- DEBIAN.ORG. **Obtendo imagens de CD do Debian com BitTorrent**. 2018. Disponível em: <<https://www.debian.org/CD/torrent-cd/>>. Acesso em: 12 jun. 2018.
- FLANAGAN, D. **JavaScript: O guia definitivo**. [S.l.]: Bookman Editora, 2007.
- FORD, B.; SRISURESH, P.; KEGEL, D. Peer-to-peer communication across network address translators. In: **USENIX Annual Technical Conference, General Track**. [S.l.: s.n.], 2005. p. 179–192.
- HARRISON, D. **Index of BitTorrent Enhancement Proposals**. 2008. Disponível em: <http://www.bittorrent.org/beps/bep_0000.html>. Acesso em: 03 set. 2018.
- HEATER, B. **Blockchain startup Tron closes BitTorrent acquisition | TechCrunch**. 2018. Disponível em: <<https://techcrunch.com/2018/07/24/blockchain-startup-tron-closes-bittorrent-acquisition/>>. Acesso em: 14 ago. 2018.
- JOHANSSON, O. E. **IPv6 Philosophy: To NAT or not to NAT – that’s the question » IPv6 Friday**. 2018. Disponível em: <<http://ipv6friday.org/blog/2011/12/ipv6-nat/>>. Acesso em: 11 ago. 2018.
- JOHNSEN, J. A.; KARLSEN, L. E.; BIRKELAND, S. S. Peer-to-peer networking with bittorrent. **Department of Telematics, NTNU**, 2005.
- KAMIENSKI, C.; SOUTO, E.; ROCHA, J.; DOMINGUES, M.; CALLADO, A.; SADOK, D. Colaboração na internet e a tecnologia peer-to-peer. In: **XXV Congresso da Sociedade Brasileira de Computação–SBC2005**. [S.l.: s.n.], 2005. v. 25.
- LOEWENSTERN, A.; NORBERG, A. **DHT Protocol**. 2008. Disponível em: <http://www.bittorrent.org/beps/bep_0005.html>. Acesso em: 03 set. 2018.
- MAHY, R.; MATTHEWS, P.; ROSENBERG, J. Traversal using relays around nat (turn): Relay extensions to session traversal utilities for nat (stun) rfc 5766. **Internet Society Request for Comments**, p. 6, 2010.
- MATTHEWS, P.; ROSENBERG, J.; WING, D.; MAHY, R. Session traversal utilities for nat (stun). **RFC 5389**, 2008.
- MOORE, R.; MORRELL, C.; MARCHANY, R.; TRONT, J. G. Utilizing the bittorrent dht for blind rendezvous and information exchange. In: IEEE. **Military Communications Conference, MILCOM 2015-2015 IEEE**. [S.l.], 2015. p. 1560–1565.

MULLER, A.; EVANS, N.; GROTHOFF, C.; KAMKAR, S. Autonomous nat traversal. In: **IEEE. Peer-to-Peer Computing (P2P), 2010 IEEE Tenth International Conference on**. [S.l.], 2010. p. 1–4.

NODEJS.ORG. **Node.js**. 2018. Disponível em: <<https://nodejs.org/en/>>. Acesso em: 11 jun. 2018.

NORBERG, A.; SILOTI, S. **Storing arbitrary data in the DHT**. 2014. Disponível em: <http://bittorrent.org/beps/bep_0044.html>. Acesso em: 03 set. 2018.

NPMJS.COM. **01 - What is npm? | npm Documentation**. 2018. Disponível em: <<https://docs.npmjs.com/getting-started/what-is-npm>>. Acesso em: 31 ago. 2018.

PEREIRA, C. R. **Aplicações web real-time com Node.js**. [S.l.]: Editora Casa do Código, 2014.

REKHTER, Y.; MOSKOWITZ, B.; KARRENBERG, D.; GROOT, G. J. de; LEAR, E. **Address allocation for private internets**. [S.l.], 1996.

ROSENBERG, J. Rfc5245: Interactive connectivity establishment (ice): A protocol for network address translator (nat) traversal for offer/answer protocols. **Request for Comments**, v. 5245, 2010.

ROSENBERG, J.; WEINBERGER, J.; HUITEMA, C.; MAHY, R. Rfc 3489. **STUN, IETF, March**, v. 54, 2003.

SRISURESH, P. Jasmine networks, k. Egevang, et al. **Traditional IP Network Address Translator (Traditional NAT)**, RFC3022, 2001.

SRISURESH, P.; FORD, B.; KEGEL, D. **RFC 5128–State of PeertoPeer (P2P) Communication across Network Address Translators (NATs)**. Mar. 2008. 2008.

SRISURESH, P.; HOLDREGE, M. **IP network address translator (NAT) terminology and considerations**. [S.l.], 1999.

STACK OVERFLOW. **Stack Overflow Developer Survey 2018**. 2018. Disponível em: <<https://insights.stackoverflow.com/survey/2018>>. Acesso em: 11 jun. 2018.

WAGNER, J.; STILLER, B. Udp hole punching in tomp2p for nat traversal. 2015.

WEARESOCIAL.COM. **Digital in 2018: World's internet users pass the 4 billion mark - We Are Social**. 2018. Disponível em: <<https://wearesocial.com/blog/2018/01/global-digital-report-2018>>. Acesso em: 11 jun. 2018.

WETHERALL, J.; TANENBAUM, A. Redes de computadores. 5ª edição. **Rio de Janeiro: Editora Campus**, 2011.