

Anatomia do BitTorrent

a Ciência da Computação por trás do protocolo

Paulo Cheadi Haddad Filho
Orientador: José Coelho de Pina

Trabalho de Formatura Supervisionado



IME-USP

Universidade de São Paulo
São Paulo, 2013

Sumário

Lista de Códigos-fonte	III
Lista de tarefas pendentes	V
1 Introdução	1
2 Napster, Gnutella, eDonkey e BitTorrent	2
2.1 Período pré-torrent	2
2.2 Nascimento do BitTorrent	5
2.3 Mundo pós-torrent	6
3 Anatomia do BitTorrent	8
3.1 Busca por informações	15
3.2 Tabelas Hash Distribuídas e o Kademlia	26
3.3 Peer Exchange	44
3.4 Jogo da troca de arquivos	47
4 Conceitos de Computação no BitTorrent	70
4.1 Estruturas de dados	70
4.2 Funções de hash	73
4.3 Criptografia	75
4.4 Bitfields	85
4.5 Protocolos de redes	86
4.6 Multicast	86
4.7 Roteamento de pacotes	86
4.8 Retomada de downloads	86
4.9 Conexão com a Internet	86
4.10 IPv6	87
4.11 Threads	87

4.12 Engenharia de Software	87
5 Comentários Finais	88
Glossário	89
Bibliografia	95

Lista de Códigos-fonte

3.1	trecho do conteúdo do arquivo .torrent do filme “A Noite dos Mortos Vivos”, de 1960 [45], com a parte binária truncada	12
3.2	trechos formatados de forma legível do conteúdo do arquivo .torrent do filme “A Noite dos Mortos Vivos”, de 1960 [45], com a parte binária truncada	13
3.3	link magnético do arquivo .torrent do filme “A Noite dos Mortos Vivos”, de 1960 [45], com parâmetros divididos entre linhas para melhor visualização	15
3.4	Logs do Transmission sobre uma requisição de announce e a respectiva resposta, com o conteúdo binário truncado	20
3.5	Logs do Transmission sobre uma requisição de scrape e a respectiva resposta, com o conteúdo binário truncado	22

Lista de Figuras

3.1	simulação de uma transferência torrent: o seeder, na parte inferior das figuras, possui todas as 5 partes de um arquivo, que os outros computadores - os leechers - baixam de forma independente e paralela. Fonte: [87]	10
3.2	amostra de uma rede de conexões BitTorrent	11
3.3	Árvore binária do Kademlia. O nó preto é a posição do ID 0011...; os ovais cinzas são as subárvores onde o nó preto deve possuir nós conhecidos. Fonte: [41]	28
3.4	Exemplo de uma busca na árvore de nós do Kademlia usando-se um ID. O nó preto, de prefixo 0011, encontra o nó de prefixo 1110 através de sucessivas buscas (setas numeradas inferiores). As setas superiores mostram a convergência da busca durante a execução. Fonte: [41]	29
3.5	trecho da seção de dados do torrent, com as divisões das partes e dos blocos	52
3.6	parâmetros da mensagem <i>request</i> e seus significados	53

Lista de tarefas pendentes

■ revisar como itemizes estão sendo formatados	8
■ referencio o código do comando anterior?	38
■ Tudo bem deixar deste jeito?	46
■ arrumar uso dos termos choke/unchoke	47
■ referenciar seção do algoritmo do fim de jogo	55
■ escrever	89
■ escrever	89
■ escrever	89
■ escrever	90
■ escrever	91
■ escrever	91
■ escrever	91
■ escrever	92
■ escrever	93
■ escrever e revisar o uso	94

Capítulo 1

Introdução

Aqui vou explicar o objetivo do trabalho e o que será mostrado ao longo dele.

Capítulo 2

Napster, Gnutella, eDonkey e BitTorrent

Para entendermos como e por que o BitTorrent se tornou o que é hoje, devemos voltar um pouco no tempo e rever a história que precedeu à sua criação, que é no fim da década dos anos 1990.

2.1 Período pré-torrent

Entre o final dos anos 80 e o início dos 90 [70, 86], a Internet deixou de ser uma rede de computadores usada somente por entidades governamentais, laboratórios de pesquisa e universidades, passando a ter seu acesso comercializado para o público em geral pelos [fornecedores de acesso a Internet \(ISPs\)](#) [75]. Com o advento do [formato de áudio MP3 \(MP3\)](#) [80], no final de 1991, e do seu primeiro reprodutor de áudio MP3 Winamp, o tráfego da Internet cresceu devido ao aumento da troca direta desse tipo de arquivo.

Entre 1998 e 1999, dois sites de compartilhamento gratuito de músicas foram criados: o MP3.com [79], que era um site de divulgação de bandas independentes, e o [Audiogalaxy.com](#) [59, 65]. Mais popular que o primeiro, o Audiogalaxy era um site de busca de músicas, sendo que o download e upload eram feitos a partir de um software cliente. A lista de músicas procuradas era enviada pelo site para o computador onde o usuário tinha instalado o cliente, que então conectava com o computador do outro usuário, que era indicado pelo servidor. A lista possuía todos os arquivos que um dia passaram pela sua rede. Se algum arquivo fosse requisitado, mas o usuário que o possuísse não estivesse conectado, o servidor central do Audiogalaxy fazia a ponte, pegando o arquivo para si e enviando-o para o cliente do requisitante em seu próximo login.

Os 3 anos seguintes à criação desses dois sites foram muito produtivo ao mundo das [redes](#)

peer-to-peer (P2P) de modo geral, onde surgiram alguns protocolos desse paradigma e inúmeros softwares que os implementavam. Os mais relevantes foram o Napster, o Gnutella, o eDonkey e o BitTorrent.

Napster

Em maio de 1999, surgiu o Napster [81], um programa de compartilhamento de MP3 que inovou por desfigurar o usual modelo cliente-servidor, no qual um servidor central localizava os arquivos nos usuários e fazia a conexão entre estes, onde ocorriam as transferências. O Napster foi contemporâneo ao Audiogalaxy, e ambos fizeram muito sucesso por cerca de 2 anos, até que começaram as ações judiciais contra ambos os programas.

Não demorou muito tempo para a indústria da música entrar em ação contra a troca de arquivos protegidos por direitos autorais sem autorização dos detentores de tais direitos pela Internet. Seu primeiro alvo foi o Napster, em dezembro de 1999, quando a [RIAA \(do inglês Recording Industry Association of America\)](#) entrou com processo judicial representando várias gravadoras, alegando quebra de direitos autorais [36]. Em abril de 2000, foi a vez da banda Metallica processar, como retaliação à descoberta de que uma música ainda não lançada oficialmente já circulava na rede [31, 35]. Um mês depois, outra ação judicial, agora encabeçada pelo rapper Dr. Dre, que tinha feito um pedido formal para a retirada de seu material de circulação [17]. Isso fez com que o Napster recebesse atenção da mídia, ganhando popularidade e atingindo os 20 milhões de usuários em meados do ano 2000 [46].

Em 2001, esses imbróglis judiciais resultaram numa liminar federal que ordenou que o Napster retirasse o conteúdo protegido das entidades representadas pela RIAA. O Napster tentou cumprir a ordem judicial, mas a juíza do caso não ficou satisfeita, ordenando então, em julho daquele ano, o desligamento da rede enquanto não conseguisse controlar o conteúdo que trafegava ali [81]. Em setembro, o Napster fez um acordo [63], onde pagou 26 milhões de dólares pelos danos já causados, pelo uso indevido de músicas, e mais 10 milhões de dólares pelos danos futuros envolvendo royalties. Para pagar esse valor, o Napster tentou cobrar o serviço que prestava de seus usuários, que acabaram migrando de rede P2P, inclusive para o Audiogalaxy. Não conseguindo quitar o acordo, em 2002, o Napster decreta falência e é forçado a liquidar seus ativos. De lá para cá, foi negociado algumas vezes, e, atualmente, pertence ao site Rhapsody [53].

O sucesso do Napster, mesmo que por curto período tempo, mostrou o potencial que as redes P2P poderiam ter, e com isso, novos softwares e protocolos de redes foram sendo lançados, sempre tentando se diferenciar dos seus antecessores, a fim de não serem novos alvos de ações judiciais. A solução para isso foi tentar descentralizar os mecanismos de indexação e de busca, que foram os pontos fracos do Napster.

Gnutella

O sucessor foi o [Gnutella](#), que em março de 2000 [72], surgiu como uma resposta de domínio público, feita com “gambiarra”, para os problemas que o Napster encontrou com relação às acusações de violação de direitos autorais. Enquanto o Napster possuía em sua estrutura um servidor central, fato este que foi explorado em seu julgamento como prova de que o sistema encorajava a violação de direitos autorais, o Gnutella foi modelado como um sistema P2P puro, onde todos os nós da rede (*peers*) são completamente iguais, sendo responsáveis pelos seus próprios atos.

O Gnutella disponibiliza arquivos da mesma forma que o Napster [5], mas sem a limitação de ser de em formato de música, ou seja, qualquer arquivo pode ser compartilhado. A diferença mais significativa entre os dois protocolos é o algoritmo de busca: a abordagem do Gnutella é baseada numa forma de *anycast*. Isso envolve duas partes: a primeira, sobre como cada usuário é conectado a outros nós e mantém a lista dessas conexões atualizada; a segunda, sobre como ele trata as buscas e trabalha inundando de pedidos para todos os nós que estão a uma certa distância do usuário (nó-cliente). Por exemplo, se a distância limite for de 4, então todos os nós que estiverem a 4 passos a partir do cliente serão verificados, começando a partir dos mais próximos. Eventualmente, algum nó possuirá o arquivo requisitado e responderá, e assim será feita a transferência desse arquivo. Muitos softwares que implementam o protocolo vão além dessa funcionalidade básica de download simples, tentando transferir de forma paralela partes diferentes do arquivo desejado de nós diferentes, de forma a amenizar eventuais problemas de velocidade de rede.

Assim, experiências sugerem que o sistema escala para um tamanho maior, tornando o mecanismo de *anycast* extremamente caro, e, em alguns casos, até proibitivo. O problema ocorre nas buscas por arquivos menos populares, onde será necessário um maior número de nós perguntados.

O Gnutella ainda teve uma segunda versão [71], no final de 2002, onde utilizou o mesmo protocolo que o original, porém, organizando a rede de *peers* em folhas (*leafs*) e *hubs*. Um *hub* poderia ter centenas de conexões à outras folhas, mas apenas 7 (em média) a outros *hubs*, enquanto uma folha se conectaria a apenas 2 *hubs* simultaneamente. Essa nova topologia, somada com uma nova tabela de índice de arquivos das folhas mantida pelos *hubs* onde estavam conectados, melhorou o desempenho das buscas, que era ruim na versão antiga.

eDonkey

O protocolo [eDonkey](#) inovou em muitos aspectos em relação aos seus precursores, tendo papel fundamental na história das redes P2P, consolidando-se como ferramenta de compartilhamento especializado em arquivos grandes.

O eDonkey implementou o primeiro método de download por **enxame de peers** (*swarming*), onde peers fazem downloads de diferentes partes de um arquivo e de peers diferentes, utilizando de forma efetiva a largura de banda de rede para todos os peers, ao invés de ficar limitado somente à banda de um único peer.

Outra melhoria deu-se na busca de arquivos: no seu lançamento, os servidores eram separados entre si, porém, nas versões seguintes, permitiu-se que eles formassem uma rede de buscas. Isso possibilitou que os servidores repassassem buscas de seus clientes conectados localmente a outros servidores, facilitando a localização de peers conectados em qualquer servidor da rede de buscas, aumentando a capacidade de download do enxame.

Diferentemente do Napster, o eDonkey utilizou-se de **valores hash** de arquivos nos resultados de busca ao invés dos simples nomes dos arquivos. As buscas geradas pelos usuários eram baseadas em palavras-chave e comparadas com a lista de nomes de arquivos armazenada no servidor, mas o servidor retornava uma lista de pares de nomes de arquivos com seus respectivos valores hash. Enfim, quando o usuário selecionasse o arquivo desejado, o cliente iniciaria o download do arquivo usando o seu valor hash. Desse modo, um arquivo poderia ter muitos nomes entre os diferentes peers e servidores, mas seria considerado idêntico para download se possuísse o mesmo valor hash.

A arquitetura da rede em dois níveis, usando cliente e servidor, alcançou um meio termo entre as redes centralizadas (como o Napster) e as descentralizadas (como o Gnutella), já que o servidor central no primeiro era um alvo garantido para ações judiciais, enquanto o segundo mostrou-se inviável à propositura de tais ações, devido ao tráfego massivo de buscas entre peers.

Por fim, a inovação mais importante foi o uso de **tabelas hash distribuídas (DHTs)**, em específico o **Kademlia**, como algoritmo de indexação e busca nos servidores centrais dos arquivos através da rede eDonkey. Além de ser uma das razões da melhoria no desempenho das pesquisas, os DHTs possuem ainda outras características, tais como tolerância a falhas e escalabilidade.

2.2 Nascimento do BitTorrent

Em meados dos anos 1990, Bram Cohen era um programador que tinha largado a faculdade no segundo ano do curso de Ciência da Computação, da Universidade de Buffalo -- Nova Iorque, para trabalhar em empresas “ponto com”. A última delas foi a MojoNation, uma empresa que desenvolvia um software de distribuição de arquivos criptografados por swarming.

Em abril de 2001, Bram saiu da MojoNation e começou a modelar o protocolo BitTorrent, lançando a primeira implementação usando a linguagem Python, em julho de 2001. Em fevereiro de 2002, ele apresentou o seu trabalho na CodeCon [10], e na mesma época começou a testá-lo,

usando como chamariz uma coleção de material pornográfico para atrair [beta testers](#) [58]. Assim, o software começou a ser usado imediatamente.

Nesse meio tempo, Bram ainda passou pela Valve [67], empresa de desenvolvimento de jogos, trabalhando no sistema de distribuição online do jogo Half Life 2. Em 2004, saiu da Valve e voltou o foco ao Torrent. Em setembro, fundou a BitTorrent Inc. com seu irmão Ross Cohen e o parceiro de negócios Ashwin Navin, sendo então responsável pelo desenvolvimento do protocolo. Ainda naquele ano, surgiram os primeiros programas de televisão e filmes compartilhados na rede através do BitTorrent, o que popularizou o protocolo.

Em maio de 2005, a empresa lançou uma nova versão do BitTorrent, que não precisava de [rastreadores \(trackers\)](#), juntamente com um site de buscas de conteúdo torrent na Internet. Em setembro, a empresa recebeu investimento na ordem de \$8.5 milhões de dólares. No final desse ano, a BitTorrent Inc. e a MPAA (*Motion Picture Association of America*, associação americana de produtoras de filmes) fizeram um acordo [78] visando a retirada dos conteúdos não autorizados dos representados pela associação, o que não evitou a pirataria, pois já havia outros sites de busca de torrent sem restrições de conteúdo, como o TorrentSpy, Mininova, The Pirate Bay, etc.

2.3 Mundo pós-torrent

Desde o fechamento de seu site de buscas, a BitTorrent Inc. tem desenvolvido outros softwares baseados na tecnologia P2P [6], como transmissão de vídeos ao vivo (BitTorrent Live), sincronização de arquivos entre computadores ligados à Internet (BitTorrent Sync), publicação e distribuição de conteúdo de artistas a seus fãs (BitTorrent Bundles), entre outros serviços comerciais.

Como protocolo, o BitTorrent criou um novo paradigma de transmissão de informações pela Internet, sendo utilizado de inúmeras formas e motivos, tais como:

- alguns softwares de podcasting, como o Miro [44], passaram a usar o protocolo como forma de lidar com a grande quantidade de downloads de programas online;
- o site da gravadora DGM Live fornece o conteúdo via torrent após a venda [14];
- VODO [60] é um site de divulgação e distribuição de filmes sob a licença Creative Commons e que faz a publicação em outros sites de busca de torrents;
- canais como a americana CBC [8] e a holandesa VPRO [13] já disponibilizaram programas de sua grade para download. A norueguesa NRK o faz para conteúdos em HD [49] e, apesar de algumas restrições de direitos, tem aumentado a oferta;
- o serviço Amazon S3, de armazenamento de conteúdo via web service, permite o uso de torrent para a transmissão de arquivos [2];

- as empresas de desenvolvimento de jogos CCP Games (Eve Online) e Blizzard (Diablo III, StarCraft II e World of Warcraft) usaram o protocolo para distribuir o instalador de seu jogo [3], e distribuir os jogos e suas eventuais atualizações [7], respectivamente;
- o governo britânico distribuiu os detalhes de seus gastos [22], enquanto a Universidade do Estado da Flórida utiliza para transmitir grandes conjuntos de dados científicos aos seus pesquisadores [32];
- Facebook [20] e Twitter [21] o usam para atualizar os seus sites, enviando de forma eficiente o código novo para seus servidores de aplicação [19].

Em 2013, o BitTorrent é um dos maiores geradores de tráfego de rede do mundo, de forma crescente, ao lado do NetFlix, Youtube, Facebook e acessos HTTP [33].

Questões legais

Desde que surgiu, o BitTorrent, bem como os outros protocolos P2P, chamou a atenção dos defensores de direitos autorais, por conta do compartilhamento não autorizado de arquivos protegidos por tais direitos, sendo alvo de medidas judiciais. Porém, assim como o Gnutella, e ao contrário do Napster, por possuir uma estrutura descentralizada e não armazenar dados sobre os compartilhamentos realizados, dificulta o trabalho de identificação das pessoas que compartilham esses dados.

Ainda assim, não existe um consenso sobre os efeitos financeiros do compartilhamento de arquivos protegidos por direitos autorais, onde o principal argumento utilizado pelos reclamantes é que estes têm grandes prejuízos e, por isso, entram com ações indenizatórias de valores vultosos. Existem alguns estudos que tentam medir esses prejuízos; um dos mais recentes, mostrou que não existem evidências de diminuição das receitas das empresas cujo conteúdo é pirateado, e que o combate aos usuários infratores não tem o impacto esperado, que é o de reduzir o compartilhamento desses arquivos [52].

Estudos acadêmicos

Academicamente, o protocolo é bastante estudado desde o seu surgimento, sendo focos de pesquisa os efeitos do algoritmo original e ajustes finos de seu funcionamento. Os pontos principais são a parte algorítmica da troca de pedaços dos arquivos, estudos sobre as topologias das redes formadas pelos peers e melhoria da eficiência do protocolo com alterações nessas topologias.

Capítulo 3

Anatomia do BitTorrent

revisar como os itens estão sendo formatados

O BitTorrent é uma rede **P2P**, onde cada um de seus usuários assume o papel híbrido de servidor (que fornece os arquivos) e de cliente (que adquire os arquivos). Cada computador é chamado de **peer**.

Cada transferência por BitTorrent está associada a um **arquivo de extensão .torrent**, que contém **metadados**, que são informações sobre arquivos que constituem um pacote chamado **torrent**. Além disso, possui um ou mais endereços de **trackers**, que são servidores que mantêm listas atualizadas dos peers que estão compartilhando aqueles arquivos, atualizadas em curtos períodos de tempo (usualmente 30 minutos).

Enquanto um peer estiver fazendo download de um torrent, ele é chamado de **sugador (leecher)**, pois ainda consumirá dados de outros peers; quando o download acabar, passará a ser um **semeador (seeder)**, que somente enviará dados.

Os arquivos .torrent ficam disponíveis em vários sites de índice (às vezes, chamados de comunidades), como o **ThePirateBay**, o **Kickass** ou **Torrentz** (muitas vezes em mais de um deles ao mesmo tempo). Apesar de todo conteúdo compartilhado possuir um arquivo .torrent, não necessariamente um arquivo .torrent está sendo compartilhado naquele momento, podendo até mesmo estar extinto.

Peers que participam do compartilhamento de um torrent específico fazem parte do **enxame (swarm)**, onde os dados contidos nesse torrent são compartilhados por partes com outros peers.

A quantidade total de partes varia de acordo com cada torrent: o tamanho total dos arquivos contidos nesse torrent é dividido em blocos de tamanho fixo (geralmente 256kB) e transmitido de forma independente das outras partes, seguindo uma ordem estabelecida pelo algoritmo de

troca de partes (explicado na seção [3.4](#)).

Essa ordem varia de acordo com o estado atual do swarm desse torrent.

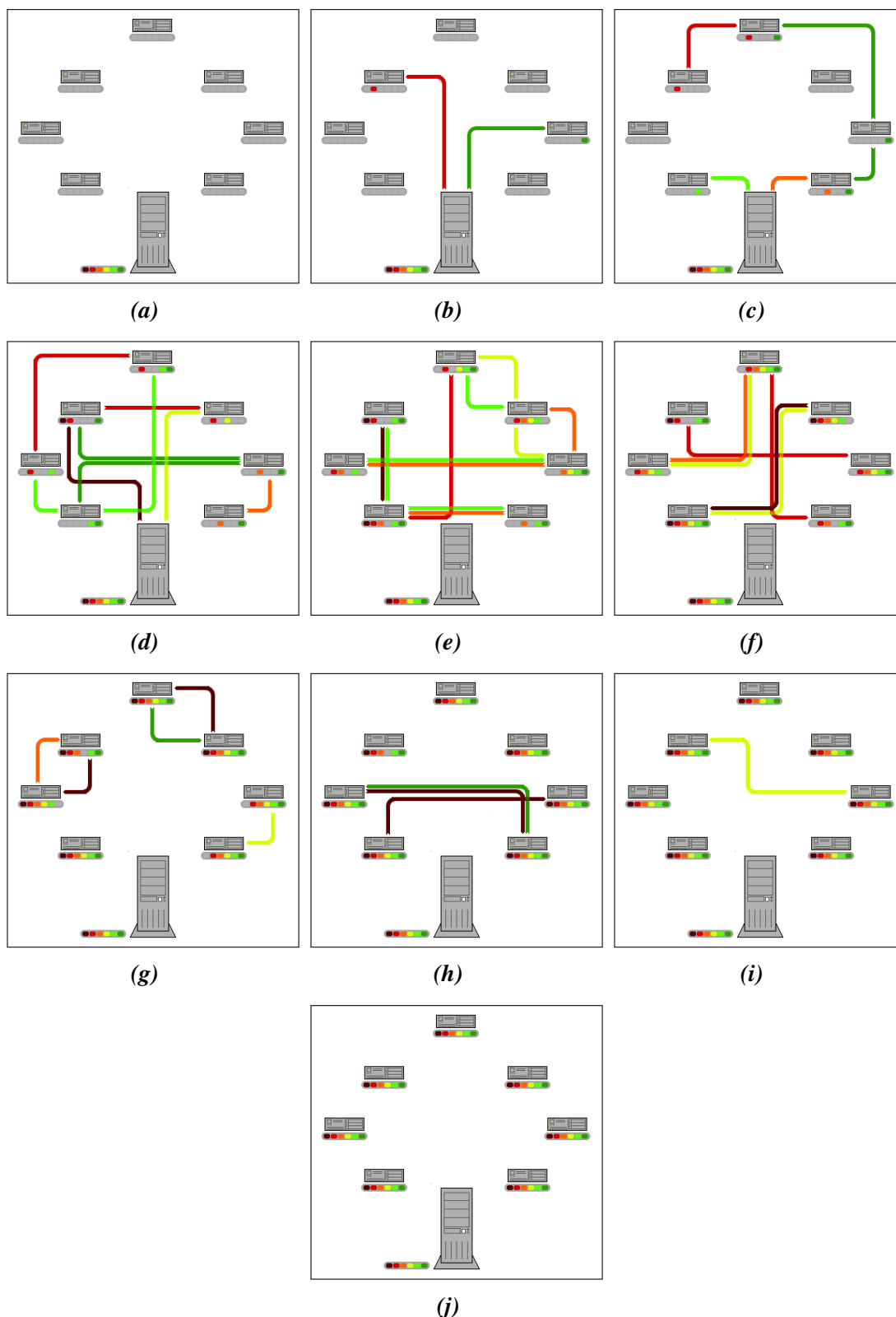


Figura 3.1: simulação de uma transferência torrent: o seeder, na parte inferior das figuras, possui todas as 5 partes de um arquivo, que os outros computadores - os leechers - baixam de forma independente e paralela. Fonte: [87]

Todos esses agentes possuem relações múltiplas entre si. Por exemplo, um mesmo arquivo .torrent pode estar indexado por vários sites indexadores. Como veremos nos capítulos seguintes, eles contêm uma informação que os identificam unicamente, gerando consistência entre esses vários sites de busca. Outra observação a ser feita é que um peer pode estar baixando um ou mais torrents simultaneamente, ou seja, participando de vários swarms ao mesmo tempo. Por fim, em alguns casos, um mesmo torrent pode possuir uma grande quantidade de peers participantes, havendo necessidade de dividi-los em vários swarms, para fins de escalabilidade da rede formada.

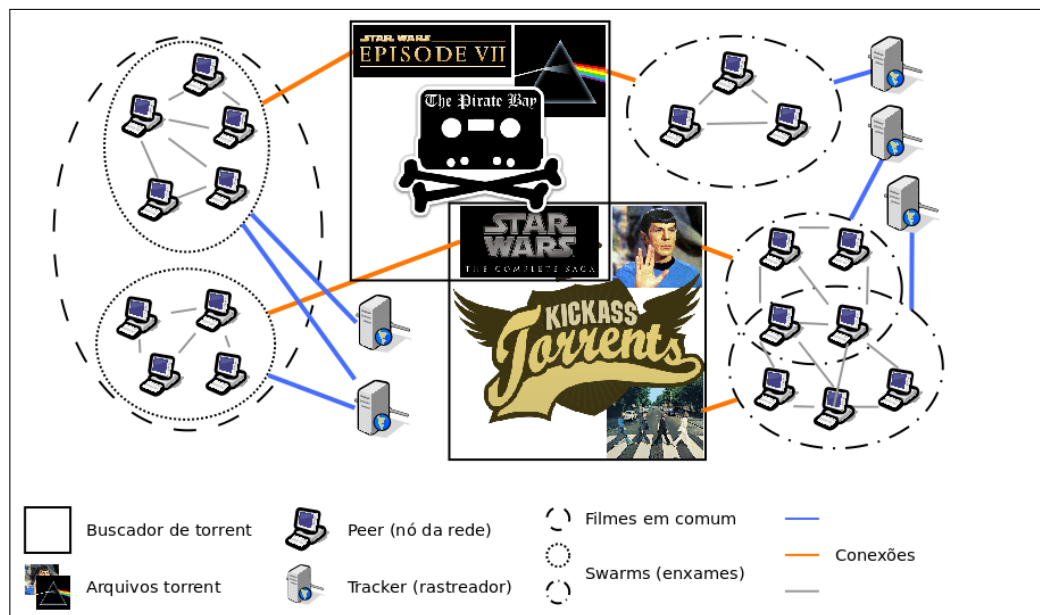


Figura 3.2: amostra de uma rede de conexões BitTorrent

Arquivo .torrent

Ao se adicionar um arquivo .torrent em um programa cliente, ocorrem muitas transmissões de dados antes do download de fato. Para demonstrar isso, usaremos um arquivo .torrent do filme “A Noite dos Mortos Vivos”, de 1960 [45], que é de domínio público e livre de direitos autorais.

Se abrirmos esse arquivo, veremos uma grande [sequência de caracteres \(string\)](#), caracteres diferentes e incomuns, formando um conteúdo ilegível (na seção binária) e sob uma forma compacta, mostrado abaixo.

```
1 d8:announce36:http://bt1.archive.org:6969/announce13:announce-list1136:http://bt1.
2 archive.org:6969/announce136:http://bt2.archive.org:6969/announce7:comment13:crea
3 tiondatei1343715473e4:infod5:filesld5:crc328:030208fe6:lengthi4127671704e3:md532:627
4 f5a428f9e454ccfcb29d31b87169a5:mtime10:10794024804:path129:night_of_the_living_dead.
5 mpege4:sha140:5e44bb1b3f700240249a5287c64dc02dc56d034bee4:name24:night_of_the_living
6 _dead12:piecelengthi4194304e6:pieces23720:<binary>
7
8 (...)
9
10 e6:locale2:en5:title24:night_of_the_living_dead8:url-list128:http://archive.org/
11 download/39:http://ia600301.us.archive.org/22/items39:http://ia700301.us.archive.
12 org/22/itemsee
```

Código-fonte 3.1: trecho do conteúdo do arquivo .torrent do filme “A Noite dos Mortos Vivos”, de 1960 [45], com a parte binária truncada

Esse conteúdo está organizado usando a [codificação B \(bencode\)](#), que é uma codificação compacta de arquivos, especial para arquivos .torrent, e ininteligível. Com alguma formatação, podemos enxergar os componentes separadamente, como mostra o código 3.2.

Esse conteúdo tem significado, sendo utilizado da seguinte forma [89]:

- **strings** são prefixos de números na base 10, que representam comprimentos, seguidos por um caractere `:` e então o conteúdo. Por exemplo, na linha 2, `8:announce` corresponde à string `"announce"`.
- **números** são representados por um `i`, seguidos do valor na base 10 (sem qualquer limite de tamanho, mas sem zeros precedentes - como em `0003` - e pode ser negativo), terminados por um `e`. Por exemplo, na linha 11, `i1343715473e` corresponde ao número `1343715473`.
- **listas** são formadas por `l`, seguidos por seus elementos (também no formato bencode), e então terminados por `e`. Por exemplo, `l3:foo3:bare` corresponde a `["foo", "bar"]`. No código 3.2, é presente entre as linhas 43 e 47.
- **dicionários** são definidos por `d`, seguidos de uma lista alternada de chaves e seus valores correspondentes, terminando com `e`, onde as chaves devem estar ordenadas

usando-se comparação binária, ao invés da usual alfanumérica. Por exemplo, a string `d3:foo3:bar6:foobar6:bazbare` corresponde ao dicionário puro `{"foo": "bar", "foobar": "bazbar"}`, e a estrutura mais complexa dada por `d3:fool6:foobar3:bazee` equivale a `{"foo": ["foobar", "baz"]}`.

```
1 d
2   8:announce
3   36:http://bt1.archive.org:6969/announce
4   13:announce-list
5   1
6     136:http://bt1.archive.org:6969/announcee
7     136:http://bt2.archive.org:6969/announcee
8   e
9   7:comment
10  13:creation date
11  i1343715473e
12  4:info
13  d
14    5:files
15    1
16      d
17        5:crc32
18        8:030208fe
19        6:length
20        i4127671704e
21        3:md5
22        32:627f5a428f9e454ccfcb29d31b87169a
23        5:mtime
24        10:1079402480
25        4:path
26        l29:night_of_the_living_dead.mpege
27        4:shal
28        40:5e44bb1b3f700240249a5287c64dc02dc56d034b
29      e
30    e
31    4:name
32    24:night_of_the_living_dead
33    12:piece length
34    i4194304e
35    6:pieces
36    23720:<binary>
37  e
38  6:locale
39  2:en
40  5:title
41  24:night_of_the_living_dead
42  8:url-list
43  1
44    28:http://archive.org/download/
45    39:http://ia600301.us.archive.org/22/items
46    39:http://ia700301.us.archive.org/22/items
47  e
48 e
```

Código-fonte 3.2: trechos formatados de forma legível do conteúdo do arquivo .torrent do filme “A Noite dos Mortos Vivos”, de 1960 [45], com a parte binária truncada

Neste arquivo, muitas informações estão contidas:

announce: são os endereços de trackers, que irão informar listas de peers para os novos nós;

info: possui lista de um ou mais arquivos que o torrent contém;

pieces: é a quantidade de partes que um arquivo possui; e

piece_length: quantidade de bytes que a respectiva parte terá.

Partes

A quantidade de partes e seus tamanhos dependem do tamanho total de dados de um torrent. De tamanhos geralmente sendo potências de 2, as partes têm esse tamanho escolhido de forma a se otimizar o processo de transferência: partes muito grandes causariam ineficiência de banda de rede, enquanto tamanhos muito pequenos gerariam trechos de valores hash muito extensos num arquivo .torrent, aumentando seu tamanho.

Todas as partes possuem tamanhos iguais, exceto pela parte final, que pode ser menor. No caso de um torrent de múltiplos arquivos, para efeito de divisão em partes, é considerado como um único arquivo contíguo, composto da concatenação de todos esses arquivos na ordem que forem listados no arquivo .torrent. Dessa forma, é possível com que uma parte contenha o final de um desses arquivos e o início do arquivo seguinte. A quantidade de partes, em ambos os casos (arquivo único ou múltiplos arquivos), é dada por $\lceil (tamanho_{total} / tamanho_{daspartes}) \rceil$.

De toda essa sequência de partes, existirá uma sequência de valores hash SHA-1 de seus dados, formando uma grande string de um único valores hash, que é colocado como valor da chave **info** do dicionário de dados do arquivo .torrent.

Historicamente, os tamanhos das partes eram escolhidos a fim de gerar arquivos .torrent entre 50kB e 75kB. Atualmente, é preferido manter tamanhos das partes em até 512kB para torrents de dados entre 8GB e 10GB. Por terem maior quantidade de dados, distribuir essa grande quantidade de bytes em poucas partes faz com que o swarm seja mais eficiente. Os tamanhos das partes mais utilizados são de 256kB, 512kB e 1MB.

Magnet Link

Além do arquivo .torrent, existe uma outra forma de se obter os metadados necessários para se iniciar a transmissão, utilizando-se de [links magnéticos \(magnet links\)](#).

Magnet links, ao contrário dos arquivos .torrent, não estão gravados em algum dispositivo de armazenamento. Basicamente, é um esquema de [identificador uniforme de recursos \(URI\)](#), usado exclusivamente para o protocolo.

No caso citado, o site de origem do arquivo .torrent que estamos usando não fornece um magnet link oficialmente. Porém, o Transmission consegue construir uma URI a partir do arquivo original, para fins de compartilhamento direto. O resultado, após decodificar o endereço

para um formato legível (retirando a [codificação de endereços URI e URL \(URL encode\)](#)) [83], foi o seguinte:

```
1 magnet:?xt=urn:btih:72d7a3179da3de7a76b98f3782c31843e3f818ee
2 &dn=night_of_the_living_dead
3 &tr=http://bt1.archive.org:6969/announce&tr=http://bt2.archive.org:6969/announce
4 &ws=http://archive.org/download/
5 &ws=http://ia600301.us.archive.org/22/items/&ws=http://ia700301.us.archive.org/22/items/
```

Código-fonte 3.3: link magnético do arquivo .torrent do filme “A Noite dos Mortos Vivos”, de 1960 [45], com parâmetros divididos entre linhas para melhor visualização

Esse endereço é composto por vários pares, compostos por nomes de parâmetros e seus respectivos valores, sem qualquer ordem específica, formando uma [string de busca \(query string\)](#). Podemos dividir esse endereço em partes, cada uma tendo o seu significado:

- **xt:** parâmetro para *exact topic*, ou tópico exato, que contém a informação mais importante do magnet link: o identificador único de torrents. Serve para encontrar e verificar os arquivos especificados. No caso, `urn:btih:<hash>` corresponde ao [nome uniforme de recursos \(URN\) btih](#) (*BitTorrent Info Hash*), que é a string [valor hash](#) resultado da [função de hash](#) SHA-1, convertida para hexadecimal;
- **dn:** parâmetro que contém o *display name*, ou nome de visualização, que é um texto de apresentação amigável para o usuário;
- **tr:** o *address tracker*, ou endereço do tracker, onde o programa cliente vai procurar as informações de peers;
- **ws:** endereço do arquivo para *webseed*, ou fornecimento web, que é o endereço de Internet de um servidor HTTP ou FTP, que será utilizado como alternativa a um swarm problemático [66];

3.1 Busca por informações

Quando adicionamos um torrent ao Transmission, o programa salva as informações em disco durante todo o período em que estas estiverem sendo gerenciadas por ele. Caso tenha sido por meio de um arquivo .torrent, uma cópia deste é salva em uma pasta pré-definida para seu controle interno; caso seja por magnet link, um novo arquivo é criado contendo as informações obtidas através dele (por questões de praticidade), para que não necessite fazer essa aquisição dos dados novamente ao ser aberto, ou quando alguma transferência for pausada e depois continuada.

Após esse arquivamento, o programa processa as informações salvas para deixá-las carregadas em memória, a fim de obter o valor hash que identifica o arquivo .torrent.

```

1 static const char*
2 tr_metainfoParseImpl(const tr_session * session, tr_info * inf,
3 bool * hasInfoDict, int * infoDictLength, const tr_variant * meta) {
4     // variáveis temporárias
5     int64_t i; size_t len; const char * str; const uint8_t * raw;
6     tr_variant * infoDict = NULL;
7
8     // flags
9     bool b, isMagnet = false;
10
11     /* info_hash: urlencoded 20-byte SHA1 hash of the value of the info key from the
12      * Metainfo file. (...) */
13     b = tr_variantDictFindDict(meta, TR_KEY_info, &infoDict);
14     if (hasInfoDict != NULL) *hasInfoDict = b;

```

Se aquele arquivo para controle interno tiver sido criado por conta de um magnet link, não possuirá a chave `info` em seu dicionário, mas deverá conter as chaves `urn:btih:<hash>` e `info_hash`.

```

1 if (!b) { // Não possui a chave "info" no dicionário.
2     // Será que não é um magnet link?
3     if (tr_variantDictFindDict(meta, TR_KEY_magnet_info, &d)) {
4         isMagnet = true;
5
6         if (!tr_variantDictFindRaw(d, TR_KEY_info_hash, &raw, &len) ||
7             len != SHA_DIGEST_LENGTH) // Se tiver a chave "info_hash" válida, a usa.
8             return "info_hash";
9
10        memcpy(inf->hash, raw, len);
11        tr_shal_to_hex(inf->hashString, inf->hash);
12        (...)
13    }
14    else return "info"; // Não é magnet link e não possui a chave "info".
15 }

```

Porém, se o arquivo para controle interno tiver sido criado como cópia do arquivo .torrent, possuirá o mesmo dicionário, que contém a chave `info_hash`, que é utilizada para calcular o valor hash do arquivo.

```

1 else {
2     int len;
3     char * bstr = tr_variantToStr(infoDict, TR_VARIANT_FMT_BENC, &len);
4     tr_shal(inf->hash, bstr, len, NULL); // Calcula o hash SHA-1 do .torrent...
5     tr_shal_to_hex(inf->hashString, inf->hash); // ...e converte de base2 para base16
6     (...)
7 }

```

Outras informações podem ser recuperadas, dependendo da origem do arquivo .torrent, tais como a privacidade do torrent, a lista de arquivos e seus respectivos tamanhos, valor hash de cada parte, entre outras. Por fim, termina coletando os [endereço web de contato do tracker \(announces\)](#).

```

1  if (!tr_variantDictFindInt(infoDict, TR_KEY_private, &i)) // privacidade do torrent
2  if (!tr_variantDictFindInt(meta, TR_KEY_private, &i)) i = 0;
3  inf->isPrivate = i != 0;
4
5  if (!isMagnet) { // quantidade de partes
6  if (!tr_variantDictFindInt(infoDict, TR_KEY_piece_length, &i) || (i < 1))
7  return "piece length";
8  inf->pieceSize = i;
9  }
10
11 if (!isMagnet) { // hashes das partes
12 if (!tr_variantDictFindRaw(infoDict, TR_KEY_pieces, &raw, &len) ||
13     len % SHA_DIGEST_LENGTH) return "pieces";
14
15 inf->pieceCount = len / SHA_DIGEST_LENGTH;
16 inf->pieces = tr_new0(tr_piece, inf->pieceCount);
17 for (i = 0; i < inf->pieceCount; i++)
18     memcpy(inf->pieces[i].hash, &raw[i * SHA_DIGEST_LENGTH], SHA_DIGEST_LENGTH);
19 }
20
21 if (!isMagnet) { // lista de arquivos
22 if ((str = parseFiles(inf, tr_variantDictFind(infoDict, TR_KEY_files),
23     tr_variantDictFind(infoDict, TR_KEY_length))))
24     return str;
25
26 if (!inf->fileCount || !inf->totalSize ||
27     (uint64_t) inf->pieceCount != (inf->totalSize+inf->pieceSize-1)/inf->pieceSize)
28     return "files";
29 }
30
31 if ((str = getannounce(inf, meta)) return str; // announce(s)
32 (...)
33 return NULL;
34 }

```

Announce

Para cada swarm gerenciado, o tracker possui uma lista dos peers que participam dele, que é enviada ao peer que a requer por meio de uma [requisição de método GET do protocolo HTTP \(HTTP GET\)](#). Quando essa requisição é recebida pelo tracker, este incluirá ou atualizará um registro para o peer solicitante, e devolverá uma lista de 50 peers aleatórios, de forma uniforme, que fazem parte do swarm. Não havendo essa quantidade total, a lista toda será enviada ao requisitante. Caso contrário, a aleatoriedade proporcionará uma diversidade de listas enviadas, ocasionando robustez ao sistema [92].

Esse contato entre um peer e um tracker é chamado de [announce](#), que pode ser feito usando-se tanto o [protocolo TCP](#), bem como o [protocolo UDP](#), e é como peers podem passar várias informações, usando um dicionário no formato bencode:

- **info_hash**: valor hash de 20 bytes resultante da função de hash SHA-1, com URL encode, do valor da chave `info` do arquivo `.torrent`;
- **peer_id**: string de 20 bytes, com URL encode, usado como identificador único do programa cliente, gerado no início da sua execução. Para isso, provavelmente deverá

incorporar informações do computador, a fim de se gerar um valor único;

- **uploaded**: a quantidade total de dados, em bytes, enviados desde o momento em que o cliente enviou o primeiro aviso ao tracker;
- **downloaded**: a quantidade total de dados, em bytes, recebidos desde momento em que o cliente enviou o primeiro aviso ao tracker;
- **left**: a quantidade total de dados, em bytes, que faltam para o requisitante terminar o download do torrent e passe a ser um seeder;
- **compact** (opcional): se o valor passado for 1, significa que o requisitante aceita respostas compactas. A lista de peers enviada é substituída por uma única string de peers, sendo que cada peer terá 6 bytes, onde os 4 bytes iniciais são o host e os 2 bytes finais são a porta de transmissão. Por exemplo, o endereço IP 10.10.10.5:80 seria transmitido como `0A 0A 0A 05 00 80`. Deve-se observar que alguns trackers suportam somente conexões deste tipo para otimização da utilização da banda de rede e, para isso, ou recusarão requisições sem `compact=1` ou, caso não as recusem, enviarão respostas compactas a menos que a requisição possua `compact=0`;
- **no_peer_id** (opcional): sinaliza que o tracker pode omitir o campo `peer_id` no dicionário de peers, porém será ignorado caso o modo compacto esteja habilitado;
- **event** (opcional): pode possuir os valores `started` (iniciado), `completed` (terminado), `stopped` (parado), ou vazio para não especificar.
 - *started* : a primeira requisição para o tracker deve enviar este valor;
 - *stopped* : avisa que o programa cliente está fechando;
 - *completed* : quando o download que estava ocorrendo termina numa mesma execução do programa cliente (não é enviado quando o programa cliente é iniciado com o torrent em 100%);
- **port** (opcional): o número da porta de conexão que o programa cliente está escutando por transmissões de dados. Em geral, portas reservadas para BitTorrent estão entre 6881 e 6889. Se esse for o caso, pode ser omitido;
- **ip** (opcional): o endereço IP verdadeiro do requisitante, no formato legível do IPv4 (4 conjuntos de número de 0 a 255 separados por `.`) ou do IPv6 (8 conjuntos de números hexadecimais de 4 dígitos separados por `:`). Não é sempre necessário, pois o endereço pode ser conhecido através da requisição. Assim, é usado quando o programa cliente está se comunicando com o tracker através de um [servidor proxy](#) ou quando ambos (cliente e tracker) estão no mesmo lado local de um roteador - com [tradução de endereço de rede \(gateway NAT\)](#) -, pois, nesse caso o endereço IP não é roteável;
- **numwant** (opcional): quantidade de peers que o requisitante gostaria de receber do tracker. É permitido valor zero. Se omitido, assumirá valor padrão de 50;
- **key** (opcional): mecanismo de identificação adicional para o programa cliente provar sua identidade, caso tenha ocorrido mudança no seu endereço IP;

- **trackerid** (opcional): se a resposta de um announce anterior continha o endereço IP de um tracker, deve ser enviado neste campo;

```

1 typedef struct {
2     (...)
3     bool partial_seed;
4     int port; // porta escutada para aguardo de chamada de peers
5     int key; // chave de sessão
6     int numwant; // qtd de peers que o cliente deseja receber do tracker
7     uint64_t up; // qtd bytes enviados desde o último evento 'started'
8     uint64_t down; // qtd bytes "bons" recebidos desde o último 'started'
9     uint64_t corrupt; // qtd bytes corrompidos recebidos desde o último 'started'
10    uint64_t leftUntilComplete; // qtd de bytes que faltam para o fim do download
11    char * url; // a URL de announce do tracker
12    char * tracker_id_str; // chave gerada por um tracker HTTP e devolvida
13    char peer_id[PEER_ID_LEN]; // o ID do peer
14    uint8_t info_hash[SHA_DIGEST_LENGTH]; // info_hash do torrent
15 }
16 tr_announce_request;

```

```

1 static void announce_request_delegate(tr_announcer * announcer,
2     tr_announce_request * request, tr_announce_response_func * callback,
3     void * callback_data) {
4     (...)
5     if (!memcmp(request->url, "http", 4)) // announce começa com "http://"
6         tr_tracker_http_announce(session, request, callback, callback_data);
7     else if (!memcmp(request->url, "udp://", 6)) // announce começa com "udp://"
8         tr_tracker_udp_announce(session, request, callback, callback_data);
9     else
10        tr_logAddError("Unsupported url: %s", request->url);
11    (...)
12 }

```

Como resposta, é recebida um outro dicionário em bencode, podendo conter as seguintes chaves:

- **failure_reason**: se presente, não podem existir outras chaves no dicionário. Seu valor é uma string de mensagem de erro legível sobre o porque a requisição falhou;
- **warning_message** (opcional): similar à chave `failure_reason`, mas com a requisição tendo sido processada normalmente. A mensagem é mostrada como um erro;
- **interval**: intervalo, em segundos, que o cliente deve esperar entre requisições de announce ao tracker;
- **min_interval** (opcional): intervalo mínimo, em segundos, entre requisições de announce. Se presente, o programa cliente não deve efetuar essas requisições acima da frequência estipulada;
- **tracker_id**: string que o programa cliente deve enviar de volta nas próximas requisições. Se ausente e um valor tiver sido passado anteriormente, o uso desse valor antigo é continuado;
- **complete**: quantidade de seeders;
- **incomplete**: quantidade de leechers;

- **peers**: pode ser uma das opções
 1. lista de dicionários bencode, com as seguintes chaves:
 - **peer_id**: identificador de um peer na forma de string, escolhido por si próprio da mesma forma que a descrito pela definição de requisição;
 - **ip**: endereço IP do peer nos formatos IPv4 (4 octetos) ou IPv6 (valores hexadecimais), ou ainda o nome de domínio DNS (string);
 - **port**: número da porta utilizada pelo peer;
 2. string binária cujo tamanho é de 6 bytes para cada peer, onde os 4 primeiros representam o endereço IP e os 2 últimos são o número da porta, em notação de rede (*big endian*);

```

1  * About to connect() to exodus.desync.com port 6969 (#8)
2  *   Trying 208.83.20.164...
3  *
4  * Connected to exodus.desync.com (208.83.20.164) port 6969 (#8)
5  > GET /announce?info_hash=%88%15%8c%7bW%e0%85%21%86~%d0%b5%de%06%5b%7dWI%cf%d7&peer_id=-TR2820-ne1joqgh8z9o&port=51413&uploaded=0&downloaded=0&left=52
6  406288292&numwant=80&key=6ee99240&compact=1&supportcrypto=1&requirecrypto=1&event=
7  started HTTP/1.1
8  User-Agent: Transmission/2.82
9  Host: exodus.desync.com:6969
10 Accept: */*
11 Accept-Encoding: gzip;q=1.0, deflate, identity
12
13 < HTTP/1.1 200 OK
14 < Content-Type: text/plain
15 < Content-Length: 136
16 <
17
18 * Connection #8 to host exodus.desync.com left intact
19 Announce response:
20 < {
21   "complete": 1,
22   "downloaded": 11,
23   "incomplete": 6,
24   "interval": 1732,
25   "min interval": 866,
26   "peers": "\"<binary>\"
27 }
```

Código-fonte 3.4: Logs do Transmission sobre uma requisição de announce e a respectiva resposta, com o conteúdo binário truncado

Essa comunicação ocorre nas seguintes situações:

- no primeiro contato do peer, para que ele tenha acesso a um swarm;
- a cada período de tempo, estipulado pelo tracker, para que o peer continue mostrando que ainda está conectado, além de poder receber endereços de peers novos;
- quando a quantidade de peers conhecidos que estiverem ativos for menor do que 5;
- quando terminar o download, notificando que passou a ser um seeder;
- quando sair do swarm, seja por desconexão ou por encerramento do programa cliente;

Scrape

Além do announce, outra forma de troca de informação entre peers e trackers se dá pelo [endereço web do tracker para informações básicas \(scrape\)](#). Geralmente usado pelos programas cliente para decidir quando realizar um announce, informa o número de peers, leechers e seeders de uma lista de um ou mais torrents. É dessa forma que os sites de indexação sabem dessas informações e as apresentam nas em páginas.

A requisição de scrape pode ser sobre todos os torrents que o tracker gerencia ou sobre um ou mais torrents em específico, quando são passados seus respectivos valores hash. Sua resposta é um dicionário na forma bencode contendo as chaves

- **files**: um dicionário contendo um par chave-valor para cada torrent especificado na requisição do scrape através do valor hash do torrent de 20 bytes:
 - **complete**: quantidade de seeders;
 - **incomplete**: quantidade de leechers;
 - **downloaded**: quantidade total de vezes que o tracker registrou uma finalização (**event=complete** ao término de um download);
 - **name** (opcional): o nome interno do torrent, como especificado pelo respectivo arquivo .torrent, na sua seção **info**;

Da mesma forma que o announce, o scrape é um endereço do tracker usando-se o TCP ou o UDP, e é tratado de forma semelhante pelo Transmission.

```
1 typedef struct {
2     // lista de info_hash de torrents para serem pesquisados por scrape
3     uint8_t info_hash[TR_MULTISCRAPES_MAX][SHA_DIGEST_LENGTH];
4     char * url; // URL de scrape
5     (...)
6 } tr_scrape_request;
7
8 struct tr_scrape_response_row { // usado para conter os dados respondidos do scrape
9     uint8_t info_hash[SHA_DIGEST_LENGTH]; // info_hash do torrent
10    int seeders; // qtd de peers que são seeders deste torrent
11    int leechers; // qtd de peers baixando este torrent
12    int downloads; // qtd de vezes que este torrent foi baixado
13    int downloaders; // qtd de leechers ativos no swarm (suportado por alguns trackers)
14 };
```

```
1 static void scrape_request_delegate(tr_announcer * announcer,
2     const tr_scrape_request * request, tr_scrape_response_func * callback,
3     void * callback_data) {
4     (...)
5     if (!memcmp(request->url, "http", 4))
6         tr_tracker_http_scrape(session, request, callback, callback_data);
7     else if (!memcmp(request->url, "udp://", 6))
8         tr_tracker_udp_scrape(session, request, callback, callback_data);
9     else
10         tr_logAddError("Unsupported url: %s", request->url);
11 }
```

```

1 * About to connect() to www.mvgroup.org port 2710 (#1)
2 * Trying 88.129.153.50...
3 * Connected to www.mvgroup.org (88.129.153.50) port 2710 (#1)
4 > GET /scrape?info_hash=%83F%24%b62%e5Q%cc4%10h%ba%1e8%e2C%f7%80%01%87 HTTP/1.1
5 User-Agent: Transmission/2.82
6 Host: www.mvgroup.org:2710
7 Accept: */*
8 Accept-Encoding: gzip;q=1.0, deflate, identity
9
10 * HTTP 1.0, assume close after body
11 < HTTP/1.0 200 OK
12 <
13 * Closing connection 1
14 Scrape response:
15 < {
16     "files": {
17         "<binary>": {
18             "complete": 9,
19             "downloaded": 59074,
20             "incomplete": 2
21         }
22     }
23 }

```

Código-fonte 3.5: Logs do Transmission sobre uma requisição de scrape e a respectiva resposta, com o conteúdo binário truncado

Convenção de scrape de trackers

O endereço de scrape pode ser obtido a partir do endereço de announce, seguindo-se a seguinte convenção:

1. comece a partir do endereço de announce
2. encontre o último caractere `/`
3. se o texto que segue a última `/` não for **announce**, é um sinal que o tracker não segue a convenção
4. caso contrário, substituir **announce** por **scrape**

Alguns exemplos:

- suportam **scrape**

1. `http://example.com/announce` → `http://example.com/scrape`
2. `http://example.com/announce.php` → `http://example.com/scrape.php`
3. `http://example.com/x/announce` → `http://example.com/x/scrape`
4. `http://example.com/announce?x2%0644` → `http://example.com/scrape?x2%0644`

- não suportam **scrape**

1. <http://example.com/a>
2. <http://example.com/announce?x=2/4>
3. <http://example.com/x%064announce>

Resultados de announce e scrape

Com a requisição do announce, o Transmission recebe a sua resposta e prepara esses dados para utilização, carregando-os em memória.

```
1 void tr_tracker_http_announce(tr_session * session, const tr_announce_request * request,  
2 tr_announce_response_func response_func, void * response_func_user_data) {  
3     char * url = announce_url_new(session, request);  
4     struct announce_data * d = tr_new0(struct announce_data, 1);  
5     (...) // preenchimento dos campos da variável 'd'  
6  
7     tr_webRun(session, url, on_announce_done, d);  
8     (...)  
9 }
```

```
1 static void on_announce_done(tr_session * session, bool did_connect, bool did_timeout,  
2 long response_code, const void * msg, size_t msglen, void * vdata) {  
3     tr_announce_response * response;  
4     struct announce_data * data = vdata;  
5     response = &data->response;  
6     (...)  
7  
8     // lê os dados da resposta do tracker  
9     if (variant_loaded && tr_variantIsDict(&benc)) {  
10         int64_t i; size_t len; tr_variant * tmp; // variáveis ...  
11         const char * str; const uint8_t * raw; // ... temporárias  
12  
13         if (tr_variantDictFindStr(&benc, TR_KEY_failure_reason, &str, &len))  
14             response->errmsg = tr_strndup(str, len);  
15  
16         if (tr_variantDictFindStr(&benc, TR_KEY_tracker_id, &str, &len))  
17             response->tracker_id_str = tr_strndup(str, len);  
18  
19         if (tr_variantDictFindInt(&benc, TR_KEY_complete, &i))  
20             response->seeders = i;  
21  
22         (...)  
23  
24         if (tr_variantDictFindRaw(&benc, TR_KEY_peers, &raw, &len)) {  
25             response->pex = tr_peerMgrCompactToPex(raw, len,  
26             NULL, 0, &response->pex_count);  
27         }  
28         else if (tr_variantDictFindList(&benc, TR_KEY_peers, &tmp)) {  
29             response->pex = listToPex(tmp, &response->pex_count);  
30         }  
31     }  
32  
33     tr_runInEventThread(session, on_announce_done_eventthread, data);  
34 }
```

Após essa preparação dos dados da resposta do announce, cuja informação principal é a lista de peers, o Transmission gerencia o tempo para a próxima requisição periódica de announce

ou scrape. Além disso, sinaliza para a [thread](#) principal do programa que recebeu a resposta do tracker.

```
1 static void on_announce_done(const tr_announce_response * response, void * vdata) {
2     struct announce_data * data = vdata;
3     (...)
4     tr_tier * tier = getTier(announcer, response->info_hash, data->tierId);
5     const time_t now = tr_time();
6
7     (...)
8
9     if (tier != NULL) {
10        tr_tracker * tracker;
11        tier->lastAnnounceTime = now;
12        tier->lastAnnounceTimedOut = response->did_timeout;
13        tier->lastAnnounceSucceeded = tier->isAnnouncing = false;
14        tier->manualAnnounceAllowedAt = now + tier->announceMinIntervalSec;
15
16        (...)
17        int i = scrape_fields = seeders = leechers = 0; const char * str;
18
19        (...)
20
21        if (response->seeders >= 0) {
22            tracker->seederCount = seeders = response->seeders;
23            (...)
24        }
25        if (response->leechers >= 0) {
26            tracker->leecherCount = leechers = response->leechers;
27            (...)
28        }
29
30        (...)
31
32        if ((i = response->min_interval)) tier->announceMinIntervalSec = i;
33        if ((i = response->interval)) tier->announceIntervalSec = i;
34
35        if (response->pex_count > 0)
36            publishPeersPex(tier, seeders, leechers, response->pex, response->pex_count);
37
38        (...)
39        tier->lastAnnounceSucceeded = true;
40        (...)
41    }
42    (...)
43 }
```

```
1 static void publishPeersPex(tr_tier * tier, int seeds, int leechers, const tr_pex * pex,
2     int n) {
3     if (tier->tor->tiers->callback) {
4         tr_tracker_event e = TRACKER_EVENT_INIT;
5         e.pex = pex; e.pexCount = n; e.messageType = TR_TRACKER_PEERS;
6         (...)
7         tier->tor->tiers->callback(tier->tor, &e, NULL);
8     }
9 }
```

A thread principal percebe que foi sinalizado o recebimento de uma resposta de tracker e começa a utilizar a lista de peers adquirida, adicionando-os ao [poço de recursos \(pool\)](#) do objeto em memória que representa o [swarm](#).

```

1 static void onTrackerResponse(tr_torrent * tor, const tr_tracker_event * event,
2 void * unused UNUSED) {
3     switch (event->messageType) {
4         case TR_TRACKER_PEERS: {
5             size_t i;
6             (...)
7             for (i = 0; i < event->pexCount; ++i)
8                 tr_peerMgrAddPex(tor, TR_PEER_FROM_TRACKER, &event->pex[i], seedProbability);
9             break;
10        }
11        (...)
12    }
13 }

```

```

1 void tr_peerMgrAddPex(tr_torrent * tor, uint8_t from, const tr_pex * pex,
2 int8_t seedProbability) {
3     if (tr_isPex(pex)) { /* safeguard against corrupt data */
4         tr_swarm * s = tor->swarm;
5         managerLock(s->manager);
6
7         if (!tr_sessionIsAddressBlocked(s->manager->session, &pex->addr))
8             if (tr_address_is_valid_for_peers(&pex->addr, pex->port))
9                 ensureAtomExists(s, &pex->addr, pex->port, pex->flags, seedProbability, from);
10
11         managerUnlock(s->manager);
12     }
13 }

```

Essa inclusão é condicionada ao fato de que o endereço de um peer ainda não está contido na lista de peers.

```

1 static void ensureAtomExists(tr_swarm * s, const tr_address * addr, const tr_port port,
2 const uint8_t flags, const int8_t seedProbability, const uint8_t from) {
3     struct peer_atom * a = getExistingAtom(s, addr);
4
5     (...)
6
7     if (a == NULL) { // Se ainda não está na lista...
8         const int jitter = tr_cryptoWeakRandInt(60 * 10);
9         a = tr_new0(struct peer_atom, 1);
10        a->addr = *addr;
11        a->port = port;
12        (...)
13
14        // ...adiciona peer na lista de peers, de forma ordenada.
15        tr_ptrArrayInsertSorted(&s->pool, a, compareAtomsByAddress);
16    }
17    (...)
18 }

```

Assim, a lista de peers do Transmission está pronta para ser utilizada nos pedidos das partes do torrent.

3.2 Tabelas Hash Distribuídas e o Kademlia

Os **DHTs** surgiram quando buscas em **redes peer-to-peer** não eram eficientes, fosse pelos problemas da supercentralização dos sistemas ou pela falta dela, tentando criar uma maneira híbrida de buscar e localizar recursos nessas redes.

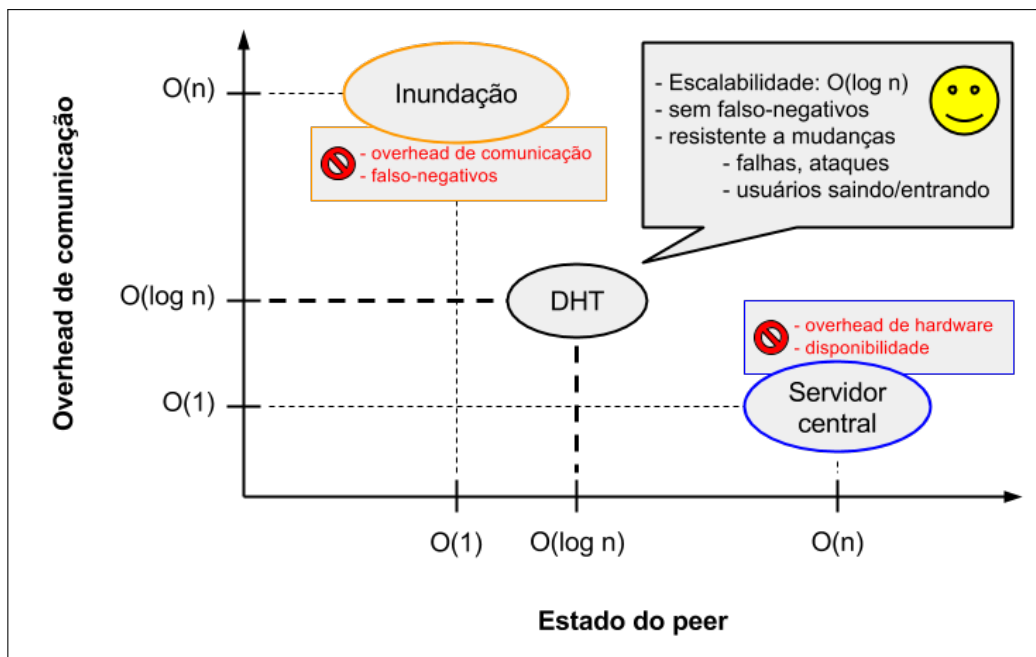
Existem duas formas de se encontrar objetos: busca e endereçamento. A primeira se baseia no casamento de palavras-chave com as das descrições dos objetos, sendo mais amigável para um usuário pois não necessita de formas complexas de identificação. Porém, é mais difícil de se tornar eficiente, além de precisar conferir também os objetos para saber se são o mesmo. As redes peer-to-peer descentralizadas utilizavam desta abordagem.

A segunda, no entanto, utiliza endereços que possam identificá-los de forma única. Dessa forma, um objeto é exclusivamente identificável, sendo possível encontrá-lo eficientemente. Contudo, é necessário algum procedimento de se conhecer seu nome único, além de ser requerido manter uma estrutura organizada por endereços. Historicamente, as redes peer-to-peer de estrutura centralizada se baseavam neste método.

Uma outra maneira de organizar esses objetos através da rede é de organizar as conexões entre os diversos peers, que acabam formando redes superpostas (*overlay networks*), que são redes virtuais que figuram sobre as tradicionais redes IP.

Anteriormente ao BitTorrent, nas redes peer-to-peer de estrutura centralizada, o servidor central tentava conhecer a situação de cada um dos peers da rede, enviando mensagens diretamente a eles. Gerenciar muitos peers ao mesmo tempo sobrecarregava o *hardware* do servidor, ou forçava o seu desligamento. Por outro lado, quando a rede tinha estrutura descentralizada, ocorria o fenômeno chamado de inundação de mensagens, no qual estas eram repassadas entre peers intermediários até o peer de destino. Esse repasse excessivo acarretava **sobrecarga de processamento** (*overhead*) nos intermediários, além de gerar mensagens falso-negativas.

Assim, as redes peer-to-peer necessitam de funcionalidades eficientes para que um peer possa entrar e sair do *overlay*, assim como armazenar objetos e encontrá-los (usando endereçamento). Tal eficiência é conseguida através de **DHTs**, onde todas essas funcionalidades dependem da troca de mensagens eficiente entre peers.



A funcionalidade do DHT passou anos sendo utilizada extraoficialmente à especificação do protocolo, quando, enfim, foi adicionada em 2008 [40].

Kademlia

O Kademlia é um DHT criado em 2002 [41] com o objetivo de melhorar os métodos de busca atuais (Napster e Gnutella), que eram ineficientes. Assim como os outros algoritmos de DHT, ele se baseou na estrutura informalmente conhecida como “rede de Plaxton” (*Plaxton mesh*), nome que remete a um dos seus autores [51]. Por ter mostrado bons resultados, foi usado na implementação da busca de arquivos no programa cliente eMule.

A sua modelagem computacional monta um mapa no formato de tabela hash onde IDs de peers ou de torrents são chaves para listas de outros peers.

Modelagem e estrutura de dados

O algoritmo implementa uma rede *overlay* cuja estrutura e comunicação se baseiam na procura de seus nós. Cada um destes nós é identificado por um identificador único (ID), que serve tanto para a identificação quanto para a localização de valores na tabela hash.

Essa tabela hash é na forma de uma árvore binária, cujas folhas são os nós da rede. Cada

folha tem suas posições estabelecidas pelo menor prefixo comum de seus IDs, organizando-os de forma que, para um dado nó x , a árvore é dividida em várias subárvores menores que não o contém. Assim, a maior subárvore consiste de metade da árvore que não contém x , a subárvore seguinte é feita da metade da árvore restante onde x também não está contido, etc. O Kademlia garante ainda que todo nó conhece um outro que está em cada uma das subárvores, se estas contiverem algum nó.



Figura 3.3: Árvore binária do Kademlia. O nó preto é a posição do ID 0011...; os ovais cinzas são as subárvores onde o nó preto deve possuir nós conhecidos. Fonte: [41]

No Kademlia, objetos e nós possuem IDs únicos de 160 bits: enquanto o primeiro utiliza o valor hash de 20 bytes SHA-1 da chave `info_hash` do arquivo .torrent, o segundo é um valor aleatório escolhido pelo próprio programa.

Durante uma busca por peers de um torrent, o processo deve conhecer a chave associado ao objeto, ou seja, o ID, e explora a rede em passos. A cada passo, encontra nós mais próximos da chave, até chegar ou ao valor buscado ou não nós existirem mais próximos que o atual. Dessa forma, para uma rede com n nós, o algoritmo visita apenas $O(\log n)$ nós.

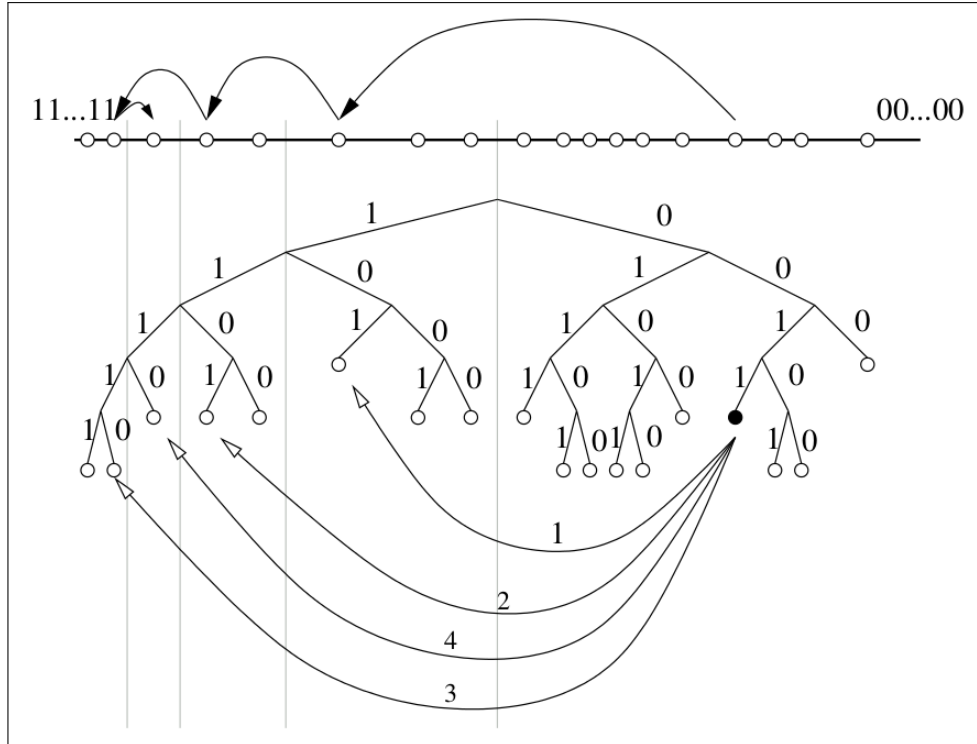


Figura 3.4: Exemplo de uma busca na árvore de nós do Kademlia usando-se um ID. O nó preto, de prefixo 0011, encontra o nó de prefixo 1110 através de sucessivas buscas (setas numeradas inferiores). As setas superiores mostram a convergência da busca durante a execução. Fonte: [41]

Para o conceito de proximidade, as distâncias são calculadas usando-se a função de distância baseada em XOR bit a bit

$$d(x, y) = x \oplus y \quad (3.1)$$

que possui certas propriedades, algumas em comum com a equação de distância euclidiana usual:

- $d(x, x) = 0$
- $x \neq y, d(x, y) > 0$
- simetria: $\forall x, y, d(x, y) = d(y, x)$
- desigualdade triangular: $d(x, y) + d(y, z) \geq d(x, z)$.
Isto vem do fato de $d(x, z) = d(x, y) \oplus d(y, z)$ e que $\forall a \geq 0, \forall b \geq 0 : a + b \geq a \oplus b$
- unidirecionalidade: para um dado ponto x e uma distância $\Delta > 0$, existe exatamente um ponto y tal que $d(x, y) = \Delta$. Isso garante que todas as procuras por uma mesma chave converjam para um mesmo percurso, independente do ponto de partida.
- para um dado x no espaço de números, o conjunto de distâncias Δ entre x e os outros números possui distribuição uniforme, ao contrário da distância euclidiana [47]

No Transmission, o Kademlia é usado a partir de uma biblioteca externa (não mantida pelos seus desenvolvedores) criada por Juliusz Chroboczek [9], sendo adicionada e adaptada para o uso no código do programa. Sua implementação não é na forma de árvore, mas sim de listas ligadas.

```

1  struct node {
2      // elemento de lista ligada de nós DHT
3      (...)
4      unsigned char id[20]; // ID do nó
5      time_t time;          // horário da última mensagem recebida
6      time_t reply_time;    // horário da última resposta recebida corretamente
7      time_t pinged_time;   // horário da última requisição
8      int pinged;           // quantidade de requisições feitas desde a última resposta
9      struct node *next;    // ponteiro para o próximo elemento da lista ligada
10 };
11
12 struct bucket {
13     // elemento de lista ligada de baldes
14     (...)
15     unsigned char first[20]; // ID do primeiro nó do bucket
16     int count;               // quantidade de nós no bucket
17     int time;                // horário da última resposta neste bucket
18     struct sockaddr_storage cached; // endereços de possíveis candidatos
19     int cachedlen;           // tamanho da lista de candidatos
20     struct node *nodes;      // ponteiro para lista ligada de nós
21     struct bucket *next;     // ponteiro para o próximo elemento da lista ligada
22 };

```

Tabela de roteamento

Cada nó do Kademlia armazena informações sobre outros nós para rotear mensagens de pesquisa. Para cada bit i dos IDs (cada ID tem 160 bits) é mantido um k -balde (k -bucket), que contém os nós cuja distância até ele está entre 2^i e 2^{i+1} . Esses k -buckets são listas de endereço IP, porta de comunicação UDP e ID de nós, ordenadas pelo horário da última notícia destes. Para distâncias pequenas, essas listas geralmente serão vazias, enquanto para distâncias maiores poderão ser de tamanho k . Este valor, que é o de replicação do sistema, é escolhido de tal forma que esses k nós possuam grande probabilidade de não falharem na próxima hora.

Quando um nó **A** recebe uma mensagem de outro nó **B**, o balde (*bucket*) correspondente ao ID do remetente (nó **B**) é atualizado. Disto, podem ocorrer as seguintes situações:

- **B** já existe no bucket: passa a ser o primeiro da lista, pois existiu mensagem recente.
- **B** não existe no bucket:
 - bucket não está cheio: **B** é adicionado no começo da lista.
 - bucket cheio: é enviado um *ping* para o nó do final da lista (nó **C**), contatado há mais tempo
 - * **C** não responde ao *ping*: **C** é retirado da lista e **B** é inserido no início
 - * **C** responde ao *ping*: **C** é movido para o início da lista e **B** é ignorado

Por conta disso, ocorre que nós mais antigos e funcionais são preferidos, pois quanto mais tempo um nó está conectado, mais provável ele se manterá conectado por mais 1 hora [56]. Outra vantagem disso é a resistência a alguns ataques de negação de serviço, pois mesmo que ocorra uma inundação de novos nós, estes só seriam inseridos nos k -buckets se os antigos fossem excluídos.

Protocolo

O protocolo de mensagens DHT utiliza o formato KRPC, que é um mecanismo de [chamada de procedimento remoto \(RPC\)](#) que envia dicionários bencode através de UDP, uma única vez por chamada (um pacote para a requisição, outro para a resposta), sem novas tentativas.

Existem 3 tipos de mensagem: consulta (*query*), resposta (*response*) e erro (*error*). Para o protocolo DHT, são 4 comandos *query*: `ping`, `find_node`, `get_peers` e `announce_peer`. Em todos, o nó sempre enviará seu ID como valor da chave `id`.

Uma mensagem KRPC é um dicionário com 2 chaves comuns a todos os 4 comandos: `y`, que especifica o tipo da mensagem, e `t`, que corresponde ao ID da transação. Este é um número binário convertido para string, geralmente formada por 2 caracteres, possuindo valor até 2^{16} , e devolvida nas respostas. Isso permite que estas se relacionem a múltiplas consultas a um nó. Esse ID é chamado de *magic cookie* (cookie mágico) [77].

Cada tipo de mensagem possui formatos diferentes entre si, permitindo parâmetros adicionais para cada chamada, possuindo as seguintes chaves e seus respectivos valores:

- *query*
 - `y`: caractere `q`
 - `q`: string do comando desejado (`ping`, `find_node`, `get_peers`, `announce_peer`)
 - `a`: dicionário contendo parâmetros adicionais, dependendo do comando passado na chave `q`
- *response*
 - `y`: caractere `r`
 - `r`: dicionário contendo valores da resposta, dependendo do comando passado na chave `q`

- *error*

- **y**: caractere **e**
- **e**: lista contendo 2 elementos: código (número inteiro) e mensagem para o erro (string). Os erros podem ser:
 - * 201 (Generic Error): erros genéricos
 - * 202 (Server Error): erros de servidor
 - * 203 (Protocol Error): para pacote mal formado, argumento inválido ou token incorreto
 - * 204 (Method Unknown): comando não conhecido
- exemplo:


```
d1:eli201e23:A Generic Error 0currede1:t2:aa1:y1:ee (bencode)
{"t":"aa", "y":"e", "e":[201, "A Generic Error 0ccurred"]}
```

 (string)

As informações retornadas podem ser sobre peers ou nós DHT: enquanto o primeiro é a “informação compacta de endereço IP/porta” - string de 6 bytes (4 bytes iniciais para o endereço IP e 2 bytes finais para a porta de comunicação usada) -, o segundo é a “informação compacta de nó” - string de 26 bytes (20 bytes iniciais para o ID do nó e 6 bytes finais para a respectiva informação compacta de endereço IP/porta).

Os 4 comandos de *query* do DHT (**ping**, **find_node**, **get_peers** e **announce_peer**) estão definidos da seguinte forma.

ping

É o comando mais simples, que verifica se o nó está online. Possui um único argumento, que é uma chave **id**, que é o ID do nó consultante (na requisição) ou do nó consultado (na resposta).

- formato da requisição

```
d1:ad2:id20:abcdefghij0123456789e1:q4:ping1:t2:aa1:y1:qe (bencode)
{"t":"aa", "y":"q", "q":"ping", "a":{"id":"abcdefghij0123456789"}}
```

 (string)

- formato da resposta

```
d1:rd2:id20:mnopqrstuvwxyz123456e1:t2:aa1:y1:re (bencode)
{"t":"aa", "y":"r", "r":{"id":"mnopqrstuvwxyz123456"}}
```

 (string)

```

1  /* We could use a proper bencoding printer and parser, but the format of DHT messages
2  is fairly stylised, so this seemed simpler. */
3  #define CHECK(offset, delta, size) \
4      if(delta < 0 || offset + delta > size) goto fail \
5
6  #define INC(offset, delta, size) \
7      CHECK(offset, delta, size); \
8      offset += delta \
9
10 #define COPY(buf, offset, src, delta, size) \
11     CHECK(offset, delta, size); \
12     memcpy(buf + offset, src, delta); \
13     offset += delta; \
14
15 #define ADD_V(buf, offset, size) \
16     if(have_v) { \
17         COPY(buf, offset, my_v, sizeof(my_v), size); \
18     }

```

```

1  int send_ping(const struct sockaddr *sa, int salen, const unsigned char *tid, int tid_len) {
2      int i = 0, rc; char buf[512]; // variáveis temporárias
3
4      rc = snprintf(buf + i, 512 - i, "d1:ad2:id20:"); INC(i, rc, 512);
5      COPY(buf, i, myid, 20, 512); // cola o ID do peer, que está em "myid"
6
7      rc = snprintf(buf + i, 512 - i, "e1:q4:ping1:t%d:", tid_len); INC(i, rc, 512);
8      COPY(buf, i, tid, tid_len, 512); // cola o ID do Transmission de 4 bytes para a transação
9      ADD_V(buf, i, 512);
10
11     rc = snprintf(buf + i, 512 - i, "l:y1:qe"); INC(i, rc, 512);
12     return dht_send(buf, i, 0, sa, salen); // envia a requisição de ping
13     (...)
14 }
15
16 int send_pong(const struct sockaddr *sa, int salen, const unsigned char *tid, int tid_len) {
17     int i = 0, rc; char buf[512]; // variáveis temporárias
18
19     rc = snprintf(buf + i, 512 - i, "d1:rd2:id20:"); INC(i, rc, 512);
20     COPY(buf, i, myid, 20, 512); // cola o ID do peer, que está em "myid"
21
22     rc = snprintf(buf + i, 512 - i, "e1:t%d:", tid_len); INC(i, rc, 512);
23     COPY(buf, i, tid, tid_len, 512); // cola o ID do Transmission de 4 bytes para a transação
24     ADD_V(buf, i, 512);
25
26     rc = snprintf(buf + i, 512 - i, "l:y1:re"); INC(i, rc, 512);
27     return dht_send(buf, i, 0, sa, salen); // envia a resposta a um ping
28     (...)
29 }

```

find_node

Este comando, que equivale à mensagem de `FIND_NODE` do artigo do Kademlia [41], é usado para encontrar as informações do nó dado seu ID. Necessita enviar 2 argumentos: a chave `id` e o ID do nó consultante, e a chave `target` e o ID do nó cujas informações o consultante está procurando (ou nó alvo).

- formato dos argumentos da requisição
`{"id": "<IDs dos nós consultantes>", "target": "<ID do nó alvo>"}`
- formato da resposta
`{"id": "<IDs dos nós consultados>", "nodes": "<info compacta do(s) nó(s)>"}`

O nó consultado deve responder com a chave `nodes` contendo uma string com a informação compacta (6 bytes) do nó alvo ou dos k nós bons (que fizeram contato recentemente) mais próximos que estão contidos em sua tabela de roteamento, de 1 ou mais k -buckets. O funcionamento do algoritmo da busca é explicado no trabalho:

O procedimento mais importante que um participante do Kademlia deve realizar é encontrar os k nós próximos a um dado ID de nó. Nós chamamos esse procedimento de “*lookup* de nós”. Kademlia utiliza de um algoritmo recursivo nas buscas por nós. O disparador das buscas começa escolhendo α nós do bucket não-vazio mais próximo (ou, se esse bucket tiver menos que α entradas, utiliza desses α nós mais próximos que conhece). Então, o disparador envia chamadas RPC assíncronas paralelas de comandos **find_node** para esses α nós escolhidos. α é um parâmetro de concorrência geral ao sistema, assumindo valor como 3.

No passo recursivo, o disparador reenvia chamadas a **find_node** para os nós que conheceu das chamadas RPC passadas. (Esta recursão pode começar antes que todos os α nós anteriores tenham respondido). Dos k nós que o disparador concluiu serem mais próximos ao alvo, ele pega α que ainda não foram consultados e envia chamadas RPC **find_node**. Nós que falharem em responder rapidamente são desconsiderados até que respondam. Se uma rodada de comandos **find_node** não retornar algum nó mais próximo do que os nós já conhecidos, o disparador reenvia comandos **find_node** para todos os k nós mais próximos que ainda não foram consultados. O *lookup* termina quando o disparador tiver consultado e obtido respostas de todos os k nós mais próximos conhecidos.

Esse algoritmo remete a um algoritmo de busca em largura onde, a partir de um nó de um grafo, adiciona-se os nós vizinhos em uma fila, e a cada nó que está nela é feita alguma análise dos vizinhos do novo nó, até que a fila se esgote. Porém, enquanto na busca em largura comum se usa uma fila FIFO (*First In First Out*), no algoritmo do Kademlia temos uma fila de prioridade cuja métrica é a distância para o nó de origem, além de uma condição de parada sendo os k nós mais próximos tendo sido avaliados.

Porém, o Transmission implementa essa busca de forma mais flexível e simples. De início, busca o bucket no qual o ID procurado está ou que contém nós mais próximos.


```

1  int send_closest_nodes(const struct sockaddr *sa, int salen, const unsigned char *tid,
2  int tid_len, const unsigned char *id, int want, int af, struct storage *st,
3  const unsigned char *token, int token_len) {
4  unsigned char nodes[8 * 26]; unsigned char nodes6[8 * 38]; // variáveis...
5  int numnodes = 0, numnodes6 = 0; struct bucket *b; // ... temporárias
6
7  (...)
8  b = find_bucket(id, AF_INET); // Busca o bucket provável.
9  if (b) {
10     // Procura por nós no bucket encontrado e nos vizinhos anterior e/ou posterior,
11     // se possuir.
12     numnodes = buffer_closest_nodes(nodes, numnodes, id, b);
13     if (b->next) numnodes = buffer_closest_nodes(nodes, numnodes, id, b->next);
14     b = previous_bucket(b);
15     if (b) numnodes = buffer_closest_nodes(nodes, numnodes, id, b);
16 }
17 (...)
18
19 // Envia os nós encontrados.
20 return send_nodes_peers(sa, salen, tid, tid_len, nodes, numnodes * 26, nodes6,
21 numnodes6 * 38, af, st, token, token_len);
22 }

```

A busca do bucket itera sobre a lista ligada de buckets.

```

1  static int id_cmp(const unsigned char *restrict id1, const unsigned char *restrict id2) {
2  /* Malloc is guaranteed to perform an unsigned comparison. */
3  return memcmp(id1, id2, 20);
4  }
5
6  static struct bucket * find_bucket(unsigned const char *id, int af) {
7  struct bucket *b = af == AF_INET ? buckets : buckets6; // seletor de buckets IPv4/IPv6
8
9  if (b == NULL) return NULL;
10 while (1) {
11     if (b->next == NULL) return b;
12     if (id_cmp(id, b->next->first) < 0) // id tem valor binário menor que b->next->first,
13         return b; // logo deve ficar neste bucket
14     b = b->next;
15 }
16 }

```

Caso retorne o bucket mais provável, efetua buscas internas nele. Se ele possuir elementos vizinhos anteriores ou posteriores, também busca por nós neles.

```

1  static int buffer_closest_nodes(unsigned char *nodes, int numnodes, const unsigned char *id,
2  struct bucket *b) {
3  struct node *n = b->nodes;
4  while (n) { // Itera sobre os nós...
5      if (node_good(n)) // ... e, caso ele tenha feito contato recentemente, ...
6          numnodes = insert_closest_node(nodes, numnodes, id, n); // ...o usa.
7      n = n->next;
8  }
9  return numnodes; // Retorna a quantidade de nós encontrados.
10 }

```

```

1 static int insert_closest_node(unsigned char *nodes, int numnodes, const unsigned char *id,
2 struct node *n) {
3     int i, size; // variáveis temporárias
4
5     (...)
6
7     // Itera sobre os k (neste caso, 8) nós mais próximos atuais (nodes) e verifica
8     // se algum é o nó procurado ou se é mais próximo que algum nó conhecido (n).
9     for (i = 0; i < numnodes; i++) {
10         if (id_cmp(n->id, nodes + size * i) == 0) return numnodes;
11         if (xorcmp(n->id, nodes + size * i, id) < 0) break;
12     }
13
14     // Se tiver iterado por todos os nós próximos atuais, eles são os mais próximos; ...
15     if (i == 8) return numnodes;
16
17     if (numnodes < 8) numnodes++; // ... caso contrário, o nó atual (n) entra para a lista.
18
19     if (i < numnodes - 1) // Abre espaço para o nó atual (n) na lista.
20         memmove(nodes + size * (i + 1), nodes + size * i, size * (numnodes - i - 1));
21
22     (...)
23     // Adiciona o nó atual (n) para a lista, convertendo os dados de endereço IP e porta.
24     struct sockaddr_in *sin = (struct sockaddr_in*) &n->ss;
25     memcpy(nodes + size * i, n->id, 20);
26     memcpy(nodes + size * i + 20, &sin->sin_addr, 4);
27     memcpy(nodes + size * i + 24, &sin->sin_port, 2);
28     (...)
29
30     return numnodes;
31 }

```

Ao fim da busca, envia a lista de nós que encontrou como resposta ao comando de `find_node` recebida. O Transmission também utiliza a função para enviar os nós encontrados pelo comando `get_peers` (pág. 38).

```

1  int send_nodes_peers(const struct sockaddr *sa, int salen, const unsigned char *tid,
2  int tid_len, const unsigned char *nodes, int nodes_len, const unsigned char *nodes6,
3  int nodes6_len, int af, struct storage *st, const unsigned char *token, int token_len) {
4  char buf[2048]; int i = 0, rc, j0, j, k, len; // variáveis temporárias
5
6  rc = snprintf(buf + i, 2048 - i, "d1:rd2:id20:");
7  INC(i, rc, 2048);
8  COPY(buf, i, myid, 20, 2048);
9  if (nodes_len > 0) {
10     // Monta a string bencode dos nós a serem enviados.
11     rc = snprintf(buf + i, 2048 - i, "5:nodes%d:", nodes_len);
12     INC(i, rc, 2048);
13     COPY(buf, i, nodes, nodes_len, 2048);
14 }
15
16 (...)
17
18 // find_node: token_len = 0
19 // get_peers: token_len > 0
20 if (token_len > 0) {
21     // Adiciona o token na resposta da chamada.
22     rc = snprintf(buf + i, 2048 - i, "5:token%d:", token_len);
23     INC(i, rc, 2048);
24     COPY(buf, i, token, token_len, 2048);
25 }
26
27 // find_node: st = NULL
28 // get_peers: st != NULL
29 if (st && st->numpeers > 0) {
30     /* We treat the storage as a circular list, and serve a randomly
31        chosen slice. In order to make sure we fit within 1024 octets,
32        we limit ourselves to 50 peers. */
33
34     len = af == AF_INET ? 4 : 16;
35     j0 = random() % st->numpeers;
36     j = j0;
37     k = 0;
38
39     rc = snprintf(buf + i, 2048 - i, "6:values1");
40     INC(i, rc, 2048);
41     do {
42         if (st->peers[j].len == len) {
43             unsigned short swapped;
44             swapped = htons(st->peers[j].port);
45             rc = snprintf(buf + i, 2048 - i, "%d:", len + 2);
46             INC(i, rc, 2048);
47             COPY(buf, i, st->peers[j].ip, len, 2048);
48             COPY(buf, i, &swapped, 2, 2048);
49             k++;
50         }
51         j = (j + 1) % st->numpeers;
52     } while (j != j0 && k < 50);
53     rc = snprintf(buf + i, 2048 - i, "e");
54     INC(i, rc, 2048);
55 }
56
57 rc = snprintf(buf + i, 2048 - i, "e1:t%d:", tid_len);
58 INC(i, rc, 2048);
59 COPY(buf, i, tid, tid_len, 2048);
60 ADD_V(buf, i, 2048);
61 rc = snprintf(buf + i, 2048 - i, "1:y1:re");
62 INC(i, rc, 2048);
63
64 return dht_send(buf, i, 0, sa, salen);
65 (...)
66 }

```

Um exemplo de requisição e resposta para este comando é

- exemplo de requisição

```
d1:ad2:id20:abcdefghij01234567896:target20:mnopqrstuvwxyz123456e1:q9:find_node1:t2:aa1:y1:qe (bencode)
```

```
{"t":"aa", "y":"q", "q":"find_node", "a":{"id":"abcdefghij0123456789", "target":"mnopqrstuvwxyz123456"}} (string)
```

- exemplo de resposta

```
d1:rd2:id20:0123456789abcdefghij5:nodes9:def456...e1:t2:aa1:y1:re (bencode)
```

```
{"t":"aa", "y":"r", "r":{"id":"0123456789abcdefghij", "nodes":"def456..."}} (string)
```

get_peers

É o comando RPC da mensagem `FIND_VALUE`, serve para buscar peers para um dado um valor hash identificador de arquivo .torrent, enviado como valor da chave `info_hash`, além do ID do nó consultante como valor da chave `id`.

O funcionamento é equivalente ao comando `find_node`, com um detalhe extra: se o nó que recebeu a mensagem possuir peers para o valor hash dado, eles são informados imediatamente na forma compacta (6 bytes para cada peer) numa lista bencode de strings, devolvida como valor da chave `values`. Por outro lado, caso o receptor da mensagem não conhecer nós para o valor hash especificado, a resposta conterá a chave `nodes` com os k nós mais próximos desse valor hash. O nó original consulta outros nós próximos ao arquivo .torrent iterativamente. Ao fim da busca, o programa cliente insere o contato para si mesmo na lista de nós próximos ao arquivo .torrent.

Em ambos os casos, uma chave `token` é informada na resposta, cujo valor é uma string binária curta, que deverá ser utilizada em futuras mensagens de `announce_peer`.

referencio o código do comando anterior?

- formato dos argumentos da requisição

```
{"id": "<IDs dos nós consultantes>", "info_hash": "<hash de 20 bytes do torrent buscado>"}
```

- formato da resposta

```
{"id": "<IDs dos nós consultados>", "token": "<token>", "values": [<info peer 1>, <info peer 2>, ...]}
```

ou

```
{"id": "<IDs dos nós consultados>", "token": "<token>", "nodes": "<info compacta do(s) nó(s)>"}
```

```

1  int dht_periodic(const void *buf, size_t buflen, const struct sockaddr *from, int fromlen,
2  time_t *tosleep, dht_callback *callback, void *closure) {
3  (...)
4  // variáveis temporárias
5  int message;
6  unsigned char tid[16], id[20], info_hash[20], target[20];
7  unsigned char nodes[256], nodes6[1024], token[128];
8  int tid_len = 16, token_len = 128;
9  int nodes_len = 256, nodes6_len = 1024;
10 unsigned short port;
11 unsigned char values[2048], values6[2048];
12 int values_len = 2048, values6_len = 2048;
13 int want;
14
15 (...)
16 // Processa a mensagem recebida e identifica seu tipo.
17 message = parse_message(buf, buflen, tid, &tid_len, id, info_hash, target,
18 &port, token, &token_len, nodes, &nodes_len, nodes6, &nodes6_len,
19 values, &values_len, values6, &values6_len, &want);
20 (...)
21
22 // Realiza os procedimentos conforme o tipo da mensagem recebida.
23 switch (message) {
24 (...)
25 case FIND_NODE:
26     // find_node: só envia os nós mais próximos ao 'target'.
27     debugf("Find node!\n");
28     new_node(id, from, fromlen, 1);
29     debugf("Sending closest nodes (%d).\n", want);
30     send_closest_nodes(from, fromlen, tid, tid_len, target, want, 0, NULL, NULL, 0);
31     break;
32 case GET_PEERS:
33     debugf("Get_peers!\n");
34     new_node(id, from, fromlen, 1);
35     if (id_cmp(info_hash, zeroes) == 0) {
36         (...)
37     }
38     else {
39         struct storage *st = find_storage(info_hash);
40         unsigned char token[TOKEN_SIZE];
41         make_token(from, 0, token);
42         // Se conhecer nós para o hash de torrent dado, os indica na resposta...
43         if (st && st->numpeers > 0) {
44             debugf("Sending found%s peers.\n",
45                 from->sa_family == AF_INET6 ? " IPv6" : "");
46             send_closest_nodes(from, fromlen, tid, tid_len, info_hash, want,
47                 from->sa_family, st, token, TOKEN_SIZE);
48         }
49         else {
50             // ..., senão procura nós próximos, como no comando find_node.
51             debugf("Sending nodes for get_peers.\n");
52             send_closest_nodes(from, fromlen, tid, tid_len, info_hash,
53                 want, 0, NULL, token, TOKEN_SIZE);
54         }
55     }
56     break;
57 (...)
58 }
59 (...)
60 }

```

- exemplo de requisição

```
d1:ad2:id20:abcdefghij01234567899:info_hash20:mnopqrstuvwxyz123456e
1:q9:get_peers1:t2:aa1:y1:qe (bencode)
{"t":"aa", "y":"q", "q":"get_peers",
 "a":{"id":"abcdefghij0123456789", "info_hash":"mnopqrstuvwxyz123456"}}
(string)
```

- exemplo de resposta

- com peers

```
d1:rd2:id20:abcdefghij01234567895:token8:aoeusnth
6:values16:axje.u6:idhtnmee1:t2:aa1:y1:re (bencode)
{"t":"aa", "y":"r", "r":{"id":"abcdefghij0123456789",
 "token":"aoeusnth", "values":["axje.u", "idhtnm"]}} (string)
```

- com nós próximos

```
d1:rd2:id20:abcdefghij01234567895:nodes9:def456...5:token
8:aoeusnthel1:t2:aa1:y1:re (bencode)
{"t":"aa", "y":"r", "r":{"id":"abcdefghij0123456789",
 "token":"aoeusnth", "nodes":"def456..."}} (string)
```

announce_peer

Com este comando, equivalente à mensagem `STORE`, um nó avisa outros que, por ter começado a baixar um arquivo .torrent, entrou no swarm, passando 4 argumentos: o ID do nó consultante como valor da chave `id`, o valor hash identificador do torrent na chave `info_hash`, `port` contém um número inteiro de porta, e o `token` recebido como resposta de uma mensagem `get_peers` anterior. O nó consultado deve verificar que esse token foi enviado anteriormente para o mesmo endereço IP que o nó consultante, e então o nó consultado armazena usando o `info_hash` do torrent como chave e a informação compacta de endereço IP/porta do nó como valor.

Existe ainda mais um argumento, opcional, que é o `implied_port`, cujo valor pode ser 0 ou 1. Se este for 1, o argumento da porta deve ser ignorado e então a fonte de pacotes UDP deve ser usada como a porta do peer. Isso é útil para peers que estão em redes internas a [gateway NAT](#), que podem não saber quais são suas portas externas, e que suportam uTP, aceitando conexões na mesma porta que o DHT.

O token tem papel fundamental para a segurança neste comando, pois serve para prevenir que um peer malicioso registre outros peers para um torrent. No BitTorrent, esse token é a string do valor hash SHA-1 do endereço IP concatenado a uma chave secreta, criada pelo programa cliente, que varia a cada 5 minutos e são aceitos até 10 minutos depois de serem criados.

Para armazenar um valor sob uma chave, um nó busca os k nós mais próximos a ela (usando *lookup* de nós) e envia o comando `announce_peer`.

- formato dos argumentos da requisição

```
{\"id\": \"<IDs dos nós consultantes>\",
  \"implied_port\": <0 ou 1>,
  \"info_hash\": \"<hash de 20 bytes do torrent>\",
  \"port\": <número da porta>,
  \"token\": \"<token>\"}
```

- formato da resposta

```
{\"id\": \"<IDs dos nós consultados>\"}
```

```
1 static int storage_store(const unsigned char *id, const struct sockaddr *sa,
2   unsigned short port) {
3   int i, len; struct storage *st; unsigned char *ip;
4
5   (...)
6   st = find_storage(id); // Encontra o banco para o ID (hash do torrent) dado.
7
8   if (st == NULL) { // Se não existe banco, cria um para o torrent e adiciona na...
9     (...) // ... lista de bancos.
10    st = calloc(1, sizeof(struct storage));
11    (...)
12    memcpy(st->id, id, 20);
13    st->next = storage;
14    storage = st;
15    numstorage++;
16  }
17
18  // Procura pelo peer passado.
19  for (i = 0; i < st->numpeers; i++) {
20    if (st->peers[i].port == port &&
21        st->peers[i].len == len &&
22        memcmp(st->peers[i].ip, ip, len) == 0) break;
23  }
24
25  // Se o peer já existir, só atualiza o momento da última notificação.
26  if (i < st->numpeers) {
27    /* Already there, only need to refresh */
28    st->peers[i].time = now.tv_sec;
29    return 0;
30  }
31  else {
32    struct peer *p;
33    // Se não tiver espaço para o peer novo, expande o banco.
34    if (i >= st->maxpeers) {
35      /* Need to expand the array. */
36      struct peer *new_peers; int n;
37
38      // Não tem mais espaço.
39      if (st->maxpeers >= DHT_MAX_PEERS) return 0;
40
41      // Expande o banco criando 2 espaços ou dobrando de tamanho, limitando em...
42      // ...DHT_MAX_PEERS espaços;
43      n = st->maxpeers == 0 ? 2 : 2 * st->maxpeers;
44      n = MIN(n, DHT_MAX_PEERS);
45      new_peers = realloc(st->peers, n * sizeof(struct peer));
46      if (new_peers == NULL) return -1;
47      st->peers = new_peers; st->maxpeers = n;
48    }
49    // Adiciona o novo peer no banco.
50    p = &st->peers[st->numpeers++];
51    p->time = now.tv_sec; p->len = len;
52    p->port = port; memcpy(p->ip, ip, len);
53    return 1;
54  }
55 }
```

- exemplo de requisição

```
d1:ad2:id20:abcdefghij01234567899:info_hash20:mnopqrstuvwxyz123456
4:porti6881e5:token8:aoeusnth1:q13:announce_peer1:t2:aa1:y1:qe
(bencode)
```

```
{"t":"aa", \"y\":\"q\", \"q\":\"announce_peer\",
 \"a\":{\"id\":\"abcdefghij0123456789\", \"implied_port\": 1,
 \"info_hash\":\"mnopqrstuvwxyz123456\", \"port\": 6881, \"token\": \"aoeusnth\"}}
```

(string)

- exemplo de resposta

```
d1:rd2:id20:mnopqrstuvwxyz123456e1:t2:aa1:y1:re
(bencode)
```

```
{"t":"aa\", \"y\":\"r\", \"r\": {\"id\":\"mnopqrstuvwxyz123456\"}} (string)
```

Entrada e saída na rede e manutenção

Um nó que queira se juntar à rede deve se preparar, numa fase que é chamada de *bootstrap*.

No início dessa fase, o novo nó deve conhecer o endereço IP e a porta de outro nó que já esteja dentro da rede Kademlia, chamado de *bootstrap node* (nó de bootstrap). Caso não esteja, um ID ainda não utilizado é computado e será usado até que esse nó saia da rede.

Feito isso, o novo nó adiciona o nó de bootstrap em um de seus k -buckets e inicia uma consulta `FIND_NODE` em si mesmo. Isso faz com que k -buckets em outros nós sejam populados com o novo ID, assim como os k -buckets do novo nó será preenchido com os nós entre ele e o nó de bootstrap. Em seguida, o novo nó atualiza todos os k -buckets mais distantes que o do nó de bootstrap, para procurar por uma chave aleatória que está nesse intervalo.

Vale notar que os buckets são sempre mantidos atualizados, devido ao grande número de mensagens que viajam pelos nós. Para evitar problemas quando não houver esse tráfego constante, cada nó deverá atualizar um bucket em que não tiver feito um *lookup* de nó na última 1 hora. Assim, deverá escolher um ID aleatório do intervalo correspondente e efetuar uma busca por esse ID.

Uma implementação ingênua pode ser usando-se um vetor de 160 buckets, um para cada possibilidade de diferença de bits. Porém, como no Kademlia a tendência é de se conhecer mais sobre peers mais próximos, muitos buckets ficarão vazios. Uma forma mais otimizada de tratar isso é fazendo com que, inicialmente, nós possuam somente um bucket. Eventualmente, estes ficarão cheios, então sendo divididos. Nesse caso, 2 novos buckets são criados, onde o conteúdo do anterior é dividido entre ambos, e ocorre uma nova tentativa de inserção. Se falhar novamente, o novo contato é descartado. Somente o bucket mais recente é divisível.

Além dessa divisão normal, pode ocorrer de árvores muito desbalanceadas atrapalharem a notificação de um novo nó. Supondo que um nó esteja entrando na rede que já possui mais do que k nós com o mesmo prefixo, estes conseguirão adicioná-lo à suas respectivas tabelas num bucket apropriado, porém o novo nó só conseguirá adicionar k nós à sua tabela. Para evitar isso, os nós mantêm todos os contatos válidos numa subárvore de tamanho $\geq k$, mesmo que deva dividir buckets que não contenham o próprio ID. Assim, quando o novo nó dividir buckets, todos os nós de mesmo prefixo saberão de sua existência.

Do Transmission, vale destacar que quando ele efetua a saída da rede, ele salva a tabela de nós num arquivo `dht.bootstrap`, cujos peers salvos serão utilizados na fase de bootstrap do próximo início do DHT. Caso esse arquivos não exista ou não tenha sido suficiente, o programa utilizará peers fornecidos pelo DHT “oficial”, localizado no endereço `dht.transmissionbt.com:6881`.

Otimizações

As atualizações periódicas de tabelas ocorrem para evitar 2 problemas no *lookup* por chaves válidas: ou nós que receberam anteriormente algum valor a ser guardado naquele chave podem ter saído da rede, ou outros nós novos podem ter entrado na rede possuindo IDs mais próximos à uma chave já armazenada. Em ambos os casos, os nós que possuem aquela entrada de chave-valor devem republicá-la, de forma a garantir que esteja disponível nos k nós mais próximos dessa chave.

Para compensar as saídas de nós, a republicação de cada chave-valor acontece uma vez por hora. Porém, uma implementação ingênua necessitaria muitas mensagens; cada um dos k nós contendo o par de chave-valor executaria um *lookup de nó* seguido de $k - 1$ comandos `announce_peer` por hora. Entretanto, essa implementação pode ser otimizada.

Primeiramente, quando um nó receber um comando `announce_peer` para um par de chave-valor, assumirá que também já foi feito para os outros $k - 1$ nós próximos, não necessitando republicar esse par na próxima hora. Assim, a menos que todos os horários de republicação estejam sincronizados, somente um nó executará a republicação do par de chave-valor.

Segundo, no caso da árvore muito desbalanceadas, nós dividem k -buckets conforme o necessário para garantir o conhecimento total de uma subárvore de tamanho $\geq k$. Se antes de republicar pares de chave-valor, um nó x atualizar todos os k -buckets dessa subárvore de k nós, automaticamente será capaz de descobrir os k nós mais próximos para a chave dada. Para entender o motivo, deve-se considerar 2 casos:

1. se a chave republicada cair no intervalo de ID da subárvore, como a subárvore é de tamanho $\geq k$ e o novo nó já conhece os nós dessa subárvore, então o nó x já conhece os k nós mais próximos à chave.

2. se a chave a ser atualizada estiver fora do intervalo da subárvore, então o nó x , ainda assim, era um dos k nós, ou seja, os k -buckets de x , para os intervalos mais próximos da chave do que a subárvore, têm menos que k nós.

Então, o nó x conhece todos os nós dentro desses k -buckets que, juntamente do conhecimento da subárvore, incluirá os k nós mais próximos à chave.

3.3 Peer Exchange

Outro mecanismo que um peer dentro da rede BitTorrent tem de encontrar outros peers além do DHT é usando o protocolo PEX (*Peer Exchange*, ou troca de peers). Ele permite enviar mensagens periodicamente para outros membros do swarm contendo listas de peers adicionados e peers removidos de sua lista de contatos, agilizando o processo de descobrimento de nós da rede. Apesar disso, não é necessário para o funcionamento de uma transmissão BitTorrent, sendo de uso opcional, onde o usuário do programa cliente pode escolher utilizar esse método habilitando-o em suas configurações.

Este mecanismo faz parte das extensões do protocolo BitTorrent [48], porém ainda não é oficial, mesmo possuindo especificações de mensagem, sua definição mais forte é uma convenção [88].

Existem 2 protocolos de extensão: o *Azureus Messaging Protocol* (AZMP) e o *libtorrent Extension Protocol* (LTEP). Em ambos os casos, a convenção estabelece que:

- no máximo 100 peers podem ser enviados numa mesma mensagem (50 adicionados e 50 removidos).
- uma mensagem de *peer exchange* não deve ser enviada com frequência maior do que 1 minuto.

No Transmission, quando o seu uso é habilitado pelo seu usuário, o programa executa periodicamente uma função de envio de mensagem PEX, quando um peer se conecta a ele ou quando recebe um pedido de mensagem PEX.

```

1 // Envio de mensagem PEX
2 static void sendPex(tr_peerMsgs * msgs) {
3     if (msgs->peerSupportsPex && tr_torrentAllowsPex(msgs->torrent)) {
4         (...)
5         // Pega os 50 nós conectados e que são úteis ao torrent sendo baixado.
6         const int newCount = tr_peerMgrGetPeers(msgs->torrent, &newPex, TR_AF_INET,
7             TR_PEERS_CONNECTED, MAX_PEX_PEER_COUNT);
8         (...)
9
10        /* build the diffs */
11        diffs.added = tr_new(tr_pex, newCount); diffs.addedCount = 0;
12        diffs.dropped = tr_new(tr_pex, msgs->pexCount); diffs.droppedCount = 0;
13        diffs.elements = tr_new(tr_pex, newCount + msgs->pexCount); diffs.elementCount = 0;
14        tr_set_compare(msgs->pex, msgs->pexCount, newPex, tr_pexCompare,
15            sizeof(tr_pex), pexDroppedCb, pexAddedCb, pexElementCb, &diffs);
16        (...)
17
18        // Se tiverem novos peers a serem enviados, o faz.
19        if (diffs.addedCount || diffs.droppedCount || !diffs6.addedCount ||
20            diffs6.droppedCount) {
21            int i; tr_variant val; uint8_t * tmp, *walk; // variáveis temporárias
22            struct evbuffer * payload; // listas de peers do PEX
23            struct evbuffer * out = msgs->outMessages; // mensagem a ser enviada
24
25            /* update peer */
26            (...)
27            msgs->pex = diffs.elements; msgs->pexCount = diffs.elementCount;
28            (...)
29
30            /* build the pex payload */
31            tr_variantInitDict(&val, 3);
32
33            if (diffs.addedCount > 0) {
34                /* "added" */
35                tmp = walk = tr_new(uint8_t, diffs.addedCount * 6);
36                for (i = 0; i < diffs.addedCount; ++i) {
37                    memcpy(walk, &diffs.added[i].addr.addr, 4); walk += 4;
38                    memcpy(walk, &diffs.added[i].port, 2); walk += 2;
39                }
40                (...)
41                tr_variantDictAddRaw(&val, TR_KEY_added, tmp, walk - tmp);
42                (...)
43            }
44            if (diffs.droppedCount > 0) {
45                /* "dropped" */
46                tmp = walk = tr_new(uint8_t, diffs.droppedCount * 6);
47                for (i = 0; i < diffs.droppedCount; ++i) {
48                    memcpy(walk, &diffs.dropped[i].addr.addr, 4); walk += 4;
49                    memcpy(walk, &diffs.dropped[i].port, 2); walk += 2;
50                }
51                (...)
52                tr_variantDictAddRaw(&val, TR_KEY_dropped, tmp, walk - tmp);
53                (...)
54            }
55            (...)
56
57            /* write the pex message */
58            payload = tr_variantToBuf(&val, TR_VARIANT_FMT_BENC);
59            evbuffer_add_uint32(out, 2 * sizeof(uint8_t) + evbuffer_get_length(payload));
60            evbuffer_add_uint8(out, BT_LTEP);
61            evbuffer_add_uint8(out, msgs->ut_pex_id);
62            evbuffer_add_buffer(out, payload);
63            pokeBatchPeriod(msgs, HIGH_PRIORITY_INTERVAL_SECS);
64            dbgmsg(msgs, "sending a pex message; outMessage size is now %zu",
65                evbuffer_get_length(out));
66            (...)
67        }
68        (...)
69    }
70 }

```

Já quando o Transmission recebe uma mensagem PEX, ele só utiliza os endereços de peers adicionados, ignorando os que o peer remetente enviou na lista de removidos.

```
1 // Recebimento e processamento de uma mensagem PEX
2 static void parseUtPex(tr_peerMsgs * msgs, int msglen, struct evbuffer * inbuf) {
3     int loaded = 0; size_t added_len; tr_variant val; const uint8_t * added;
4     uint8_t * tmp = tr_new(uint8_t, msglen);
5     tr_torrent * tor = msgs->torrent;
6
7     tr_peerIoReadBytes(msgs->io, inbuf, tmp, msglen);
8
9     if (tr_torrentAllowsPex(tor) && (
10         (loaded = !tr_variantFromBenc(&val, tmp, msglen)) )) {
11         if (tr_variantDictFindRaw(&val, TR_KEY_added, &added, &added_len)) {
12             tr_pex * pex; size_t i, n; size_t added_f_len = 0; const uint8_t * added_f = NULL;
13
14             tr_variantDictFindRaw(&val, TR_KEY_added_f, &added_f, &added_f_len);
15             pex = tr_peerMgrCompactToPex(added, added_len, added_f, added_f_len, &n);
16
17             n = MIN(n, MAX_PEX_PEER_COUNT);
18             for (i = 0; i < n; ++i) {
19                 (...)
20                 tr_peerMgrAddPex(tor, TR_PEER_FROM_PEX, pex + i, seedProbability);
21             }
22             (...)
23         }
24     }
25     (...)
26 }
```

Tudo bem deixar deste jeito?

3.4 Jogo da troca de arquivos

arrumar uso dos termos choke/unchoke

Nesta altura do processo de download de um torrent, o Transmission já realizou muitos procedimentos. Tudo começou com a adição de um arquivo .torrent ao programa, que leu seus dados e identificou os endereços de trackers; então, entrou em contato com eles, que respondeu com uma lista de peers que já estão no swarm. Ou seja, até agora, não foi baixado nenhum byte sequer do arquivo contido no pacote do torrent.

O BitTorrent trata a troca de dados entre peers como um jogo que, usufruindo das lógicas existentes na teoria dos jogos - área da matemática que estuda modelos matemáticos de conflito e cooperação entre agentes inteligentes que tomam decisões - , tenta se tornar um ambiente na qual se buscará as melhores condições de velocidades de download e upload de arquivos e maior tempo de disponibilidade do torrent na rede. Para isso, existe o conceito *tit-for-tat* (olho por olho), que move o BitTorrent de uma maneira geral: um peer preferirá ajudar peers que o ajudam, ou seja, só fará upload de partes para aqueles que o fizerem de volta.

Nesta seção, mostraremos o protocolo de mensagens para trocas de arquivos entre esses peers dessa lista e os algoritmos do BitTorrent dessas trocas.

Estados dos nós e informações

Existem 2 características independentes que formam as possibilidades de estados que um peer pode assumir enquanto participa de um swarm:

- **choking** (estrangulamento): se um peer **A** estrangulará a conexão com outro peer **B** (*choked*) ou a deixará livre (*unchoked*).
- **interested** (interesse): se um peer **A** terá interesse em um peer **B** (*interested*) ou não (*not interested*)

Uma nova conexão entre peers inicia em *choked* e *not interested* em ambos os sentidos, ou seja, com **A** e **B** estrangulando suas conexões mutuamente e sem interesse no outro. Esses estados ditarão todas as estratégias de troca de partes entre peers.

Outra informação utilizada é o *bitfield*, que é um mapa de bits onde cada bit representa uma parte que o peer já possui.

Mensagens

O protocolo é definido por 12 mensagens e 2 tipos de assinaturas. Essas mensagens são enviadas entre peers e serve para estes tomarem conhecimento da situação de download de um torrent. A primeira assinatura é exclusiva da mensagem de handshake, enquanto todas as outras seguem o mesmo padrão.

handshake

assinatura: `<tam. header><header><bytes reservados><info_hash><peer_id>`

O *handshake* (aperto de mãos) é a primeira mensagem a ser enviada por um peer que recém-chegado à rede.

- `<tam. header>`: tamanho da string `<header>`, representado em binário por 1 byte. O comprimento oficial é 19.
- `<header>`: string identificadora do protocolo. Na versão 1.0 do protocolo BitTorrent, a string oficial é `BitTorrent protocol`.
- `<bytes reservados>`: seção de 8 bytes (= 64 bits) reservados para a habilitação de funcionalidades extras do protocolo. Um e-mail enviado pelo criador do BitTorrent Bram Cohen [90] sugere que os bits menos significativos sejam usados primeiro, para que os mais significativos possam ser usados para alterar o significado dos bits finais. A implementação de cada uma das funcionalidades não-oficiais depende do programa cliente. A tabela abaixo mostra os bits e seus respectivos usos, oficiais (*) ou não-oficiais.

Bit	Uso
1	Azureus Extended Messaging
1-16	BitComet Extension protocol
21	BitTorrent Location-aware Protocol 1.0
44	Extension protocol
47-48	Extension Negotiation Protocol
61	NAT Traversal
62	Fast Peers*
63	XBT Peer Exchange
64	DHT* ou XBT Metadata Exchange

- `<info_hash>`: o ID do torrent, que é a string valor hash de 20 bytes resultante da função de hash SHA-1, com URL encode, do valor da chave `info` do arquivo `.torrent`;

- `<peer_id>`: ID único do cliente, que é uma string de 20 bytes, geralmente sendo o mesmo valor `peer_id` enviado nas requisições ao tracker, prefixado pelas informações como o nome do programa cliente e a sua versão. Por exemplo, o Transmission envia o prefixo `-TR2820-...`

```

1  static bool buildHandshakeMessage(tr_handshake * handshake, uint8_t * buf) {
2      const unsigned char * peer_id = NULL;
3      const uint8_t * torrentHash;
4      tr_torrent * tor;
5
6      // Cria ou recupera o ID do cliente e carrega o ID do torrent.
7      if ((torrentHash = tr_cryptoGetTorrentHash(handshake->crypto)))
8          if ((tor = tr_torrentFindFromHash(handshake->session, torrentHash)))
9              peer_id = tr_torrentGetPeerId(tor);
10
11      (...)
12      uint8_t * str = buf;
13
14      // Começa a montar a string str com a mensagem de handshake
15      // OBS: \023 = valor octal para o correspondente binário de 19
16      // (tamanho da string "BitTorrent protocol")
17      memcpy(str, "\023BitTorrent protocol", 20); str += 20;
18
19      memset(str, 0, HANDSHAKE_FLAGS_LEN); // Zera 8 bytes (bytes reservados).
20
21      HANDSHAKE_SET_LTEP(str); // Habilita LTEP e Fast Peers, setando...
22      HANDSHAKE_SET_FASTTEXT(str); // ... os bits correspondentes.
23
24      /* Note that this doesn't depend on whether the torrent is private.
25       * We don't accept DHT peers for a private torrent,
26       * but we participate in the DHT regardless. */
27      if (tr_dhtEnabled(handshake->session))
28          HANDSHAKE_SET_DHT(str); // Habilita DHT, se o usuário tiver escolhido.
29
30      str += HANDSHAKE_FLAGS_LEN;
31
32      // Adiciona o hash identificador do torrent e o id do cliente
33      memcpy(str, torrentHash, SHA_DIGEST_LENGTH); str += SHA_DIGEST_LENGTH;
34      memcpy(str, peer_id, PEER_ID_LEN); str += PEER_ID_LEN;
35
36      (...)
37      return true;
38  }

```

Essa mensagem é enviada imediatamente pelo peer que inicia uma conexão. O receptor deve responder, assim que ver o ID do torrent na seção de `info_hash` da mensagem, com o seu `peer_id`. A conexão deve ser fechada em dois casos: pelo receptor, se ele receber a mensagem para um ID de torrent desconhecido para si, ou pelo iniciador, caso o `peer_id` recebido como resposta seja diferente daquele indicado na lista de peers recebida do DHT.

keep-alive

keep-alive: `<tamanho=0000>`

A mensagem de *keep-alive* (“mantenha vivo” em português literal) serve para manter uma conexão aberta caso nenhuma outra mensagem seja enviada num período de tempo (geralmente,

2 minutos).

Assim como as outras mensagens, esta mensagem usa a assinatura

`<tamanho><ID da mensagem><dados>`

- `<tamanho>`: valor de 4 bytes em *big endian*
- `<ID da mensagem>`: decimal de 1 byte
- `<dados>`: dados a serem enviados ao outro peer, dependente da mensagem

Porém, não possui ID da mensagem nem dados a serem enviados, possuindo tamanho 0.

```
1  static size_t fillOutputBuffer(tr_peerMsgs * msgs, time_t now) {
2      size_t bytesWritten = 0;
3      (...)
4
5      // KEEPALIVE_INTERVAL_SECS = 100
6      if ((msgs != NULL) && (msgs->clientSentAnythingAt != 0)
7          && ((now - msgs->clientSentAnythingAt) > KEEPALIVE_INTERVAL_SECS)) {
8          dbgmsg(msgs, "sending a keepalive message");
9          evbuffer_add_uint32(msgs->outMessages, 0); // Envia uma mensagem vazia.
10         (...)
11     }
12
13     return bytesWritten;
14 }
```

choke e unchoke

choke: `<tamanho=0001><ID da mensagem=0>`

unchoke: `<tamanho=0001><ID da mensagem=1>`

Estas mensagens servem para indicar que a mudança de estado de choking que o peer remetente tratará o receptor. Ou seja, o receptor *choked* não terá suas requisições atendidas, ao contrário de quando estiver *unchoked*.

```
1  // int choke: 1 ==> choke
2  //             0 ==> unchoke
3  static void protocolSendChoke(tr_peerMsgs * msgs, int choke) {
4      struct evbuffer * out = msgs->outMessages; // buffer de saída
5
6      evbuffer_add_uint32(out, sizeof(uint8_t)); // <tamanho> = 1 byte
7
8      // <ID da mensagem>: BT_CHOKE (0) ou BT_UNCHOKE (1)
9      evbuffer_add_uint8(out, choke ? BT_CHOKE : BT_UNCHOKE);
10
11     dbgmsg(msgs, "sending %s...", choke ? "Choke" : "Unchoke");
12     (...)
13 }
```


interested e not interested

interested: <tamanho=0001><ID da mensagem=2>
not interested: <tamanho=0001><ID da mensagem=3>

Assim como as mensagens de *choke*, estas 2 mensagens também servem para indicar a mudança de estado, desta vez sendo a mudança de interesse que o peer remetente terá no receptor.

```
1 // libtransmission/peer-msgs.c:769
2 //
3 // bool b: true => interested
4 //         false => not interested
5 static void sendInterest(tr_peerMsgs * msgs, bool b) {
6     struct evbuffer * out = msgs->outMessages; // buffer de saída
7     (...)
8     dbgmsg(msgs, "Sending %s", b ? "Interested" : "Not Interested");
9     evbuffer_add_uint32(out, sizeof(uint8_t)); // <tamanho> = 1 byte
10
11     // <ID da mensagem>: BT_INTERESTED (2) ou BT_NOT_INTERESTED (3)
12     evbuffer_add_uint8(out, b ? BT_INTERESTED : BT_NOT_INTERESTED);
13     (...)
14 }
```

have

have: <tamanho=0005><ID da mensagem=4><dados=i-ésima parte>

A definição é que esta mensagem avisa um peer que a *i*-ésima parte do torrent foi baixada pelo remetente e verificada através de valor hash. Porém, não necessariamente é usada dessa forma.

```
1 // libtransmission/peer-msgs.c:399
2 //
3 // uint32_t index: índice i da parte adquirida
4 static void protocolSendHave(tr_peerMsgs * msgs, uint32_t index) {
5     struct evbuffer * out = msgs->outMessages; // buffer de saída
6     (...)
7     evbuffer_add_uint32(out, sizeof(uint8_t) + sizeof(uint32_t)); // <tamanho> = 1B + 4B
8     evbuffer_add_uint8(out, BT_HAVE); // <ID da mensagem> = 4
9     evbuffer_add_uint32(out, index); // <índice> = índice i
10
11     dbgmsg(msgs, "sending Have %u", index);
12     (...)
13 }
```

Uma implementação de algoritmo do jogo da troca pode fazer com que um peer, que acabou de adquirir a parte, não emitir aviso para todos os peers vizinhos que já a possuem, diminuindo o **overhead** de mensagens do protocolo. Por outro lado, enviar esse aviso pode ajudar na determinação de qual parte é mais rara.

bitfield

bitfield: <tamanho=0001+X><ID da mensagem=5><dados=mapa de bits>

O BitTorrent usa bitfields (mapas de bits) em *big endian* para representar na forma de *flags* quais partes de um torrent já possui.

Esta mensagem só deve ser enviada imediatamente após o processo de *handshake* terminar e antes de qualquer outra mensagem. Se o peer não tiver nenhuma parte o campo de dados pode ser omitido. Como bitfields variam de acordo com o tamanho total do torrent, o comprimento da mensagem é variável, onde *X* é o comprimento do bitfield, acrescentado de alguns bits 0 de sobra no seu final (*spare bits*).

Um bitfield de comprimento errado ou sem bits de sobra no final são considerados erros, fazendo com que as respectivas conexões devam ser fechadas.

```
1 static void sendBitfield(tr_peerMsgs * msgs) {  
2     void * bytes; size_t byte_count = 0; // bitfield e seu comprimento  
3     struct evbuffer * out = msgs->outMessages; // buffer de saída  
4  
5     (...)  
6     // Cria o bitfield conforme as partes que possui no momento.  
7     bytes = tr_cpCreatePieceBitfield(&msgs->torrent->completion, &byte_count);  
8  
9     // <tamanho> = 1 + tamanho do bitfield  
10    evbuffer_add_uint32(out, sizeof(uint8_t) + byte_count);  
11  
12    evbuffer_add_uint8(out, BT_BITFIELD); // <ID da mensagem> = 5  
13    evbuffer_add(out, bytes, byte_count); // <dados=mapa de bits>  
14    dbgmsg(msgs, "sending bitfield... outMessage size is now %zu", evbuffer_get_length(out));  
15    (...)  
16 }
```

request

request: <tamanho=0013><ID da mensagem=6><dados=<índice><início><tamanho>>

Conforme foi explicado anteriormente (página 14), um torrent divide os dados em partes, que por sua vez são divididas em blocos, que são os conteúdos trocados entre os peers. Assim, o tamanho das partes é um múltiplo do tamanho dos blocos.

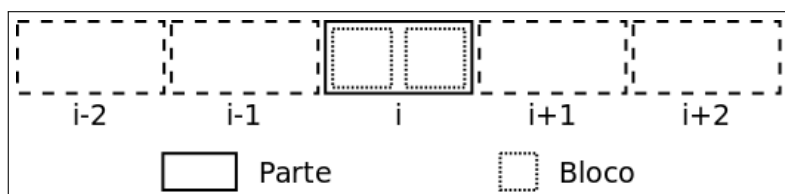


Figura 3.5: trecho da seção de dados do torrent, com as divisões das partes e dos blocos

Com esta mensagem, um peer pede por uma parte do torrent. A seção de dados contém os seguintes números inteiros:



Figura 3.6: parâmetros da mensagem request e seus significados

- índice: índice da parte base i
- início: deslocamento, em bytes, da posição do bloco da parte i pedido
- tamanho: tamanho, em bytes, do bloco pedido

```

1 // libtransmission/peer-msgs.c:356
2 // const struct peer_request * req: requisição a ser cancelada
3 static void protocolSendRequest(tr_peerMsgs * msgs, const struct peer_request * req) {
4     struct evbuffer * out = msgs->outMessages; // buffer de saída
5
6     // sizeof(uint8_t) + 3 * sizeof(uint32_t) = 1B + 3 * 4B = 13B
7     evbuffer_add_uint32(out, sizeof(uint8_t) + 3 * sizeof(uint32_t)); // <tamanho=0013>
8     evbuffer_add_uint8(out, BT_REQUEST); // <ID da mensagem> = 6
9
10    // <dados=<índice><início><tamanho>>
11    evbuffer_add_uint32(out, req->index); // <índice>
12    evbuffer_add_uint32(out, req->offset); // <início>
13    evbuffer_add_uint32(out, req->length); // <tamanho>
14
15    dbgmsg(msgs, "requesting %u:%u->%u...", req->index, req->offset, req->length);
16    (...)
17 }

```

piece

piece: <tamanho=0009+X><ID da mensagem=7><dados=<índice><início><bloco>>

Esta mensagem é a resposta para a requisição de um bloco por meio da mensagem *request*, tendo assinatura análoga com exceção do trecho com os dados do bloco. Assim, um peer envia um bloco de dados a outro.

O tamanho da mensagem é variável, onde X é o tamanho do segmento de dados, já que este pode ter tamanhos diferentes entre mensagens diferentes. A seção de dados possui os seguintes números inteiros:

- índice: índice da parte base i
- início: deslocamento, em bytes, da posição do bloco da parte i pedido
- bloco: conteúdo dos dados do bloco

```

1  static size_t fillOutputBuffer(tr_peerMsgs * msgs, time_t now) {
2      int piece;
3      size_t bytesWritten = 0;
4      struct peer_request req;
5
6      (...)
7
8      /**
9       *** Data Blocks
10      **/
11     if ((tr_peerIoGetWriteBufferSize(msgs->io, now) >= msgs->torrent->blockSize) &&
12         popNextRequest(msgs, &req))
13     { // Quando tiver espaço disponível no buffer de saída e outra requisição de...
14         // ... bloco a ser atendida na fila.
15         (...)
16
17         if (requestIsValid(msgs, &req)
18             && tr_cpPieceIsComplete(&msgs->torrent->completion, req.index))
19         { // Se a requisição tiver enviado parâmetros corretos e se o Transmission...
20             // ... já tiver sido baixado-a.
21             int err; struct evbuffer * out;
22             struct evbuffer_iovec iovec[1];
23             const uint32_t msglen = 4 + 1 + 4 + 4 + req.length;
24
25             out = evbuffer_new();
26             evbuffer_expand(out, msglen);
27
28             // <tamanho=0009+X>
29             // sizeof(uint8_t) + 2 * sizeof(uint32_t) = 1B + 2 * 4B = 9B
30             evbuffer_add_uint32(out, sizeof(uint8_t) + 2 * sizeof(uint32_t) + req.length);
31             evbuffer_add_uint8(out, BT_PIECE); // <ID da mensagem=7>
32
33             // <dados=<índice><início><bloco>
34             evbuffer_add_uint32(out, req.index); // <índice>
35             evbuffer_add_uint32(out, req.offset); // <início>
36
37             // <início>
38             evbuffer_reserve_space(out, req.length, iovec, 1);
39             err = tr_cacheReadBlock(getSession(msgs)->cache, msgs->torrent, req.index,
40                                   req.offset, req.length, iovec[0].iov_base); // conteúdo de dados do bloco
41             iovec[0].iov_len = req.length;
42             evbuffer_commit_space(out, iovec, 1);
43
44             (...)
45         }
46         (...)
47     }
48     (...)
49     return bytesWritten;
50 }
51

```

cancel

cancel: <tamanho=0013><ID da mensagem=8><dados=<índice><início><tamanho>>

A mensagem *cancel* serve para cancelar a mensagem *request* de mesmos parâmetros. É mais utilizada durante o algoritmo de fim de jogo.

referenciar seção do algoritmo do fim de jogo

```
1 // libtransmission/peer-msgs.c:372
2 // const struct peer_request * req: requisição a ser cancelada
3 static void protocolSendCancel(tr_peerMsgs * msgs, const struct peer_request * req) {
4     struct evbuffer * out = msgs->outMessages; // buffer de saída
5
6     // sizeof(uint8_t) + 3 * sizeof(uint32_t) = 1B + 3 * 4B = 13B
7     evbuffer_add_uint32(out, sizeof(uint8_t) + 3 * sizeof(uint32_t)); // <tamanho=0013>
8     evbuffer_add_uint8(out, BT_CANCEL); // <ID da mensagem> = 8
9
10    // <dados=<índice><início><tamanho>>
11    evbuffer_add_uint32(out, req->index); // <índice>
12    evbuffer_add_uint32(out, req->offset); // <início>
13    evbuffer_add_uint32(out, req->length); // <tamanho>
14
15    dbgmsg(msgs, "cancelling %u:%u->%u...", req->index, req->offset, req->length);
16    (...)
17 }
```

port

cancel: <tamanho=0003><ID da mensagem=9><dados=porta>

Para os casos do peer estar usando a função de DHT, esta mensagem serve para avisar ao receptor qual a porta de comunicação TCP que o remetente está usando para receber mensagens de DHT; Com isso, é adicionado à tabela de roteamento do receptor.

```
1 // libtransmission/peer-msgs.c:387
2 // uint16_t port: número da porta a ser utilizada
3 static void protocolSendPort(tr_peerMsgs * msgs, uint16_t port) {
4     struct evbuffer * out = msgs->outMessages; // buffer de saída
5
6     dbgmsg(msgs, "sending Port %u", port);
7     evbuffer_add_uint32(out, 3); // <tamanho=0003>
8     evbuffer_add_uint8(out, BT_PORT); // <ID da mensagem=9>
9     evbuffer_add_uint16(out, port); // <dados=porta>
10 }
```

Algoritmos de seleção de partes

Por se tratar de um protocolo de troca de dados em partes, uma escolha ruim de quais destas se adquirir primeiro faz com que seja grande a possibilidade de um peer baixar alguma parte que seja sempre ofertada por outros peers [11]. Isso acarretará em não se ter nenhuma das partes que se deseja. Assim, durante a troca das partes, um peer adota estratégias diferentes para fornecer e receber blocos de dados, com o intuito de tentar otimizar a obtenção do conjunto total de dados dos torrents e ajudando a difundir o conteúdo deste ao resto do swarm.

As estratégias a seguir são medidas que um peer BitTorrent pode adotar com efeitos em si mesmo, sem afetar outros peers.

Random First Piece

No início do download de um torrent, um peer não possui partes. Para que comece a receber partes, ele avisa que é recém-chegado, e assim algum membro do swarm envia-lhe uma parte comum aleatoriamente. Dessa forma, ele possuirá uma “moeda de troca”, podendo então ajudar outros peers e, assim, conseguirá melhores condições de ser atendido. É importante que a parte seja comum, pois assim será possível conseguir blocos de locais diferentes; se fosse rara, seria mais difícil conseguir completá-la.

Após completar a primeira parte, o algoritmo passa para a estratégia de *Rarest First*.

Rarest First

Nesta fase, o peer passa a pedir as partes mais raras antes. Para isso, utiliza os bitfields recebidos dos outros peers, mantendo-os atualizados a cada mensagem `have` que recebe. Feito isso, das partes que são menos frequentes nos bitfields, escolhe uma aleatoriamente, devido ao fato de que uma parte rara poderia ser muito requisitada, o que seria improdutivo. Da mesma forma, a estratégia de deixar partes mais comuns para serem baixadas mais tarde não é prejudicial, pois a probabilidade de que um peer que está disponibilizando essas partes num dado momento deixará de ser interessante é reduzida.

É fácil ver que, enquanto um seeder não enviar todas as partes do torrent que está fornecendo, não haverá nenhum peer que possa ter terminado de baixá-lo. Assim, quando o seeder possuir capacidade de upload menor do que de seus leechers, a melhor situação será se cada um destes baixar partes diferentes dos outros, que maximiza o espalhamento dos dados e alivia a carga sobre o seeder, justificando a prioridade em se fazer download das partes raras.

Em uma outra situação, o seeder pode sair da rede, tornando os leechers responsáveis pela

distribuição do torrent. Assim, corre-se o risco de alguma parte se tornar indisponível. O *rarest first* também ajuda neste caso, pois replica as partes raras o mais rápido possível, reduzindo o risco do torrent se tornar incompleto.

Strict Priority

Cada parte é composta de blocos, que são os pacotes de dados trocados entre peers. Esta política de prioridade faz com que, se um bloco for requisitado, o restante dos blocos da mesma parte serão pedidos antes dos blocos de outras partes, a fim de se ter partes completas o mais rápido possível.

Endgame Mode

Próximo ao fim do download de um torrent, a tendência é que os últimos blocos de dados demorem, chegando aos poucos. Para agilizar isso, o cliente pede todos os blocos faltantes para todos os peers, enviando uma mensagem `cancel` para todos que não tiverem respondido à requisição assim que o bloco é recebido para evitar desperdício de banda de rede na recepção redundante de dados.

Implementação do Transmission

O Transmission implementa as partes desejadas e os blocos do torrent a serem baixados como vetores. O vetor de partes é ordenado, utilizando seus campos auxiliares, de forma que a parte mais importante que se deseja receber será requisitada antes. Ele é usado para decidir quais blocos serão requisitados.

```
1 // elemento do vetor de partes desejadas
2 struct weighted_piece {
3     tr_piece_index_t index; // índice i da parte
4     int16_t salt;           // valor aleatório entre 0 e 4096
5     int16_t requestCount;   // qtd de requisições feitas
6 };
```

Conforme são recebidas mensagens `have` e `bitfield` de outros peers, ocorre o processamento da carga de dados dessas mensagens e a atualização das informações das partes de cada remetente. Assim, o Transmission estima quais partes do torrent são mais raras que outras, guardando essa informação no vetor de replicações.

```

1 // Função de callback para recepção de mensagens de peers.
2 static void peerCallbackFunc(tr_peer * peer, const tr_peer_event * e, void * vs) {
3     tr_swarm * s = vs;
4
5     swarmLock(s);
6     switch (e->eventType) {
7         (...)
8         case TR_PEER_CLIENT_GOT_HAVE: // peer avisou que tem uma parte
9             if (replicationExists(s)) {
10                 tr_incrReplicationOfPiece(s, e->pieceIndex); // replicação para uma parte
11                 (...)
12             }
13             break;
14
15         case TR_PEER_CLIENT_GOT_HAVE_ALL: // peer avisou que tem todas as partes
16             if (replicationExists(s)) {
17                 tr_incrReplication(s); // replicação para todas as partes
18                 (...)
19             }
20             break;
21
22         (...)
23
24         case TR_PEER_CLIENT_GOT_BITFIELD: // peer enviou bitfield
25             (...)
26             if (replicationExists(s)) {
27                 tr_incrReplicationFromBitfield(s, e->bitfield); // replicação usando bitfield
28                 (...)
29             }
30             break;
31         (...)
32     }
33     swarmUnlock(s);
34 }

```

Essa informação é utilizada na reordenação desse vetor, utilizando a função da linguagem C `qsort()`, que recebe em um dos parâmetros uma função comparadora. A função do Transmission considera, nesta ordem:

1. *strict policy*: completude de uma parte
2. prioridade (assinalada manualmente pelo usuário): prioridade do usuário
3. *rarest first*: raridade da parte
4. *random first piece*: aleatoriedade


```

1 // Função que compara duas partes e calcula a importância de uma sobre a outra.
2 static int comparePieceByWeight(const void * va, const void * vb) {
3     const struct weighted_piece * a = va;
4     const struct weighted_piece * b = vb;
5     int ia, ib, missing, pending;
6     const tr_torrent * tor = weightTorrent;
7     const uint16_t * rep = weightReplication;
8
9     // 1o fator: peso (strict policy)
10    // Uma parte que possui muitos blocos faltando e alguns pedidos tem prioridade menor...
11    // ... do que uma parte que está próxima de terminar.
12    missing = tr_cpMissingBlocksInPiece(&tor->completion, a->index);
13    pending = a->requestCount;
14    ia = missing > pending ?
15        missing - pending : (tor->blockCountInPiece + pending);
16    missing = tr_cpMissingBlocksInPiece(&tor->completion, b->index);
17    pending = b->requestCount;
18    ib = missing > pending ?
19        missing - pending : (tor->blockCountInPiece + pending);
20    if (ia < ib) return -1;
21    if (ia > ib) return 1;
22
23    // 2o fator: prioridade
24    // O usuário pode ter assinalado para o Transmission que um dos arquivos do torrent...
25    // ... deve ser baixado com prioridade maior do que outros.
26    ia = tor->info.pieces[a->index].priority;
27    ib = tor->info.pieces[b->index].priority;
28    if (ia > ib) return -1;
29    if (ia < ib) return 1;
30
31    // 3o fator: raridade (rarest first)
32    // Se um elemento da lista de partes tiver menos réplicas do que outro, então é...
33    // ... mais raro, logo, tem maior prioridade.
34    ia = rep[a->index];
35    ib = rep[b->index];
36    if (ia < ib) return -1;
37    if (ia > ib) return 1;
38
39    // 4o fator: aleatoriedade (random first piece)
40    // Isso afeta o início do download, quanto não se possui nenhuma parte e os peers...
41    // ... vizinhos ainda não são bem conhecidos. Nesse caso, a aleatoriedade está no...
42    // ... salt (valor aleatório entre 0 e 4096).
43    if (a->salt < b->salt) return -1;
44    if (a->salt > b->salt) return 1;
45
46    return 0; // partes podem ser consideradas de prioridade igual
47 }

```

Já o vetor de blocos mantém a lista dos blocos pedidos com o momento da requisição e para quem o foi feito. É usada para cancelar requisições que ficaram pendentes por muito tempo ou para evitar requisições duplicadas antes do modo de fim de jogo.

```

1 // vetor de pedidos de blocos
2 struct block_request {
3     tr_block_index_t block;
4     tr_peer * peer;
5     time_t sentAt;
6 };

```

Ambos os vetores são armazenados em uma estrutura que guarda essas e outras informações sobre o swarm do torrent que está sendo baixado.

```

1 // Estrutura de informações sobre o swarm de um torrent.
2 typedef struct tr_swarm {
3     (...)
4     tr_ptrArray peers; // vetor de objetos peer
5     tr_torrent * tor;  // informações sobre o torrent
6
7     struct block_request * requests; // vetor de pedidos de blocos
8     int requestCount;                // tamanho do vetor de pedidos de blocos
9
10    struct weighted_piece * pieces;   // vetor de partes desejadas
11    int pieceCount;                   // tamanho do vetor de partes
12
13    // estado do vetor de partes: {não ordenado, por índice, por importância}
14    enum piece_sort_state pieceSortState;
15
16    uint16_t * pieceReplication; // vetor sobre quantos peers possuem cada parte
17    size_t pieceReplicationSize; // tamanho do vetor de índices de partes comuns
18
19    // indicador do modo de fim de jogo
20    // - antes do fim de jogo: endgame = 0
21    // - durante o fim de jogo: endgame = média de qtd de requisições pendentes por peer
22    // Somente peers "rápidos" serão utilizados no fim de jogo.
23    int endgame;
24 } tr_swarm;

```

Eventualmente, o Transmission verifica a necessidade e a capacidade de enviar pedidos de blocos. Caso seja possível, realiza o processamento da lista de partes, montando uma lista de blocos a serem requisitados.

```

1 // Função que cria a lista de blocos para serem requisitados a partir da lista de partes.
2 void tr_peerMgrGetNextRequests(tr_torrent * tor, tr_peer * peer, int numwant,
3     tr_block_index_t * setme, int * numgot, bool get_intervals)
4 {
5     int i, got = 0;
6     struct weighted_piece * pieces;
7     tr_swarm * s = tor->swarm;
8     const tr_bitfield * const have = &peer->have;
9     (...)
10
11     // Se não existir uma lista de partes, cria uma já ordenada
12     if (s->pieces == NULL) pieceListRebuild(s);
13
14     // Caso já exista a lista de partes, verifica o estado da ordenação por importância.
15     if (s->pieceSortState != PIECES_SORTED_BY_WEIGHT)
16         pieceListSort(s, PIECES_SORTED_BY_WEIGHT);
17
18     (...)
19     // Confirma o estado do torrent, verificando se está no modo de fim de jogo.
20     updateEndgame(s); pieces = s->pieces;
21
22     // Para cada parte que falta ser baixada, ...
23     for (i = 0; i < s->pieceCount && got < numwant; ++i) {
24         struct weighted_piece * p = pieces + i;
25
26         // ... se o peer tiver a parte desejada, ....
27         if (tr_bitfieldHas(have, p->index)) {
28             tr_block_index_t b, first, last;
29             tr_ptrArray peerArr = TR_PTR_ARRAY_INIT;
30
31             // ... calcula o intervalo de bytes da parte.
32             tr_torGetPieceBlockRange(tor, p->index, &first, &last);
33
34             // Dentro do intervalo de bytes da parte:
35             for (b = first; b <= last && (...)) {
36                 int peerCount; tr_peer ** peers;
37
38                 // Ignore blocos já possuídos.
39                 if (tr_cpBlockIsComplete(&tor->completion, b)) continue;
40
41                 // Encontra peers novos para se requisitar o bloco,...
42                 // ... juntando às antigas requisições para ele.
43                 tr_ptrArrayClear(&peerArr);
44                 getBlockRequestPeers(s, b, &peerArr);
45                 peers = (tr_peer **) tr_ptrArrayPeek(&peerArr, &peerCount);
46                 if (peerCount != 0) {
47                     // Não faça uma 2ª requisição antes do fim do jogo.
48                     if (!s->endgame) continue;
49
50                     // Não requisiute o bloco para mais de 1 peer simultaneamente.
51                     if (peerCount > 1) continue;
52
53                     // Não refaça a requisição para o mesmo peer.
54                     if (peer == peers[0]) continue;
55
56                     // No modo de fim de jogo, permita requisitar para um 2º peer...
57                     // ... somente se este estiver atendendo pedidos rapidamente.
58                     if (peer->pendingReqsToPeer + numwant - got < s->endgame) continue;
59                 }
60                 (...)
61
62                 // Atualiza a lista de blocos pedidos.
63                 requestListAdd(s, b, peer);
64                 ++p->requestCount;
65             }
66             (...)
67         }
68     }
69     (...)
70 }

```

Tendo criado a lista de blocos, os requisita.

```
1 // Verifica a possibilidade de comunicação com um dado peer e envia requisições de blocos
2 static void updateBlockRequests(tr_peerMsgs * msgs) {
3     (...)
4     blocks = tr_new(tr_block_index_t, numwant); // Cria um vetor de blocos.
5
6     // Monta a lista de quais blocos pedir.
7     tr_peerMgrGetNextRequests(msgs->torrent, &msgs->peer, numwant, blocks, &n, false);
8
9     for (i = 0; i < n; ++i) {
10         struct peer_request req;
11         blockToReq(msgs->torrent, blocks[i], &req); // Cria uma requisição para um bloco.
12         protocolSendRequest(msgs, &req);           // Envia uma mensagem de request.
13     }
14 }
```

Durante a execução da função de criação da lista de blocos a serem pedidos, existe a verificação da completude do torrent, onde se percebe que a quantidade de bytes requisitados já é maior do que a quantidade de bytes que faltam ser baixados. Eis que o modo de fim de jogo se inicia, quando é calculado a capacidade de um novo peer receber e atender a nova requisição de bloco de forma melhor ou igual à taxa que atuais peers estão fazendo.

```
1 //libtransmission/peer-mgr.c:878
2 static bool testForEndgame(const tr_swarm * s) {
3     // Modo de fim de jogo acontece quando, em um dado momento:
4     // qtd de bytes requisitados \geq qtd de bytes para completar o torrent
5     return (s->requestCount * s->tor->blockSize) >= tr_cpLeftUntilDone(&s->tor->completion);
6 }
```

```
1 //libtransmission/peer-mgr.c:887
2 static void updateEndgame(tr_swarm * s) {
3     (...)
4     if (!testForEndgame(s)) { /* not in endgame */
5         s->endgame = 0;
6     }
7     else if (!s->endgame) { /* only recalculate when endgame first begins */
8         int i, numDownloading = 0;
9         const int n = tr_ptrArraySize(&s->peers);
10
11         // Soma os peers BitTorrent ativos...
12         for (i = 0; i < n; ++i) {
13             const tr_peer * p = tr_ptrArrayNth(&s->peers, i);
14             if (p->pendingReqsToPeer > 0) ++numDownloading;
15         }
16
17         // ... com os possíveis seeders web.
18         numDownloading += countActiveWebseeds(s);
19
20         // Média instantânea de requisições pendentes por peers fornecedores
21         s->endgame = s->requestCount / MAX(numDownloading, 1);
22     }
23 }
```

Algoritmos de enforcamento

Além das estratégias mostradas, existem as estratégias que são as formas com que um peer se relacionará com seus vizinhos. Cada um é responsável por melhorar as suas taxas de download e, para isso, baixam partes de quem eles conseguem e escolhem se para quais enviará outras, de forma a mostrar cooperação. Caso a escolha seja de não cooperar, um peer enforca (*choke*) outro, que implica num cancelamento temporário do envio de partes para ele. O recebimento de partes continua normalmente e a conexão não precisa ser rediscutida quando o enforcamento terminar.

O choking não faz parte do protocolo de mensagens, mas é necessário para a boa performance do protocolo. Um bom algoritmo de choking, que utilize de todos os recursos disponíveis, traz boas condições para todos os peers cooperativos e é resistente contra aqueles que só fazem download.

Eficiência de Pareto

Sistemas *pareto eficientes* [82] são aqueles onde duas entidades não podem fazer trocas e ficar mais “felizes” que antes. Em termos da Ciência da Computação, buscar a eficiência de Pareto é usar um algoritmo de otimização local onde as entidades procuram meios de melhorar mutuamente, que convergem a uma situação ótima global. No contexto do BitTorrent, se 2 peers não estão tendo vantagem recíproca por enviar partes, eles podem começar a trocar partes entre si e conseguir taxas de download melhores do que as anteriores.

Algoritmo de choking

Cada peer sempre fará unchoke de 4 peers (geralmente), então o problema passa ser escolher quais deles fazer, deixando que o TCP controle de congestionamento de banda. Essa escolha é baseada estritamente na taxa de download, calculando a média dessa taxa durante 20 segundos. Antigamente, esse cálculo era feito sobre quantidades transferidas a longos prazos, mas notou-se que era uma medida fraca por causa das variações da largura da banda de rede.

Para evitar que recursos sejam desperdiçados pelo rápido choke e unchoke de peers, um cálculo é realizado a cada 10 segundos a fim de saber quem sofrerá o choke, deixando a situação atual como está até o próximo intervalo acabar. Esse tempo é suficiente para que o TCP acelere transferências até sua capacidade total.

Optimistic Unchoking

Fazer upload para peers que possuem as melhores taxas de download sofrem do fato de não possuir meios de descoberta de conexões melhores. Para tentar contornar esse problema, um peer será escolhido para ser alvo de um “unchoke otimista”, que é um unchoke independentemente da taxa de download que ele provê.

A escolha do peer ocorre a cada 30 segundos (no terceiro ciclo de choke), que é tempo suficiente para o upload chegar à velocidade máxima, um download ocorrer como recompensa e chegar à sua velocidade máxima.

Este algoritmo tem como objetivo demonstrar cooperação, sendo equivalente a demonstrar colaboração como iniciativa num caso de jogo do dilema do prisioneiro.

Anti-snubbing

Eventualmente, um peer sofrerá choke de todos os peers de onde estava fazendo download. Nesses casos, terá taxas de download ruins até que um unchoke otimista seja executado.

Quando ficar mais de 1 minuto sem ter recebido nenhuma parte de outro peer, o peer que não recebeu nada perceberá que foi censurado (*snubbed*), e retaliará deixando de enviar partes para ele a menos que tenha sido por sorteio de um unchoke otimista. Dessa forma, vários unchokes otimistas serão executados simultaneamente, que implicará na recuperação das taxas de download.

Upload Only

Uma vez que um peer terminou de baixar os dados do torrent, ele não precisa mais escolher para quem fará o upload usando sua taxa de download. Ao invés disso, passará a enviar dados para os peers com quem possui as maiores taxas de upload, que são os que não estão recebendo muitos dados, então utilizando toda a sua capacidade de upload.

Implementação do Transmission

Quando um torrent é executado e em intervalos de 10 segundos, o Transmission executa uma função que faz os chokes de upload e também verifica os downloads.

```

1 static void rechokePulse(int foo UNUSED, short bar UNUSED, void * vmgr) {
2     tr_torrent * tor = NULL;
3     tr_peerMgr * mgr = vmgr;
4     const uint64_t now = tr_time_msec();
5
6     managerLock(mgr);
7     while ((tor = tr_torrentNext(mgr->session, tor))) {
8         if (tor->isRunning) {
9             tr_swarm * s = tor->swarm;
10
11             if (s->stats.peerCount > 0) {
12                 rechokeUploads(s, now);
13                 rechokeDownloads(s);
14             }
15         }
16     }
17     tr_timerAddMsec(mgr->rechokeTimer, RECHOKE_PERIOD_MSEC);
18     managerUnlock(mgr);
19 }

```

Primeiro, verifica os uploads. A cada execução da função, verifica se será um turno de unchoke otimista, que dura 4 execuções. Além disso, o vetor de peers é usado para o carregamento de um vetor temporário de informações para choke sobre cada um deles.

```

1 static void rechokeUploads(tr_swarm * s, const uint64_t now) {
2     int i, size, unchokedInterested;
3     const int peerCount = tr_ptrArraySize(&s->peers);
4     tr_peer ** peers = (tr_peer**) tr_ptrArrayBase(&s->peers);
5     struct ChokeData * choke = tr_new0(struct ChokeData, peerCount);
6     const tr_session * session = s->manager->session;
7     const int chokeAll = !tr_torrentIsPieceTransferAllowed(s->tor, TR_CLIENT_TO_PEER);
8     const bool isMaxedOut = isBandwidthMaxedOut(&s->tor->bandwidth, now, TR_UP);
9
10    // Confere se existe um peer que está unchoked de forma otimista
11    // (unchoke otimista dura por 4 execuções da função)
12    if (s->optimisticUnchokeTimeScaler > 0) s->optimisticUnchokeTimeScaler--;
13    else s->optimistic = NULL;
14
15    // Ordena os peers por preferência e rate
16    for (i = 0, size = 0; i < peerCount; ++i) {
17        tr_peer * peer = peers[i];
18        tr_peerMsgs * msgs = PEER_MSGS(peer);
19        (...)
20
21        if (tr_peerIsSeed(peer)) {
22            // Choke se peer é seeder total ou parcial (não tem o que baixar,...
23            // ... só envia dados).
24            tr_peerMsgsSetChoke(PEER_MSGS(peer), true);
25        }
26        else if (chokeAll) { // Choke todos se não está fazendo upload.
27            tr_peerMsgsSetChoke(PEER_MSGS(peer), true);
28        }
29        else if (msgs != s->optimistic) { // Se o peer ainda não estiver no modo otimista.
30            struct ChokeData * n = &choke[size++];
31            n->msgs = msgs;
32            n->isInterested = tr_peerMsgsIsPeerInterested(msgs);
33            n->wasChoked = tr_peerMsgsIsPeerChoked(msgs);
34            n->rate = getRate(s->tor, atom, now);
35            n->salt = tr_cryptoWeakRandInt(INT_MAX); // valor aleatório entre 0 e INT_MAX
36            n->isChoked = true;
37        }
38    }
39
40    qsort(choke, size, sizeof(struct ChokeData), compareChoke);
41    (...)

```

A ordenação é feita com `qsort()` usando uma função comparadora que classifica os peers considerando, nesta ordem:

1. velocidade de transferência de dados (download e upload)
2. estado de choke: preferência por unchoked
3. aleatoriedade

```
1 static int compareChoke(const void * va, const void * vb) {  
2     const struct ChokeData * a = va;  
3     const struct ChokeData * b = vb;  
4  
5     if (a->rate != b->rate) /* prefer higher overall speeds */  
6         return a->rate > b->rate ? -1 : 1;  
7  
8     if (a->wasChoked != b->wasChoked) /* prefer unchoked */  
9         return a->wasChoked ? 1 : -1;  
10  
11    if (a->salt != b->salt) /* random order */  
12        return a->salt - b->salt;  
13  
14    return 0;  
15 }
```

Então, trata o vetor temporário. Os melhores peers mudarão seu estado para unchoked, enquanto o resto será choked. Além disso, no caso de uma rodada de unchoke otimista, sorteia um dos peers para ter a sua vez.


```

1      (...)
2      /**
3       * (...)
4       * Peers which have a better upload rate (as compared to the downloaders) but
5       * aren't interested get unchoked. If they become interested, the downloader with
6       * the worst upload rate gets choked. If a client has a complete file, it uses its
7       * upload rate rather than its download rate to decide which peers to unchoke.
8       * (...)
9       */
10
11     // O vetor choke está ordenado do peer mais interessante para o menos interessante.
12     // Sendo N = session->uploadSlotsPerTorrent, que é a qtd de peers que receberão
13     // uploads simultaneamente, N peers serão unchoked. Se já tiver chegado ao limite
14     // máximo, ele mantém o estado anterior.
15     unchokedInterested = 0;
16     for (i = 0; i < size && unchokedInterested < session->uploadSlotsPerTorrent; ++i) {
17         choke[i].isChoked = isMaxedOut ? choke[i].wasChoked : false;
18         if (choke[i].isInterested) ++unchokedInterested;
19     }
20
21     // Se ainda existirem peers na lista e nenhum no modo otimista, monta o sorteio para
22     // escolher um.
23     if (!s->optimistic && !isMaxedOut && (i < size)) {
24         int n; struct ChokeData * c;
25         tr_ptrArray randPool = TR_PTR_ARRAY_INIT;
26
27         for (; i < size; ++i) {
28             if (choke[i].isInterested) {
29                 const tr_peerMsgs * msgs = choke[i].msgs;
30                 int x = 1, y;
31                 if (isNew(msgs)) x *= 3; // peers novos têm mais chance no sorteio
32                 for (y = 0; y < x; ++y)
33                     tr_ptrArrayAppend(&randPool, &choke[i]);
34             }
35         }
36
37         // Sorteia um peer aleatoriamente para o unchoke otimista.
38         if ((n = tr_ptrArraySize(&randPool))) {
39             c = tr_ptrArrayNth(&randPool, tr_cryptoWeakRandInt(n));
40             c->isChoked = false;
41             s->optimistic = c->msgs;
42             s->optimisticUnchokeTimeScaler = OPTIMISTIC_UNCHOKES_MULTPLIER;
43         }
44         (...)
45     }
46
47     // Envia mensagens choke.
48     for (i = 0; i < size; ++i)
49         tr_peerMsgsSetChoke(choke[i].msgs, choke[i].isChoked);
50     (...)
51 }

```

Depois, trata os downloads. Primeiro, calcula para quantos peers mandar mensagens **interested** usando como base as estatísticas de pedidos de blocos e quais deles foram cancelados, dentro de limites de bom funcionamento.

```

1  /* determines who we send "interested" messages to */
2  static void rechokeDownloads(tr_swarm * s) {
3      int i, maxPeers, rechoke_count = 0;
4      struct tr_rechoke_info * rechoke = NULL;
5      const int MIN_INTERESTING_PEERS = 5;
6      const int peerCount = tr_ptrArraySize(&s->peers);
7      const time_t now = tr_time();
8
9      if (tr_torrentIsSeed(s->tor)) return; // Não é necessário se já somos um seeder.
10     if (!tr_torrentIsPieceTransferAllowed(s->tor, TR_PEER_TO_CLIENT)) return;
11
12     /* decide HOW MANY peers to be interested in */
13     {
14         int blocks = 0, cancels = 0; time_t timeSinceCancel;
15
16         // Conta quantos blocos pedidos e cancelados cada peer tem. Isso é devido ao
17         // fato de que o Transmission gastou toda a banda de rede disponível, e agora
18         // precisa saber quantos peers se interessar. Peers não responsivos não serão
19         // levados em conta.
20         for (i = 0; i < peerCount; ++i) {
21             const tr_peer * peer = tr_ptrArrayNth(&s->peers, i);
22             const int b = tr_historyGet(&peer->blocksSentToClient, now, CANCEL_HISTORY_SEC);
23             const int c = tr_historyGet(&peer->cancelsSentToPeer, now, CANCEL_HISTORY_SEC);
24
25             if (b == 0) continue; // ignora peers não responsivos
26             blocks += b; cancels += c;
27         }
28
29         if (cancels > 0) {
30             /* cancelRate: of the block requests we've recently made, the percentage
31              * we cancelled. higher values indicate more congestion. */
32             const double cancelRate = cancels / (double) (cancels + blocks);
33             const double mult = 1 - MIN(cancelRate, 0.5);
34             maxPeers = s->interestedCount * mult;
35             tordbg(s, "cancel rate is %.3f -- reducing the number of peers we're interested "
36                 "in by %.0f percent", cancelRate, mult * 100);
37             s->lastCancel = now;
38         }
39
40         timeSinceCancel = now - s->lastCancel;
41         if (timeSinceCancel) {
42             const int maxIncrease = 15;
43             const time_t maxHistory = 2 * CANCEL_HISTORY_SEC;
44             const double mult = MIN(timeSinceCancel, maxHistory) / (double) maxHistory;
45             const int inc = maxIncrease * mult;
46             maxPeers = s->maxPeers + inc;
47             tordbg(s, "time since last cancel is %li -- increasing the number of peers"
48                 "we're interested in by %d", timeSinceCancel, inc);
49         }
50     }
51
52     /* don't let the previous section's number tweaking go too far... */
53     if (maxPeers < MIN_INTERESTING_PEERS) maxPeers = MIN_INTERESTING_PEERS;
54     if (maxPeers > s->tor->maxConnectedPeers) maxPeers = s->tor->maxConnectedPeers;
55     s->maxPeers = maxPeers;
56     (...)

```

Após saber o limite de peers, o Transmission procura quais deles contatará, de acordo com as partes que lhe faltam. Para isso, monta um vetor com informações dos peers, ordenando usando `qsort()` e passando como função comparadora que avalia, nesta ordem:

1. estado de choke: se um peer tiver cancelado 10% ou menos dos pedidos de blocos, terá preferência sobre os outros.
2. aleatoriedade.

```

1 typedef enum {
2     RECHOKE_STATE_GOOD, RECHOKE_STATE_UNTESTED, RECHOKE_STATE_BAD
3 } tr_rechoke_state;

1 static int compare_rechoke_info(const void * va, const void * vb) {
2     const struct tr_rechoke_info * a = va, * b = vb;
3     if (a->rechoke_state != b->rechoke_state)
4         return a->rechoke_state - b->rechoke_state;
5     return a->salt - b->salt;
6 }

1 if (peerCount > 0) {
2     bool * piece_is_interesting;
3     const tr_torrent * const tor = s->tor;
4     const int n = tor->info.pieceCount;
5
6     // Constrói um bitfield com as partes que ainda precisam ser baixadas.
7     piece_is_interesting = tr_new(bool, n);
8     for (i = 0; i < n; i++)
9         piece_is_interesting[i] = !tor->info.pieces[i].dnd &&
10             !tr_cpPieceIsComplete(&tor->completion, i);
11
12     // Decide quais peers se interessará, baseado na razão de cancelamentos / pedidos
13     for (i = 0; i < peerCount; ++i) {
14         tr_peer * peer = tr_ptrArrayNth(&s->peers, i);
15
16         if (!isPeerInteresting(s->tor, piece_is_interesting, peer)) {
17             tr_peerMsgsSetInterested(PEER_MSGS(peer), false);
18         }
19         else {
20             tr_rechoke_state rechoke_state;
21             const int blocks = tr_historyGet(&peer->blocksSentToClient, now,
22                 CANCEL_HISTORY_SEC);
23             const int cancels = tr_historyGet(&peer->cancelsSentToPeer, now,
24                 CANCEL_HISTORY_SEC);
25
26             // Peers bons cancelam 10% ou menos dos pedidos.
27             if (!blocks && !cancels) rechoke_state = RECHOKE_STATE_UNTESTED;
28             else if (!cancels) rechoke_state = RECHOKE_STATE_GOOD;
29             else if (!blocks) rechoke_state = RECHOKE_STATE_BAD;
30             else if ((cancels * 10) < blocks) rechoke_state = RECHOKE_STATE_GOOD;
31             else rechoke_state = RECHOKE_STATE_BAD;
32
33             if (rechoke == NULL) rechoke = tr_new(struct tr_rechoke_info, peerCount);
34
35             // valor aleatório entre 0 e INT_MAX
36             rechoke[rechoke_count].salt = tr_cryptoWeakRandInt(INT_MAX);
37             rechoke[rechoke_count].peer = peer;
38             rechoke[rechoke_count].rechoke_state = rechoke_state;
39             rechoke_count++;
40         }
41     }
42     (...)
43 }
44
45 // Ordena os peers de acordo com o interesse.
46 qsort(rechoke, rechoke_count, sizeof(struct tr_rechoke_info), compare_rechoke_info);
47 s->interestedCount = MIN(maxPeers, rechoke_count);
48 for (i = 0; i < rechoke_count; ++i) // Envia mensagens interested.
49     tr_peerMsgsSetInterested(PEER_MSGS(rechoke[i].peer), i < s->interestedCount);
50 (...)
51 }

```

Capítulo 4

Conceitos de Computação no BitTorrent

O BitTorrent é um protocolo cuja existência depende de vários componentes das mais variadas áreas de estudo da Computação. Neste capítulo, mostraremos alguns desses componentes e de que forma o Transmission os implementa, a fim de conseguir desempenhar bem sua função de cliente BitTorrent.

4.1 Estruturas de dados

Conjuntos são tão fundamentais na Ciência da Computação quanto na matemática. Os conjuntos manipulados por algoritmos são, em geral, alterados ao longo do tempo, sendo chamados de dinâmicos. Alguns algoritmos utilizam conjuntos, realizando diversas operações sobre eles, como inserção, remoção e testes de pertinência de elementos. Conjuntos dinâmicos que permitem essas operações são chamados de dicionários [12].

Na linguagem C, geralmente são implementados usando estruturas (*structs*), que são coleções de variáveis (membros), independentes de tipo, agrupadas sobre o mesmo nome. Com isso, várias implementações de dicionário foram criadas, cada uma com suas peculiaridades. Dentre as estruturas utilizadas no Transmission encontramos:

- vetores (*arrays*);
- listas ligadas (*linked lists*);
- filas (*queues*); e
- tabelas hash (*hash tables*).

Vetores

Vetores (ou *arrays*) são a implementação de vetores matemáticos de maneira virtual. Na prática, consistem de sequência ou listas de variáveis do mesmo tipo. Na linguagem C, podem ser declaradas de forma estática ou dinâmica.

Vetores estáticos têm tamanho fixo estabelecido na sua declaração em tempo de compilação, tendo espaços de memória reservados na pilha de execução de acordo com o tipo, não podendo ser alterado durante a execução do programa.

```
1 char announce[1024]; // URL de announce
```

Já os vetores dinâmicos são blocos de memória alocados e liberados durante a execução do programa. Para isso, são utilizados ponteiros para um objeto do mesmo tipo de cada elemento do vetor. Dessa forma, a memória é reservada em tempo de execução na memória heap. Assim, pode ter seu tamanho redimensionado conforme a necessidade.

```
1 int *a = malloc( 3*sizeof(int) ); // aloca memoria para um vetor de 3 inteiros
2 free(a);                        // desaloca a memoria alocada
```

Apesar dessa diferença, ambos os tipos de vetores funcionam da mesma maneira, usufruindo da aritmética de ponteiros e acesso instantâneo ao valor armazenado.

```
1 int b[3];
2 int *c = malloc( 3*sizeof(int) );
3
4 b[0] = 1; b[1] = 3; b[2] = 5;
5 c[0] = 2; c[1] = 4; c[2] = 6;
6
7 printf("b[1] = %d, *(c+2) = %d\n", b[1], *(c+2)); // b[1] = 3, *(c+2) = 6
```

O Transmission não aloca seus vetores dinâmicos literalmente desta forma, pois possui suas funções próprias onde encapsula o código mostrado.

Listas ligadas

Listas ligadas é uma estrutura de dados que organiza os objetos de forma linear, assim como os vetores. Porém, enquanto estes possuem índices que determinam a sua posição, em listas, cada elemento possuem um ponteiro para o elemento seguinte. Por causa disso, seu comprimento se altera organicamente conforme novos elementos vão sendo criados e inseridos, consumindo somente a memória necessária.

Os nós de listas ligadas são definidos usando-se estruturas.

```
1 typedef struct tr_list {  
2     void * data;  
3     struct tr_list * next; // ponteiro para o próximo elemento  
4     struct tr_list * prev; // ponteiro para o elemento anterior  
5 } tr_list;
```

Para a estrutura ser usada, o Transmission utiliza uma função que aloca memória dinamicamente para um objeto com valor nulo em todos os seus campos.

```
1 static tr_list * recycled_nodes = NULL;  
2  
3 static tr_list* node_alloc(void) {  
4     tr_list * ret; // ponteiro que aponta para a região alocada  
5  
6     if (recycled_nodes == NULL) { // Se não houver elementos reciclados,...  
7         ret = tr_new(tr_list, 1); // ... aloque um novo.  
8     }  
9     else { // Caso contrário, reutilize, reapontando as tomando o controle do central  
10         ret = recycled_nodes; // ... referências dos elementos adjacentes...  
11         recycled_nodes = recycled_nodes->next; // ... e tomando o controle do central  
12     }  
13  
14     *ret = TR_LIST_CLEAR; // limpa campos do elemento  
15     return ret; // devolve o ponteiro para o elemento  
16 }
```

Existem vários tipos de listas ligadas:

simplesmente ligada: possui somente um ponteiro para o próximo elemento;

duplamente ligada : possui 2 ponteiros, um para o elemento anterior e outro para o próximo elemento;

multiplamente ligada : possui ponteiros vários elementos, porém ligando-os em ordens diferentes;

circularmente ligada : quando o último elemento liga a lista de volta ao 1º elemento; e

com cabeça : quando possui um elemento falso somente para ajudar a manipular as listas.

Comparando-se vetores e listas ligadas, cada um tem suas vantagens e desvantagens em relação à complexidade de seus algoritmos de manipulação.

Operação	Vetor	Lista ligada
Busca por posição	$\Theta(1)$	$\Theta(n)$
Inserção/Remoção (início)	$\Theta(n)$	$\Theta(1)$
Inserção/Remoção (fim)	$\Theta(1)$	$\Theta(1)$ (c/ cabeça) $\Theta(n)$ (s/ cabeça)
Inserção/Remoção (meio)	$\Theta(n)$	$\Theta(n)$
Redimensionamento	$\Theta(n)$ (estático) ? (dinâmico)	não necessita

Tabela 4.1: tabela com os consumos de tempo de voperações sobre vetores e listas ligadas. OBS: tempos de buscas são considerados lineares. Redimensionamento de vetor dinâmico depende da implementação da linguagem C.

Por conta da agilidade que é conseguida na manipulação de listas ligadas de tamanhos imprevisíveis, o Transmission as utiliza em várias partes do seu código.

Tabelas hash

Tabelas hash são estruturas de dados eficientes na implementação de dicionários. Apesar de buscas demorarem tanto quanto procurar um elemento em uma lista ligada - $\Theta(n)$ no pior caso -, o espalhamento é bastante eficiente. Isso faz com que o tempo médio de uma busca seja $O(1)$ [12].

Uma tabela hash generaliza a noção do vetor de elementos comum. Nele, o endereçamento direto nos permite avaliar o conteúdo de uma posição em $O(1)$. O que torna esta tabela especial é a vantagem de transformar um certo conteúdo possuir uma chave específica e exclusiva, fornecendo um meio de se encontrar essa chave. Esse meio é uma **função de hash**.

Às vezes, funções de hash fazem com que 2 conteúdos possuam a mesma chave, ou seja, as chaves colidem. Para esses casos, existem várias técnicas de solução de conflitos, porém colisões podem ser evitadas com boa função de hash, descritas a seguir.

A tabela hash usada pelo Transmission aparece no DHT, porém de uma forma mais simples: não existe “a função de hash do DHT” como de costume, onde existe uma função característica para uma modelagem de tabela. Ao invés disso, as chaves já estão calculadas, sendo os IDs do torrents e dos peers do Kademlia.

4.2 Funções de hash

Funções de hash são funções matemáticas usadas para gerar conteúdo de comprimento fixo para referência ao conteúdo original.

Isso é útil quando existe grandes quantidades de dados a serem indexados. Por exemplo, numa busca em uma tabela de dados ou tarefas de comparação de dados tal como detecção de duplicatas ou de trechos de sequências de DNA semelhantes. Outro uso é na criptografia, quando é utilizado para comparar um conjunto de dados recebido com outro já existente, verificando sua igualdade.

Em geral, funções de hash não são inversíveis, ou seja, não é possível recuperar o valor de entrada para um dado [valor hash](#). Quando usado para fins criptográficos, são construídas de forma que essa reconstrução seja impossível sem que um imenso poder computacional seja utilizado. Por conta disso, é igualmente difícil fingir um valor hash para esconder dados maliciosos, sendo então usado pelo algoritmo [algoritmo PGP \(Pretty Good Privacy\)](#).

Outra característica importante é o determinismo. Quando a função é executada para dois dados de entrada iguais, ela deve produzir o mesmo valor. Essa condição é fundamental no caso de uma tabela hash, pois a busca deve encontrar o mesmo local onde o algoritmo de inserção armazenou o dado, logo precisando do mesmo valor hash.

Outros usos para funções de hash são em [verificações por soma \(checksums\)](#), códigos de correção de erros e cifras.

SHA-1

O SHA-1 é uma função de hash criada pela NSA, a Agência de Segurança Nacional americana, em 1995, e tem seu nome da abreviação de *Secure Hash Algorithm* (algoritmo de hash seguro). Seu uso foi difundido depois que seu predecessor, o algoritmo MD5, foi constatado com colisão de valor hash prática realizada em um computador comum [94].

Pertencente a uma família de algoritmos, que conta ainda com as versões SHA-0, SHA-2 (com funcionamento para vários comprimentos de bits de saída) e SHA-3, o SHA-1 teve falhas expostas comprovadas por colisão, aida que de difícil realização atualmente. Essa família é caracterizada por possuir algoritmos iterativos, baseada no desenho do algoritmo MD4 [55].

O resulta dessa função é um valor hash de 160 bits (ou 20 bytes), que forma um número hexadecimal de 40 caracteres. A função de compressão do algoritmo consiste de três partes:

1. expansão da mensagem: a mensagem de entrada é expandida para que o bloco de dado total seja múltiplo de 512 bits;
2. transformação de estado: consiste de passos simples de operações de números binários usando alguns valores pré-definidos, onde uma variável de encadeamento é usada de mensagem de entrada para iteração seguinte e os blocos da mensagem expandida se tornam as novas chaves de iteração;

3. retroalimentação: ao final do processamento de um bloco de 512 bits, a mensagem de entrada da transformação de estado é adicionada ao valor de saída. Esta operação é chamada de construção de Davies-Meyer e garante que se a mensagem de entrada for fixada então a função de compressão é não-inversível na variável de encadeamento.

Uma das implementações conhecidos para a linguagem C é a biblioteca OpenSSL [50], de código aberto, e é usada pelo Transmission no desenvolvimento de códigos de funções de hash, criptografia de dados e dados pseudo-aleatórios. O OpenSSL foi programado de forma otimizada, possuindo um código bastante diferente do usual.

O Transmission, seguindo a API do OpenSSL, possui a sua função de hash SHA-1. Na sua versão, calcula o valor hash de um bloco de dados em partes, utilizando-na para obter o valor hash do torrent, na criação de um ID para o DHT que ele possui, e na verificação da integridade das partes obtidas de outros peers. Neste último, a parte completada tem seu valor hash calculado e comparado com o valor que está contido no arquivo .torrent: se ambos os valores coincidirem, então a parte foi adquirida sem perdas, caso contrário será baixada novamente.

```
----- <openssl/sha.h>:101 -----
1 typedef struct SHAstate_st {
2     SHA_LONG h0, h1, h2, h3, h4;
3     SHA_LONG Nl, Nh;
4     SHA_LONG data[SHA_LBLOCK];
5     unsigned int num;
6 } SHA_CTX;

----- ./libtransmission/crypto.c:38 -----
1 void tr_shal(uint8_t * setme, const void * content1, int content1_len, ...) {
2     va_list vl;
3     SHA_CTX sha;
4     const void * content;
5
6     SHA1_Init(&sha);
7     SHA1_Update(&sha, content1, content1_len);
8
9     va_start(vl, content1_len);
10    while ((content = va_arg(vl, const void*)))
11        SHA1_Update(&sha, content, va_arg(vl, int));
12    va_end(vl);
13
14    SHA1_Final(setme, &sha);
15 }
```

4.3 Criptografia

A criptografia é o estudo e prática de técnicas que visam a segurança de informações de várias maneiras, a fim de que todas as partes de uma transação confiem que os objetivos daquela segurança tenham sido alcançadas. Com início há mais de quatro mil anos atrás, no Egito antigo, a área se manteve em atividade com a necessidade de se cifrar e quebrar mensagens, se tornando mais necessária com o advento da computação.

Os quatro objetivos principais da criptografia são de oferecer [43]:

confidencialidade, para se manter um conteúdo de informação sob sigilo somente àqueles autorizados a visualizá-lo. Suas inúmeras formas vão desde a proteção física até os algoritmos matemáticos que tornam o conteúdo ininteligível;

integridade dos dados, que foca na alteração não autorizada de dados, de forma a detectar essa manipulação;

autenticação, relacionada a identificação de dados e entidades. Duas partes que iniciam uma comunicação devem se identificar. Da mesma forma, uma informação deve poder ser autenticada a partir do envio para o destinatário;

aceitação, que previne uma entidade de negar uma ação ou compromisso previamente estabelecido, necessitando de um procedimento que envolva um terceiro para resolver alguma disputa.

Existem dois tipos principais de criptografia: por chaves simétricas ou por chaves públicas. Por meio de chaves simétricas, um sistema criptográfico usa uma mesma chave para encriptar e decriptar mensagens, sejam estas feitas por cifragem de fluxo de dados (onde cada caractere da mensagem é computado por vez) ou de blocos de dados (onde blocos da mensagem são computados). Uma desvantagem das chaves simétricas [68] é que é necessário um gerenciamento das chaves utilizadas, a fim de se manter a criptografia segura. Para isso, idealmente, cada par de entidades que desejam se comunicar entre si devem compartilhar a mesma chave, fazendo com que o número total de chaves necessárias numa rede seja proporcional ao quadrado do número de membros. Este era o único método conhecido até junho de 1976 [15].

Já as criptografias que utilizam de chaves públicas surgiram a partir de outro trabalho de Whitfield Diffie, Martin Hellman [16], onde propuseram o sistema criptográfico por meio da troca de chaves assimétricas (*Diffie-Hellman Key Exchange*). Nessa troca, são usadas duas chaves diferentes porém relacionadas matematicamente, onde a chave privada - para posse apenas do seu dono - é usada para a geração de uma chave pública para distribuição livre. Assim, enquanto a chave pública de uma entidade é usada para a criptografia dos dados pela outra entidade, apenas a respectiva chave privada pode descriptografá-la.

Atualmente, existem vários algoritmos e métodos de criptografia, cada qual com suas vantagens e desvantagens, e cenários de aplicação mais adequados do que outros. Porém, a criptografia tem como mote “não existe sistema de segurança impenetrável”, o que nos leva à área de criptoanálise, que é a arte e ciência de se analisar sistemas de segurança de forma a descobrir seus aspectos ocultos, utilizando-os para quebrar os respectivos sistemas criptográficos. Assim, muitos dos métodos existentes possuem falhas descobertas: enquanto algumas já são práticas, outras são apenas teóricas por falta de capacidade computacional atual.

Troca de chaves Diffie-Hellman

Este método, também conhecido por *Diffie-Hellman-Merkle key exchange*, é uma das formas mais conhecidas de troca de chaves criptográficas. Ele permite que duas partes que não possuem conhecimento a priori do outro estabeleçam um segredo comum utilizando métodos de comunicação inseguros.

O método algébrico utiliza grupos multiplicativos de inteiros módulo p , onde p é um número primo e g é chamado de número gerador. O protocolo pode ser explicado no seguinte exemplo, que usa duas partes (Alice e Bob): suponha que Alice e Bob desejam estabelecer um meio seguro de comunicação, que Eve deseja espionar realizando um “ataque de homem no meio”, tendo acesso a todas as informações que Alice e Bob trocarem.

1. Alice e Bob escolhem dois números inteiros p primo e g ;
2. Alice escolhe um número inteiro X_A para sua chave privada, e envia para Bob o resultado $Y_A = g^{X_A} \bmod p$.
3. Bob então escolhe um número inteiro X_B para sua chave privada, e envia para Alice o resultado $Y_B = g^{X_B} \bmod p$.
4. Alice, então, calcula a chave compartilhada $S_A = Y_B^{X_A} \bmod p$. Bob faz o mesmo, ou seja, calcula $S_B = Y_A^{X_B} \bmod p$.
5. como $S_A = S_B = S$, Alice e Bob passam a utilizar a chave S .

A princípio, não é óbvio ver que $S_A = S_B = S$, mas é fácil mostrar. Considere Alice e suas chaves. A chave que ela recebeu de Bob, Y_B , foi resultado de $Y_B = g^{X_B} \bmod p$. Então, o cálculo de S_A feito por ela é equivalente a $S_A = (g^{X_B})^{X_A} \bmod p$.

Analogamente, Bob recebeu Y_A de Alice, que foi resultado de $Y_A = g^{X_A} \bmod p$. Assim, o cálculo dele de S_B é equivalente a $S_B = (g^{X_A})^{X_B} \bmod p$. Porém, podemos manipular um pouco as equações, chegando em

$$\begin{aligned} S_A &= (g^{X_B})^{X_A} \bmod p \\ &= g^{(X_B \cdot X_A)} \bmod p \\ &= g^{(X_A \cdot X_B)} \bmod p \\ &= (g^{X_A})^{X_B} \bmod p \\ &= S_B \end{aligned}$$

Veja que não importa que Eve tenha obtido p , g , Y_A ou Y_B . Ela não conseguirá obter S pois este depende de X_A e X_B . Além disso, Eve pode tentar calcular X_A e X_B , porém a

difficuldade deste cálculo depende dos tamanhos de p , X_A e X_B : quanto maiores forem esses números, mais difíceis serão os cálculos inversos, que são chamados de “problemas de logaritmo discreto”. É praticamente impossível descobrir essas chaves privadas em uma quantidade de tempo razoável. Assim, esse método é considerado seguro, enquanto a computação quântica não estiver desenvolvida o suficiente para que novos algoritmos possam ser usados [57].

RC4

O RC4 (ou ainda “Rivest Cypher 4”, “Ron’s Code 4” ou “Arc Four”) é uma função criptográfica criada por Ron Rivest em 1987. Inicialmente era segredo comercial, porém em 1994 seu código foi publicado na lista de discussão de criptografia CypherPunks [34], se espalhando pela Internet rapidamente. Seu uso se tornou comum, sendo usado por muitos softwares, chegando a protocolos como as encriptações de placas de rede sem fio WEP e WPA ou ainda o protocolo de segurança TLS para conexões de Internet.

O RC4 é um algoritmo de chave simétrica que se divide em duas partes: na primeira parte, ele executa o algoritmo de escalonamento de chaves (*key scheduling*), que utiliza uma chave de tamanho variável entre 1 e 256 bytes para inicializar uma tabela de estados. Cada elemento dessa tabela é permutado pelo menos uma vez e será usado na geração de bytes pseudoaleatórios na segunda parte.

Na segunda parte, executa o algoritmo de geração pseudoaleatório, onde modifica o estado (também permutando os elementos pelo menos uma vez) e resulta em 1 byte da chave de fluxo, que então é mesclada usando XOR bit a bit com o próximo byte da mensagem para produzir ou próximo byte da mensagem cifrada (na encriptação) ou da decifrada (na decriptação), já que o XOR é uma função involuntária (ou seja, é uma função que é a própria inversa).

```

1 void tr_cryptoEncrypt(tr_crypto * crypto, size_t buf_len, const void * buf_in, void * buf_out)
2 {
3     RC4(&crypto->enc_key, buf_len, (const unsigned char*) buf_in, (unsigned char*) buf_out);
4 }

1 void tr_cryptoDecrypt(tr_crypto * crypto, size_t buf_len, const void * buf_in, void * buf_out)
2 {
3     RC4(&crypto->dec_key, buf_len, (const unsigned char*) buf_in, (unsigned char*) buf_out);
4 }
```

Falhas de segurança

Apesar de ainda ser bastante utilizado, o RC4 não pode ser considerado criptograficamente seguro. Na primeira década dos anos 2000 foram publicados diversos trabalhos sobre falhas do método e até ataques diferentes que conseguiram obter os dados originais [1, 27–29].

Porém, como é dito na especificação da extensão criptográfica no protocolo BitTorrent [30], “o objetivo não é de criar um protocolo criptograficamente que consegue sobreviver a observações dos pacotes de dados e recursos computacionais consideráveis na escala de tempo de redes. O objetivo é de aumentar a dificuldade o suficiente a fim de deter ataques baseados na obtenção de endereços IP e porta na comunicações entre peers e trackers”. Especificamente, o objetivo é prevenir que ISPs ou outros administradores de rede consigam bloquear ou impedir conexões BitTorrent entre esse peer e qualquer outro peer cujo endereço IP e porta apareça em alguma lista de peers enviada por um tracker. Por isso, essa especificação é chamada de ofuscação de peers, onde a idéia é usar o `info_hash` de um torrent como chave compartilhada entre o peer e o tracker.

Por exemplo, em um dos ataques ao RC4 usado no protocolo de rede sem fio WEP, é necessário que o atacante obtenha uma quantidade de bytes muito grande para que consiga decifrar o texto. Porém, enquanto em redes sem fio esses dados trafegados são facilmente alcançam aquela quantidade necessária, numa comunicação entre peers e trackers os dados transmitidos são muito menores, dificultando ainda mais o processo de quebra.

Criptografia no BitTorrent

O BitTorrent usa criptografia somente quando o usuário habilita a opção correspondente no programa cliente, criptografando comunicações com trackers e peers. O método escolhido pelo protocolo é o que está definido em uma de suas propostas de melhoria. Porém, esta está suspensa, sendo então usada de forma extraoficial atualmente pelos programas cliente em geral, inclusive o Transmission.

Comunicação com trackers

Quando criptografia for habilitada, as comunicações com trackers, ou seja, as requisições de `announces`, não devem enviar o `info_hash` do torrent. Ao invés disso, deve enviar o `sha_1h`, que é o valor hash SHA-1 do `info_hash` (que também é um valor hash SHA-1) na forma URL encode.

Já a resposta de trackers a `announces` se mantêm no mesmo formato, exceto pela lista de peers, que será ofuscada.

Para isso, a requisição do `announce` deve passar como parâmetros

supportcrypto: valor 1 indica que o peer pode criar e receber conexões criptografadas. Neste caso, se o tracker aceitar esta extensão do BitTorrent, as listas de peers que enviará em suas respostas priorizarão outros peers que também enviaram `supportcrypto=1` antes dos que não o fizeram.

requirecrypto: valor 1 indica que o peer irá criar e aceitar somente conexões criptografadas. Neste caso, as listas de peers que o tracker enviará conterão somente peers que também enviaram `supportcrypto=1` e `requirecrypto=1`.

cryptoport: quando o parâmetro de `requirecrypto=1`, é inteiro que representa a porta na qual o cliente irá utilizar para conexões criptografadas.

```
1  static char* announce_url_new(const tr_session * session, const tr_announce_request * req) {
2      (...)
3      tr_http_escape_sh1(escaped_info_hash, req->info_hash);
4      (...)
5
6      evbuffer_add_printf(buf, "%s%cinfo_hash=%s(...) &supportcrypto=1",
7                          req->url, strchr(req->url, '?') ? '&' : '?', escaped_info_hash, (...));
8
9      if (session->encryptionMode == TR_ENCRYPTION_REQUIRED)
10         evbuffer_add_printf(buf, "&requirecrypto=1");
11
12     (...)
13     return evbuffer_free_to_str(buf);
14 }
```

Comunicação com peers

A comunicação entre peers é criptografada usando RC4 e troca de chaves Diffie-Hellman-Merkle [93]. Para isso, o protocolo de *handshake* para mensagens entre peers é estendido de forma a efetuar esses cinco procedimentos criptográficos:

```

1 // Comentários retirados de
2 // http://wiki.vuze.com/w/Message_Stream_Encryption#Message_Stream_Encryption
3
4 // A é o remetente
5 // chave pública de A:  $Y_A = (G_A^X) \bmod P$ 
6 // B é o receptor
7 // chave pública de B:  $Y_B = (G_B^X) \bmod P$ 
8
9 /* P, S,  $Y_A$  and  $Y_B$  are 768bits long */
10 #define KEY_LEN 96
11
12 /* Prime P is a 768 bit safe prime,
13 "0xFFFFFFFFFFFFFFFFC90FDAA22168C234C4C6628B80DC1CD129024E088A67CC74
14 020BBEA63B139B22514A08798E3404DDEF9519B3CD3A431B302B0A6DF25F1437
15 4FE1356D6D51C245E485B576625E7EC6F44C42E9A63A36210000000000090563"*/
16 #define PRIME_LEN 96
17 static const uint8_t dh_P[PRIME_LEN] =
18 {
19     0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xC9, 0x0F, 0xDA, 0xA2,
20     0x21, 0x68, 0xC2, 0x34, 0xC4, 0xC6, 0x62, 0x8B, 0x80, 0xDC, 0x1C, 0xD1,
21     0x29, 0x02, 0x4E, 0x08, 0x8A, 0x67, 0xCC, 0x74, 0x02, 0x0B, 0xBE, 0xA6,
22     0x3B, 0x13, 0x9B, 0x22, 0x51, 0x4A, 0x08, 0x79, 0x8E, 0x34, 0x04, 0xDD,
23     0xEF, 0x95, 0x19, 0xB3, 0xCD, 0x3A, 0x43, 0x1B, 0x30, 0x2B, 0x0A, 0x6D,
24     0xF2, 0x5F, 0x14, 0x37, 0x4F, 0xE1, 0x35, 0x6D, 0x6D, 0x51, 0xC2, 0x45,
25     0xE4, 0x85, 0xB5, 0x76, 0x62, 0x5E, 0x7E, 0xC6, 0xF4, 0x4C, 0x42, 0xE9,
26     0xA6, 0x3A, 0x36, 0x21, 0x00, 0x00, 0x00, 0x00, 0x00, 0x09, 0x05, 0x63,
27 };
28
29 /* Generator G is "2" */
30 static const uint8_t dh_G[] = { 2 };
31
32 /*  $X_A$  and  $X_B$  are a variable size random integers.
33 They are the private key for each side and have to be discarded after the Diffie-Hellman
34 (DH) handshake is done. Minimum length is 128 bit. Anything beyond 180 bit is not
35 believed to add any further security and only increases the necessary calculation time.
36 You should use a length of 160bits whenever possible, lower values may be used when CPU
37 time is scarce. */
38 #define DH_PRIVKEY_LEN_MIN 16
39 #define DH_PRIVKEY_LEN 20
40
41 /* DH secret:  $S = (Y_A^{X_B}) \bmod P = (Y_B^{X_A}) \bmod P$  */

```

1) A envia Y_A para B

O Transmission, A, envia sua chave pública (Y_A) com um trecho de dados aleatórios cujo comprimento qualquer entre 0 e 512 bytes.

```

1  /* 1 A → B: Diffie Hellman YA, PadA */
2  static void sendYa(tr_handshake * handshake) {
3      int len; const uint8_t * public_key;
4      char outbuf[KEY_LEN + PadA_MAXLEN]; // 96 + 512 = 608 bytes
5      char *walk = outbuf;
6
7      /* add our public key (YA) */
8      public_key = tr_cryptoGetMyPublicKey(handshake->crypto, &len);
9      (...)
10     memcpy(walk, public_key, len); walk += len;
11
12     /* add some bullshit padding */
13     len = tr_cryptoRandInt(PadA_MAXLEN);
14     tr_cryptoRandBuf(walk, len); walk += len;
15
16     /* send it */
17     setReadState(handshake, AWAITING_YB);
18     tr_peerIoWriteBytes(handshake->io, outbuf, walk - outbuf, false);
19 }

```

2) A recebe Y_B de B

O outro peer, B, responde com sua chave pública (Y_B). Assim, a chave compartilhada S .

```

1  /* 2 A ← B: Diffie Hellman YB, PadB */
2  static int readYb(tr_handshake * handshake, struct evbuffer * inbuf) {
3      int isEncrypted;
4      const uint8_t * secret;
5      uint8_t yb[KEY_LEN];
6      struct evbuffer * outbuf;
7      size_t needlen = HANDSHAKE_NAME_LEN;
8
9      (...)
10     isEncrypted = memcmp(evbuffer_pullup(inbuf, HANDSHAKE_NAME_LEN),
11                          HANDSHAKE_NAME, HANDSHAKE_NAME_LEN);
12     (...)
13     tr_peerIoSetEncryption(handshake->io, isEncrypted ? PEER_ENCRYPTION_RC4
14                                                             : PEER_ENCRYPTION_NONE);
15
16     if (!isEncrypted) {
17         setState(handshake, AWAITING_HANDSHAKE); // Handshake não encriptado acaba aqui.
18         return READ_NOW;
19     }
20
21     (...)
22     /* compute the secret S */
23     evbuffer_remove(inbuf, yb, KEY_LEN);
24     secret = tr_cryptoComputeSecret(handshake->crypto, yb);
25     memcpy(handshake->mySecret, secret, KEY_LEN);

```

3) A envia para B as opções de criptografia e a mensagem de *handshake*

Então, A envia dados de forma criptografada: a mensagem de *handshake* e outras informações sobre a criptografia para B, na forma

HASH('req1', S), HASH('req2', SKEY) xor HASH('req3', S),
 ENCRYPT(VC, crypto_provide, len(PadC), PadC, len(IA)), ENCRYPT(IA)

onde:

HASH(): é a função que calcula o valor hash SHA-1 de todos os parâmetros de entrada concatenados;

ENCRYPT(): é a função RC4 com chave `HASH('keyA', S, SKEY)` (se $A \rightarrow B$), ou `HASH('keyB', S, SKEY)` (se $B \rightarrow A$). Os primeiros 1024 bytes da encriptação RC4 são descartados. O uso seguido desta função por uma parte a encriptar o fluxo de dados sem reinicializações ou trocas;

VC: é uma constante de verificação (*verification constant*), que é uma string de 8 bytes de valor 0x00, usada para verificar se a outra parte conhece *S* e SKEY, evitando ataques de por repetição do SKEY;

crypto_provide/crypto_select: são bitfield de 32 bits. Dois valores são usados atualmente, com o restante sendo reservado para uso futuro. O peer *A* deve ligar os bits de todos os métodos suportados por ele, enquanto o peer *B* deve ligar o bit do método escolhido dentre os oferecidos e enviar como resposta. Por enquanto, 0x01 indica sem encriptação e 0x02 indica o RC4;

PadC/PadD: reservados para futuras extensões do handshake. Hoje, possuem 0 bytes;

IA: conjunto de dados inicial de *A*. Pode ser 0 bytes se quiser esperar por negociação de encriptação;

```
1  /* 3 A → B now send these:
2  * HASH ('req1', S), HASH ('req2', SKEY) xor HASH ('req3', S),
3  * ENCRYPT (VC, crypto_provide, len (PadC), PadC, len (IA)), ENCRYPT (IA) */
4  outbuf = evbuffer_new();
5
6  /* HASH ('req1', S) */
7  {
8      uint8_t req1[SHA_DIGEST_LENGTH];
9      tr_shal(req1, "req1", 4, secret, KEY_LEN, NULL);
10     evbuffer_add(outbuf, req1, SHA_DIGEST_LENGTH);
11 }
12
13 /* HASH ('req2', SKEY) xor HASH ('req3', S) */
14 // SKEY = chave compartilhada para evitar ataques de homem no meio.
15 // No BitTorrent, SKEY = info hash do torrent.
16 {
17     int i;
18     uint8_t req2[SHA_DIGEST_LENGTH], buf[SHA_DIGEST_LENGTH],
19            req3[SHA_DIGEST_LENGTH];
20
21     tr_shal(req2, "req2", 4, tr_cryptoGetTorrentHash(handshake->crypto),
22            SHA_DIGEST_LENGTH, NULL);           // HASH ('req2', SKEY)
23     tr_shal(req3, "req3", 4, secret, KEY_LEN, NULL); // HASH ('req3', SKEY)
24
25     for (i = 0; i < SHA_DIGEST_LENGTH; ++i)
26         buf[i] = req2[i] ^ req3[i];
27
28     evbuffer_add(outbuf, buf, SHA_DIGEST_LENGTH);
29 }
30
31
32
33
```

```

34  /* ENCRYPT (VC, crypto_provide, len (PadC), PadC */
35  {
36      uint8_t vc[VC_LENGTH] = { 0, 0, 0, 0, 0, 0, 0, 0, 0 };
37
38      tr_peerIoWriteBuf(handshake->io, outbuf, false);
39      tr_cryptoEncryptInit(handshake->crypto);           // ENCRYPT(...)
40      tr_peerIoSetEncryption(handshake->io, PEER_ENCRYPTION_RC4);
41
42      evbuffer_add(outbuf, vc, VC_LENGTH);              // ...VC, ...
43      evbuffer_add_uint32(outbuf, getCryptoProvide(handshake)); // ...crypto_provide, ...
44      evbuffer_add_uint16(outbuf, 0);                   // ..., 0 bytes, ...
45  }
46
47  /* ...len (IA)), ENCRYPT (IA) */
48  {
49      uint8_t msg[HANDSHAKE_SIZE];
50      if (!buildHandshakeMessage(handshake, msg))
51          return tr_handshakeDone(handshake, false);
52
53      evbuffer_add_uint16(outbuf, sizeof(msg));          // ...len (IA)),
54      evbuffer_add(outbuf, msg, sizeof(msg));           // ... ENCRYPT (IA)
55      (...)
56  }
57
58  tr_cryptoDecryptInit(handshake->crypto);
59  (...)

```

4) *A* recebe de *B* a opção de criptografia escolhida e a mensagem de resposta ao *handshake* encriptada por RC4

Aqui, *B* envia como resposta

ENCRYPT(VC, crypto_select, len(padD), padD), ENCRYPT2(Payload Stream)

```

./libtransmission/handshake.c:493
1  /* 4 A ← B: ENCRYPT(VC, crypto_select, len(padD), padD), ENCRYPT2(Payload Stream) */
2  static int readVC(tr_handshake * handshake, struct evbuffer * inbuf) {
3      uint8_t tmp[VC_LENGTH];
4      const int key_len = VC_LENGTH;
5      const uint8_t key[VC_LENGTH] = { 0, 0, 0, 0, 0, 0, 0, 0, 0 };
6
7      /* note: this works w/o having to 'unwind' the buffer if
8       * we read too much, but it is pretty brute-force.
9       * it would be nice to make this cleaner. */
10     for (;;) {
11         (...)
12         memcpy(tmp, evbuffer_pullup(inbuf, key_len), key_len);
13         tr_cryptoDecryptInit(handshake->crypto);
14         tr_cryptoDecrypt(handshake->crypto, key_len, tmp, tmp); // (DE)CRYPT(VC, ...
15         if (!memcmp(tmp, key, key_len)) break;
16         (...)
17     }
18     (...)
19 }
20
21 static int readCryptoSelect(tr_handshake * handshake, struct evbuffer * inbuf) {
22     uint16_t pad_d_len; uint32_t crypto_select;
23     static const size_t needlen = sizeof(uint32_t) + sizeof(uint16_t);
24     (...)
25
26     tr_peerIoReadUInt32(handshake->io, inbuf, &crypto_select); // ... crypto_select, ...
27     handshake->crypto_select = crypto_select;
28     (...)
29     tr_peerIoReadUInt16(handshake->io, inbuf, &pad_d_len);      // ... len(padD), ...
30     (...)
31 }

```

```

32 static int readPadD(tr_handshake * handshake, struct evbuffer * inbuf) {
33     const size_t needlen = handshake->pad_d_len;
34
35     (...)
36     tr_peerIoDrain(handshake->io, inbuf, needlen); // ... padD), ...
37     tr_peerIoSetEncryption(handshake->io, handshake->crypto_select);
38     (...)
39 }
40
41 static int readHandshake(tr_handshake * handshake, struct evbuffer * inbuf) {
42     uint8_t pstrlen;
43     (...)
44     /* peek, don't read. We may be handing inbuf to AWAITING_YA */
45     pstrlen = evbuffer_pullup(inbuf, 1)[0];
46
47     if (pstrlen == 19) { /* unencrypted */
48         (...) // handshake não encriptado é tratado aqui
49     }
50     else { /* encrypted or corrupt */
51         tr_peerIoSetEncryption(handshake->io, PEER_ENCRYPTION_RC4);
52         (...)
53         // ENCRYPT2(Payload Stream)
54         tr_cryptoDecrypt(handshake->crypto, 1, &pstrlen, &pstrlen);
55         (...)
56     }
57     (...)

```

5) A envia para B o fim da mensagem de *handshake* encriptada por RC4

```

1 /* 5 A → B: ENCRYPT2(Payload Stream) */
2 (...)
3 if (!handshake->haveSentBitTorrentHandshake) {
4     uint8_t msg[HANDSHAKE_SIZE];
5     if (!buildHandshakeMessage(handshake, msg))
6         return tr_handshakeDone(handshake, false);
7     tr_peerIoWriteBytes(handshake->io, msg, sizeof(msg), false);
8     handshake->haveSentBitTorrentHandshake = 1;
9 }
10 (...)
11 }

```

```

1 void tr_peerIoWriteBytes(tr_peerIo * io, const void * bytes, size_t byteCount,
2     bool isPieceData)
3 {
4     struct evbuffer_iovec iovec;
5     (...)
6     if (io->encryption_type == PEER_ENCRYPTION_RC4) // ENCRYPT2(Payload Stream)
7         tr_cryptoEncrypt(&io->crypto, iovec.iov_len, bytes, iovec.iov_base);
8     (...)
9 }

```

4.4 Bitfields

Apesar de ser um simples array de bits usado no gerenciamento de partes que o programa já baixou ou não, foi percebido que o seu uso de forma não-convencional, chamado de *lazy bitfield*, pode ajudar a evitar o controle de banda (chamado de modelagem de tráfego, ou *traffic shaping* em inglês), feito por **ISPs**.

4.5 Protocolos de redes

Aqui vou explicar o que são os protocolos de rede TCP e UDP, apontar suas diferenças e mostrar os motivos pelos quais o UDP é preferido ao TCP no uso de endereços de announce de trackers.

4.6 Multicast

Apesar de não ser utilizado pelo protocolo BitTorrent, o multicast, que é uma forma de entregar dados a um grupo de computadores simultaneamente numa só transmissão, é usado pelo Transmission para tentar descobrir peers que estão na mesma rede local, otimizando as conexões.

4.7 Roteamento de pacotes

Em alguns roteadores que são ponte entre a Internet e a rede que ele gerencia, existe uma função de se configurar portas de comunicação de rede automaticamente usando-se o *Network Address Translation Port Mapping Protocol*. Assim, não é necessário realizar uma configuração específica somente para esse fim.

4.8 Retomada de downloads

No Transmission e em alguns outros softwares que realizam downloads, existe a função de se pausar a transferência do arquivo para que seja retomado em outro momento. Nesta seção, mostrarei qual a idéia por trás desse mecanismo e a forma como foi implementado no Transmission.

4.9 Conexão com a Internet

Aqui mostrarei a parte técnica da programação em linguagem C para utilização de transmissão de dados por rede.

4.10 IPv6

O IPv6 é a versão mais recente do protocolo de Internet (IP), que foi criado para substituir o IPv4, que atualmente é mais o usado porém sofre de exaustão de endereços. Nesta seção, falarei sobre o novo protocolo e quais as implicações na programação de softwares com comunicação de redes.

4.11 Threads

Dentro de um contexto de programas que utilizam a Internet e funcionalidades que chegam próximas ao tempo real, processamentos pesados devem ser tratados com cautela a fim de se manter a instantaneidade do processo. Aqui explicarei como o Transmission usa o conceito de *threads* para paralelizar esses processamentos e, com isso, conseguir utilizar as informações de rápida mudança antes que seja necessário obtê-las novamente.

4.12 Engenharia de Software

O Transmission é um programa extenso e complexo, desenvolvido por vários programadores que estão espalhados pelo globo. Nesta seção, abordarei alguns pontos utilizados pelos desenvolvedores na manutenção do código aberto de qualidade e em funcionamento.

Capítulo 5

Comentários Finais

Glossário

announce

endereço [URL](#) do [tracker](#) para troca de informações, onde este recebe de [peers](#) informações sobre as respectivas situações com relação a um torrent específico naquele momento, as processa e então responde informando sobre a situação geral daquele torrent e uma lista de outros peers conectados naquele momento. [16](#), [17](#), [19](#), [21--23](#), [79](#), [86](#), [92](#)

anycast

método de endereçamento e roteamento de rede onde os datagramas de um único remetente são roteados para um membro de um grupo de receptores potenciais que estão definidos pelo mesmo intervalo no endereço de destino. Geralmente é usado para serviços que demandem alta disponibilidade. [4](#)

API

Application Programming Interface (ou Interface de Programação de Aplicativos) . [75](#) [escrever](#)

arquivo .torrent

arquivo que contém [metadados](#), como a lista dos nomes dos arquivos a serem baixados e seus tamanhos, [checksums](#) das partes desses arquivos, endereços de um ou mais [trackers](#), etc, formando um pacote chamado [torrent](#). [8](#), [11](#), [12](#), [14--17](#), [21](#), [28](#), [38](#), [40](#), [47](#), [48](#), [75](#), [93](#)

Audiogalaxy

Rede [P2P](#) de compartilhamento de músicas [MP3](#) criado em 1998. [2](#), [3](#)

bencode

“codificação B”, pronunciado *bê encode*; formato de codificação compacta de arquivos [torrent](#) para transmissão de [metadados](#). [12](#), [17](#), [19--21](#), [31](#), [32](#), [38](#), [40](#), [42](#)

beta tester

usuários de uma versão beta de um software. [6](#)

bootstrap

. [42](#), [43](#) [escrever](#)

bucket

. [30](#), [34](#), [35](#), [42](#), [43](#) [escrever](#)

case insensitive

em português, insensível ao tamanho das letras (maiúsculas ou minúsculas); uma [string](#) case insensitive não é diferenciada com letras iguais em caixas alta ou baixa. Assim, [F00](#), [Foo](#) e [foo](#) seriam [strings](#) dadas como iguais. [93](#)

checksum

em português, soma de verificação; bloco de dados de tamanho fixo gerado por algum algoritmo de soma para verificação, usado no certificado de integridade contra problemas durante a transmissão ou de armazenamento (leitura ou escrita). [74](#), [89](#)

DHT

do inglês *distributed hash table*; [tabela hash](#) distribuída, ou seja, é um serviço de busca similar a uma [tabela hash](#), mas descentralizada e na forma de sistema distribuído. [5](#), [26](#), [27](#), [31](#), [32](#), [40](#), [43](#), [44](#), [49](#), [55](#), [73](#), [75](#), [91](#)

eDonkey

lançado em 6 de setembro de 2000, o protocolo foi inaugurado juntamente com o software que o utilizava, o eDonkey2000, mas inúmeros softwares cliente para diferentes plataformas surgiram nos dias seguintes ao lançamento. [4](#), [5](#)

endian

. [50](#), [52](#)

escrever

função de hash

é uma função ou algoritmo matemático que mapeia um dado de comprimento variável em outro de comprimento fixo. [15](#), [17](#), [48](#), [73--75](#), [94](#)

gateway NAT

do inglês *Network Address Translation*; tradução de endereço de rede, é uma funcionalidade empregada por roteadores de rede, que por associarem um endereço de Internet (o fornecido pelo [ISP](#)) para vários dispositivos dentro dessa rede, precisa manter uma tabela de tradução de endereços para que saiba rotear os pacotes de dados transmitidos entre as redes interna e externa. [18](#), [40](#)

Gnutella

software de compartilhamento [P2P](#) desenvolvido por 3 programadores da empresa Nullsoft, recém adquirida da AOL Inc., lançado em 2000 sob a licença GPL. No dia seguinte ao lançamento, a AOL ordenou indisponibilizar o software, alegando problemas legais e proibindo a continuação do desenvolvimento. Alguns dias depois, o protocolo já tinha sido alvo de engenharia reversa e já havia softwares que o implementavam. [4](#), [5](#), [7](#), [27](#)

HTTP GET

método de requisição do protocolo HTTP que permite uso de [query strings](#) para troca de informações. [17](#)

ISP

do inglês *Internet Service Provider*; fornecedores de acesso à Internet, que são empresas que vendem serviço e equipamento que permitem o acesso de um computador pessoal à Internet. [2](#), [79](#), [85](#), [90](#)

Kademlia

[DHT](#) usado em [P2P](#) que especifica a estrutura da rede e a troca de informações através de buscas de nós, guardando as localizações de recursos que estão na rede. [5](#)

k-bucket

. [30](#), [31](#), [34](#), [42--44](#)

escrever

leecher

em português, sugador; nome dado ao [peer](#) que ainda não terminou um download de um torrent. [8](#), [19](#), [21](#), [56](#)

magnet link

em português, link magnético; padrão aberto, definido por convenção, de esquema de [URI](#) utilizado para localizar recursos de rede BitTorrent para download. [14--16](#)

metadado

dados sobre outros dados; informação sobre outra informação. [8](#), [14](#), [89](#)

MP3

do inglês *MPEG-1/2 Audio Layer 3*; formato patenteado de compressão de dados de áudio digital, que usa um método de compressão de dados com perdas. [2](#), [3](#), [89](#)

overhead

. [26](#), [51](#)

escrever

P2P

do inglês *peer-to-peer*; redes de arquitetura descentralizada e distribuída, onde cada nó ([peer](#)) fornece e consome recursos. [2--4](#), [6--8](#), [26](#), [89--91](#)

peer

em português, significa par, colega; nome que se dá a cada nó da rede, ou seja, a um computador conectado. [4](#), [5](#), [7](#), [8](#), [11](#), [13](#), [15](#), [17](#), [18](#), [20](#), [21](#), [23--28](#), [32](#), [38](#), [40](#), [42--44](#), [46--53](#), [55--57](#), [62--68](#), [73](#), [75](#), [79](#), [80](#), [82](#), [83](#), [86](#), [89](#), [91--93](#)

PGP

. [74](#)

escrever

pool

em português, poço de recursos; conjunto de recursos alocados em memória que são pré-computados, a fim de estarem prontos para serem utilizados. [24](#)

proxy

servidor que funciona como intermediário de acessos, repassando requisições de um computador cliente para o servidor de destino. 18

query string

em português, **string** de busca; parte de uma **URL** que possui um dicionário de dados a serem transmitidos para alguma aplicação de Internet, determinado por um caractere **?**. Por exemplo, em **http://um/endereco/qualquer/?key1=value1&key2=value2**. 15, 90

RIAA

do inglês *Recording Industry Association of America*; Associação da Indústria de Gravação da América, organização que representa as gravadoras musicais e distribuidores, e tem sido autora de ações judiciais devido a quebra de direitos autorais causada por compartilhamento indevido de música. 3

RPC

. 31, 34, 38

escrever

scrape

do inglês *screen scraping*; sondagem de tela, que era um processo automatizado para recolhimento de informações sobre torrents que acessavam páginas web dos **trackers**. Por questões práticas e econômicas, foi estabelecido que neste endereço **URL** do tracker, ao contrário do **announce**, o tracker somente informa as quantidades de **peers** que estão participando de uma lista de torrents naquele momento. 21, 22, 24

seeder

em português, semeador; nome dado ao **peer** que já terminou um download de um torrent e que, por ainda estar conectado à rede, fornece partes a possíveis interessados. 8, 18--21, 56

string

sequência de caracteres. 12--15, 17--20, 31, 32, 34, 38, 40, 42, 48, 49, 90, 92, 93

swarm

em português, enxame; grupo de **peers** que estão compartilhando dados de um mesmo torrent num determinado momento. 8, 9, 11, 14, 15, 17, 20, 24, 40, 44, 47, 56, 59

swarming

também chamado de transmissão de arquivos por segmentação ou de múltiplas fontes, é a transmissão em paralelo de um arquivo, a partir de um ou vários locais onde estiver disponível, para um único destino. Cabe ao software do destinatário juntar as partes recebidas. 5

tabela hash

ou *mapa de hash*, é uma estrutura de dados que cria uma lista de correspondência chave-valor, onde os dados são guardados como valores e indexados por seus respectivos *valores hash*. 5, 27, 70, 73, 74, 90

TCP

do inglês *Transmission Control Protocol*; protocolo de controle de transmissão, é um dos protocolos principais de Internet (IP). Pertencente à camada de transporte de dados de rede, provê conexões cujos pacotes de dados são pesados, mas com entregas confiáveis, ordenadas e com verificação de erros e de congestionamento. 17, 21, 55, 63

thread

. 24

escrever

torrent

conjunto de um ou mais arquivos definidos por um [arquivo .torrent](#). 8, 9, 11, 13--16, 18, 21, 25, 27, 28, 40, 47--49, 51--53, 56, 57, 59, 62, 64, 73, 75, 79, 89

tracker

em português, rastreador; servidor que funciona como um ponto de encontro de [peer](#). 6, 8, 13, 15--19, 21, 22, 24, 47, 49, 79, 80, 86, 89, 92

UDP

do inglês *User Datagram Protocol*; protocolo de controle de transmissão, é um dos protocolos principais de Internet (IP). Pertencente à camada de transporte de dados de rede, provê conexões cujos pacotes de dados são leves, mas com entregas não confiáveis, desordenadas e sem verificação de erros (feita na camada de aplicação) e de congestionamento. 17, 21, 30, 31, 40

URI

do inglês *Uniform Resource Identifier*; Identificador Uniforme de Recursos, é uma [string](#) usada para identificar algum recurso, especificando algum protocolo e um caminho. Por exemplo, o URI [file:///arquivo.txt](#) indica um arquivo computador local (nome de esquema [file](#)), enquanto [http://pagina.com](#) se refere a uma página de Internet (nome de esquema [http](#)). 14, 91, 93

URL

do inglês *Uniform Resource Locator*; Localizador Uniforme de Recursos, é uma [string](#) usada para identificar algum recurso na Internet, especificando algum protocolo de comunicação. 89, 92, 93

URL encode

em português, codificação de [URL](#); por possuir caracteres especiais, [URLs](#) devem converter esses caracteres quando se desejar transmití-los. Letras (A-Z e a-z), números (0-9), caracteres ([. ~ _](#)); espaços são convertidos para [+](#) ou [%20](#); o restante é convertido para o respectivo valor em hexadecimal do caractere convertido para UTF-8. 15, 17, 48, 79

URN

do inglês *Uniform Resource Name*; Nome Uniforme de Recursos, é o nome histórico dado a uma [URI](#) que usa o esquema [urn](#). Sua sintaxe é [urn:<NID>:<NSS>](#), onde [urn](#) é o prefixo [case insensitive](#), [<NID>](#) é o identificador de espaço de nomes, que determina a interpretação sintática de [<NSS>](#), que é a [string](#) específica do espaço de nomes usado. 15

valor hash

ou *hash*; valores gerados por uma *função de hash*. 5, 14--17, 21, 28, 38, 40, 48, 51, 74, 75, 79, 83

XOR

. 29, 78

escrever
e re-
visar
o uso

Bibliografia

- [1] Nadhem J AlFardan et al. “On the security of RC4 in TLS and WPA”. Em: *USENIX Security Symposium*. 2013. URL: <http://www.isg.rhul.ac.uk/tls/>.
- [2] *Amazon Simple Storage Service (Amazon S3) --- Amazon S3 Functionality*. [Online; acessado em 7-outubro-2013]. URL: <http://aws.amazon.com/s3/#functionality>.
- [3] CCP Aporia. *All quiet on the EVE Launcher front?* [Online; acessado em 5-outubro-2013]. 11 de mar. de 2013. URL: <http://community.eveonline.com/news/dev-blogs/74573>.
- [4] Kennon Ballou. *R.I.P. Audiogalaxy*. [Online; acessado em 30-setembro-2013]. 21 de jun. de 2002. URL: <http://www.kuro5hin.org/story/2002/6/21/171321/675>.
- [5] Kenneth P. Birman. *Reliable Distributed Systems: Technologies, Web Services, and Applications*. Secaucus, NJ, USA: Springer-Verlag New York, Inc., 2005, pp. 532–534. ISBN: 0387215093.
- [6] *BitTorrent*. URL: <http://www.bittorrent.com/>.
- [7] *Blizzard Downloader --- Wowpedia*. [Online; acessado em 5-outubro-2013]. 2013. URL: http://wowpedia.org/index.php?title=Blizzard_Downloader&oldid=3198520.
- [8] *CBC to BitTorrent Canada’s Next Great Prime Minister*. URL: <http://archive.is/VYmFD>.
- [9] Juliusz Chroboczek. *BitTorrent DHT library*. [Online; accessed 3-January-2014]. URL: <http://github.com/jech/dht>.
- [10] *Codecon 2002 --- Schedule*. 2002. URL: <http://web.archive.org/web/20021012072819/http://codecon.org/2002/program.html#bittorrent>.
- [11] Bram Cohen. *Incentives Build Robustness in BitTorrent*. 2003.

- [12] T. H. Cormen et al. *Introduction to Algorithms*. 3rd. The MIT Press, 2009. ISBN: 978-0-262-03384-8.
- [13] M. Denters. *Download California Dreaming*. [Online; acessado em 7-outubro-2013]. 8 de nov. de 2010. URL: <http://tegenlicht.vpro.nl/nieuws/2010/november/creative-commons.html>.
- [14] *DGM Live --- FAQ*. [Online; acessado em 7-outubro-2013]. URL: <http://www.dgmlive.com/help.htm#whatisbittorrent>.
- [15] W. Diffie e M. Hellman. “New Directions in Cryptography”. Em: *IEEE Trans. Inf. Theor.* 22.6 (set. de 2006), pp. 644–654. ISSN: 0018-9448. DOI: [10.1109/TIT.1976.1055638](https://doi.org/10.1109/TIT.1976.1055638). URL: <http://dx.doi.org/10.1109/TIT.1976.1055638>.
- [16] Whitfield Diffie e Martin E. Hellman. “Multiuser Cryptographic Techniques”. Em: *Proceedings of the June 7-10, 1976, National Computer Conference and Exposition. AFIPS '76*. New York, NY, USA: ACM, 1976, pp. 109–112. DOI: [10.1145/1499799.1499815](https://doi.org/10.1145/1499799.1499815). URL: <http://doi.acm.org/10.1145/1499799.1499815>.
- [17] *Dr. Dre Raps Napster*. [Online; acessado em 30-setembro-2013]. 18 de abr. de 2000. URL: <http://www.wired.com/techbiz/media/news/2000/04/35749>.
- [18] Ernesto. *BitTorrent And MPAA join forces*. 23 de nov. de 2005. URL: <http://torrentfreak.com/BitTorrent-and-mpaa-join-forces/>.
- [19] Ernesto. *BitTorrent Makes Twitter's Server Deployment 75x Faster*. 16 de jul. de 2013. URL: <http://torrentfreak.com/bittorrent-makes-twitters-server-deployment-75-faster-100716/>.
- [20] Ernesto. *Facebook Uses BitTorrent, and They Love It*. 25 de jun. de 2013. URL: <http://torrentfreak.com/facebook-uses-bittorrent-and-they-love-it-100625/>.
- [21] Ernesto. *Twitter Uses BitTorrent For Server Deployment*. 10 de fev. de 2013. URL: <http://torrentfreak.com/twitter-uses-bittorrent-for-server-deployment-100210/>.
- [22] Ernesto. *UK Government Uses BitTorrent to Share Public Spending Data*. 4 de jun. de 2010. URL: <http://torrentfreak.com/uk-government-uses-bittorrent-to-share-public-spending-data-100604/>.
- [23] Roy T. Fielding et al. *RFC 2616 --- Hypertext Transfer Protocol -- HTTP/1.1*. Jun. de 1999. URL: <http://tools.ietf.org/html/rfc2616>.

- [24] Dennis Fisher. *Attack exploits weakness in RC4 cypher to decrypt user sessions*. 14 de mar. de 2013. URL: <http://threatpost.com/attack-exploits-weakness-rc4-cipher-decrypt-user-sessions-031413>.
- [25] Elle Cayabyab Gitlin. *BitTorrent gets US\$8.75 million in VC money*. 29 de set. de 2005. URL: <http://arstechnica.com/uncategorized/2005/09/5363-2/>.
- [26] Dan Goodin. *SHA1 crypto algorithm underpinning Internet security could fall by 2018*. [Online; accessed 16-January-2014]. 6 de dez. de 2012. URL: <http://arstechnica.com/security/2012/10/sha1-crypto-algorithm-could-fall-by-2018/>.
- [27] Matthew Green. *Attack of the week: RC4 is kind of broken in TLS*. 12 de mar. de 2013. URL: <http://blog.cryptographyengineering.com/2013/03/attack-of-week-rc4-is-kind-of-broken-in.html>.
- [28] Matthew Green. *How (not) to use symmetric encryption*. 1 de dez. de 2011. URL: <http://blog.cryptographyengineering.com/2011/11/how-not-to-use-symmetric-encryption.html>.
- [29] Matthew Green. *What's the deal with RC4?* 15 de dez. de 2011. URL: <http://blog.cryptographyengineering.com/2011/12/whats-deal-with-rc4.html>.
- [30] David Harrison et al. *BitTorrent Enhancement Proposals --- Tracker Peer Obfuscation*. 29 de abr. de 2008. URL: http://www.bittorrent.org/beps/bep_0008.html.
- [31] Daniela Hernandez. *April 13, 2000: Seek and Destroy – Metallica Sues Napster*. [Online; acessado em 30-setembro-2013]. 13 de abr. de 2012. URL: <http://www.wired.com/thisdayintech/2012/04/april-13-2000-seek-and-destroy-metallica-sues-napster/>.
- [32] *HPC Data Repository*. [Online; acessado em 7-outubro-2013]. URL: http://www.hpc.fsu.edu/index.php?option=com_wrapper&view=wrapper&Itemid=80.
- [33] Sandvine Inc. *Global Internet Phenomena Report --- 1H 2013*. Online; retirado de http://macaubas.com/wp-content/uploads/2013/05/Sandvine_Global_Internet_Phenomena_Report_1H_2013.pdf. 2013. URL: <https://www.sandvine.com/downloads/general/global-internet-phenomena/2013/sandvine-global-internet-phenomena-report-1h-2013.pdf>.
- [34] *Internet Archive Wayback Machine --- Cypherpunks Mailing List*. [Online; accessed 20-January-2014]. 91994. URL: <http://web.archive.org/web/20080404222417/http://cypherpunks.venona.com/date/1994/09/msg00304.html>.

- [35] Christopher Jones. *Metallica Rips Napster*. [Online; acessado em 30-setembro-2013]. 13 de abr. de 2000. URL: <http://www.wired.com/politics/law/news/2000/04/35670>.
- [36] David Kravets. *Dec. 7, 1999: RIAA Sues Napster*. [Online; acessado em 30-setembro-2013]. 7 de dez. de 2009. URL: <http://www.wired.com/thisdayintech/2009/12/1207riaa-sues-napster/>.
- [37] J.F. Kurose e K.W. Ross. *Computer Networking: A Top-Down Approach*. Pearson Education, Limited, 2010. ISBN: 9780131365483. URL: <http://books.google.com.br/books?id=2hv3PgAACAAJ>.
- [38] M. Lehmann et al. “Swarming: como BitTorrent revolucionou a Internet”. Em: *Atualizações em Informática*. Ed. por PUC-Rio. Vol. 1. [Online; accessed 23-outubro-2013]. Rio de Janeiro, 2011. Cap. 6, pp. 209–258. URL: <http://www.lbd.dcc.ufmg.br/colecoes/jai/2012/006.pdf>.
- [39] John Leyden. *That earth-shattering NSA crypto-cracking: Have spooks smashed RC4?* 6 de set. de 2013. URL: http://www.theregister.co.uk/2013/09/06/nsa_cryptobreaking_bullrun_analysis.
- [40] Andrew Loewenstern e Arvid Norberg. *BitTorrent Enhancement Proposals --- DHT Protocol*. 29 de fev. de 2008. URL: http://www.bittorrent.org/beps/bep_0005.html.
- [41] Petar Maymounkov e David Mazières. “Kademlia: A Peer-to-Peer Information System Based on the XOR Metric”. Em: (). [Online; acessado em 30-December-2013], p. 6. URL: <http://pdos.csail.mit.edu/~petar/papers/maymounkov-kademlia.pdf>.
- [42] Petar Maymounkov e David Mazières. “Kademlia: A Peer-to-Peer Information System Based on the XOR Metric”. Em: *Revised Papers from the First International Workshop on Peer-to-Peer Systems*. IPTPS '01. London, UK, UK: Springer-Verlag, 2002, pp. 53–65. ISBN: 3-540-44179-4. URL: <http://dl.acm.org/citation.cfm?id=646334.687801>.
- [43] Alfred J. Menezes, Scott A. Vanstone e Paul C. Van Oorschot. *Handbook of Applied Cryptography*. 1st. Boca Raton, FL, USA: CRC Press, Inc., 1996. ISBN: 0849385237.
- [44] *Miro*. [Online; acessado em 7-outubro-2013]. URL: <http://getmiro.com>.
- [45] *Moving Image Archive - Night of the Living Dead (1968)*. [Online; acessado em 16-outubro-2013]. URL: http://archive.org/details/night_of_the_living_dead.

- [46] *Napster: 20 million users*. [Online; acessado em 30-setembro-2013]. 19 de jul. de 2000. URL: <http://cnnfn.cnn.com/2000/07/19/technology/napster/index.htm>.
- [47] Arvid Norberg. *BitTorrent Tech Talks: DHT*. 22 de jan. de 2013. URL: <http://engineering.bittorrent.com/2013/01/22/bittorrent-tech-talks-dht/>.
- [48] Arvid Norberg, Ludvig Strigeus e Greg Hazel. *BitTorrent Enhancement Proposals --- Extension Protocol*. 28 de fev. de 2008. URL: http://www.bittorrent.org/beps/bep_0010.html.
- [49] *NRKbeta*. [Online; acessado em 7-outubro-2013]. URL: <http://nrkbeta.no/bittorrent/>.
- [50] *OpenSSL Cryptography and SSL/TLS Toolkit*. [Online; accessed 17-January-2014]. URL: <http://www.openssl.org/>.
- [51] C. Greg Plaxton, Rajmohan Rajaraman e Andréa W. Richa. “Accessing Nearby Copies of Replicated Objects in a Distributed Environment”. Em: *Proceedings of the Ninth Annual ACM Symposium on Parallel Algorithms and Architectures*. SPAA '97. Newport, Rhode Island, New York, NY, USA: ACM, 1997, pp. 311–320. ISBN: 0-89791-890-8. DOI: [10.1145/258492.258523](http://doi.acm.org/10.1145/258492.258523). URL: <http://doi.acm.org/10.1145/258492.258523>.
- [52] *Press Release: Global Napster usage plummets, but new file-sharing alternatives gaining ground, reports Jupiter Media Metrix*. [Online; acessado em 8-outubro-2013]. Out. de 2013. URL: <http://www.lse.ac.uk/media@lse/documents/MPP/LSE-MPP-Policy-Brief-9-Copyright-and-Creation.pdf>.
- [53] *Rhapsody.com*. [Online; acessado em 30-setembro-2013]. URL: <http://www.rhapsody.com>.
- [54] Vincent Rijmen. *Current Status of SHA-1*. Rel. téc. [Online; accessed 16-January-2014]. Rundfunk und Telekom Regulierungs-GmbH, Austria, fev. de 2007. URL: <https://www.signatur.rtr.at/repository/rtr-sha1-20070221-en.pdf>.
- [55] R. Rivest. *RPF1320: The MD4 Message-Digest Algorithm*. Rel. téc. 1320. Internet Engineering Task Force, abr. de 1992. URL: <http://www.ietf.org/rfc/rfc1320.txt>.
- [56] Stefan Saroiu, P. Krishna Gummadi e Steven D. Gribble. “A Measurement Study of Peer-to-Peer File Sharing Systems”. Em: 2002.
- [57] Peter W. Shor. “Polynomial-Time Algorithms for Prime Factorization and Discrete Logarithms on a Quantum Computer”. Em: *SIAM J. Comput.* 26.5 (out. de 1997), pp. 1484–1509. ISSN: 0097-5397. DOI: [10.1137/S0097539795293172](http://dx.doi.org/10.1137/S0097539795293172). URL: <http://dx.doi.org/10.1137/S0097539795293172>.

- [58] Clive Thompson. *The BitTorrent Effect*. Jan. de 2005. URL: <http://www.wired.com/wired/archive/13.01/bittorrent.html>.
- [59] “Um pouco de história: redes P2P de compartilhamento de arquivos”. Em: *Revista PnP* 10 (out. de 2008). [Online; retirado de http://www.thecnica.com/artigos/PnP_10_02.pdf], p. 12. URL: http://www.revistapnp.com.br/pnp_10.php.
- [60] *VODO --- About*. [Online; acessado em 7-outubro-2013]. URL: <http://www.dgmlive.com/help.htm#whatisbittorrent>.
- [61] Michael Welzl. *Peer to Peer Systems --- Structured P2P file sharing systems*. [Online; accessed 17-December-2013]. University of Innsbruck, Austria. URL: <http://heim.ifi.uio.no/michawe/teaching/p2p-ws08/p2p-4-6.pdf>.
- [62] Wikibooks. *The World of Peer-to-Peer (P2P) --- Wikibooks, The Free Textbook Project*. [Online; acessado em 3-outubro-2013]. 2012. URL: [http://en.wikibooks.org/w/index.php?title=The_World_of_Peer-to-Peer_\(P2P\)&oldid=2316492](http://en.wikibooks.org/w/index.php?title=The_World_of_Peer-to-Peer_(P2P)&oldid=2316492).
- [63] Wikipedia. *A&M Records, Inc. v. Napster, Inc. --- Wikipedia, The Free Encyclopedia*. [Online; acessado em 2-dezembro-2013]. 2013. URL: http://en.wikipedia.org/w/index.php?title=A%26M_Records,_Inc._v._Napster,_Inc.&oldid=579733684.
- [64] Wikipedia. *Anycast --- Wikipedia, The Free Encyclopedia*. [Online; acessado em 2-outubro-2013]. 2013. URL: <http://en.wikipedia.org/w/index.php?title=Anycast&oldid=574680440>.
- [65] Wikipedia. *Audiogalaxy --- Wikipedia, The Free Encyclopedia*. [Online; acessado em 29-setembro-2013]. 2013. URL: <http://en.wikipedia.org/w/index.php?title=Audiogalaxy&oldid=560950036>.
- [66] Wikipedia. *BitTorrent --- Wikipedia, The Free Encyclopedia*. [Online; accessed 28-October-2013]. 2013. URL: <http://en.wikipedia.org/w/index.php?title=BitTorrent&oldid=578877679>.
- [67] Wikipedia. *Bram Cohen --- Wikipedia, The Free Encyclopedia*. [Online; acessado em 7-outubro-2013]. 2013. URL: http://en.wikipedia.org/w/index.php?title=Bram_Cohen&oldid=574084830.
- [68] Wikipedia. *Cryptography --- Wikipedia, The Free Encyclopedia*. [Online; accessed 19-January-2014]. 2014. URL: <http://en.wikipedia.org/w/index.php?title=Cryptography&oldid=591122478>.

- [69] Wikipedia. *EDonkey network* --- *Wikipedia, The Free Encyclopedia*. [Online; acessado em 3-outubro-2013]. 2013. URL: http://en.wikipedia.org/w/index.php?title=EDonkey_network&oldid=568576016.
- [70] Wikipedia. *File sharing* --- *Wikipedia, The Free Encyclopedia*. [Online; acessado em maio de 2013]. 2013. URL: http://en.wikipedia.org/w/index.php?title=File_sharing&oldid=556034682.
- [71] Wikipedia. *Gnutella2* --- *Wikipedia, The Free Encyclopedia*. [Online; acessado em 2-outubro-2013]. 2013. URL: <http://en.wikipedia.org/w/index.php?title=Gnutella2&oldid=556794729>.
- [72] Wikipedia. *Gnutella* --- *Wikipedia, The Free Encyclopedia*. [Online; acessado em 2-outubro-2013]. 2013. URL: <http://en.wikipedia.org/w/index.php?title=Gnutella&oldid=574304390>.
- [73] Wikipedia. *Hash function* --- *Wikipedia, The Free Encyclopedia*. [Online; acessado em 5-outubro-2013]. 2013. URL: http://en.wikipedia.org/w/index.php?title=Hash_function&oldid=574871670.
- [74] Wikipedia. *Hash table* --- *Wikipedia, The Free Encyclopedia*. [Online; acessado em 5-outubro-2013]. 2013. URL: http://en.wikipedia.org/w/index.php?title=Hash_table&oldid=575499828.
- [75] Wikipedia. *Internet service provider* --- *Wikipedia, The Free Encyclopedia*. [Online; acessado em 29-setembro-2013]. 2013. URL: http://en.wikipedia.org/w/index.php?title=Internet_service_provider&oldid=573549991.
- [76] Wikipedia. *Kademlia* --- *Wikipedia, The Free Encyclopedia*. [Online; acessado em 3-outubro-2013]. 2013. URL: <http://en.wikipedia.org/w/index.php?title=Kademlia&oldid=575742258>.
- [77] Wikipedia. *Magic cookie* --- *Wikipedia, The Free Encyclopedia*. [Online; accessed 31-December-2013]. 2013. URL: http://en.wikipedia.org/w/index.php?title=Magic_cookie&oldid=568707666.
- [78] Wikipedia. *Motion Picture Association of America* --- *Wikipedia, The Free Encyclopedia*. [Online; acessado em 7-outubro-2013]. 2013. URL: http://en.wikipedia.org/w/index.php?title=Motion_Picture_Association_of_America&oldid=575124240.

- [79] Wikipedia. *MP3.com* --- *Wikipedia, The Free Encyclopedia*. [Online; acessado em 29-setembro-2013]. 2013. URL: <http://en.wikipedia.org/w/index.php?title=MP3.com&oldid=571025541>.
- [80] Wikipedia. *MP3* --- *Wikipedia, The Free Encyclopedia*. [Online; acessado em 29-setembro-2013]. 2013. URL: <http://en.wikipedia.org/w/index.php?title=MP3&oldid=574123988>.
- [81] Wikipedia. *Napster* --- *Wikipedia, The Free Encyclopedia*. [Online; acessado em 30-setembro-2013]. 2013. URL: <http://en.wikipedia.org/w/index.php?title=Napster&oldid=573999770>.
- [82] Wikipedia. *Pareto efficiency* --- *Wikipedia, The Free Encyclopedia*. [Online; accessed 12-January-2014]. 2014. URL: http://en.wikipedia.org/w/index.php?title=Pareto_efficiency&oldid=589518713.
- [83] Wikipedia. *Percent-encoding* --- *Wikipedia, The Free Encyclopedia*. [Online; accessed 27-October-2013]. 2013. URL: <http://en.wikipedia.org/w/index.php?title=Percent-encoding&oldid=573113056>.
- [84] Wikipedia. *Recording Industry Association of America* --- *Wikipedia, The Free Encyclopedia*. [Online; acessado em 30-setembro-2013]. 2013. URL: http://en.wikipedia.org/w/index.php?title=Recording_Industry_Association_of_America&oldid=574192660.
- [85] Wikipedia. *Segmented file transfer* --- *Wikipedia, The Free Encyclopedia*. [Online; acessado em 5-outubro-2013]. 2013. URL: http://en.wikipedia.org/w/index.php?title=Segmented_file_transfer&oldid=533100656.
- [86] Wikipedia. *Timeline of file sharing* --- *Wikipedia, The Free Encyclopedia*. [Online; acessado em 28 de setembro de 2013]. 2013. URL: http://en.wikipedia.org/w/index.php?title=Timeline_of_file_sharing&oldid=571061187.
- [87] Wikipédia. *BitTorrent* --- *Wikipédia, a enciclopédia livre*. [Online; accessed 22-outubro-2013]. 2013. URL: <http://pt.wikipedia.org/w/index.php?title=BitTorrent&oldid=36953538>.
- [88] Theory.org Wiki. *BitTorrentPeerExchangeConventions* --- *Theory.org Wiki*, [Online; accessed 3-January-2014]. 2 de mar. de 2013. URL: <https://wiki.theory.org/index.php?title=BitTorrentPeerExchangeConventions&oldid=2223>.

- [89] Theory.org Wiki. *BitTorrentSpecification: Bencoding* --- *Theory.org Wiki*, [Online; accessed 17-October-2013]. 2013. URL: <https://wiki.theory.org/index.php?title=BitTorrentSpecification&oldid=2527#Bencoding>.
- [90] Theory.org Wiki. *BitTorrentSpecification: Reserved Bytes* --- *Theory.org Wiki*, [Online; accessed 6-January-2014]. 2013. URL: https://wiki.theory.org/index.php?title=BitTorrentSpecification&oldid=2527#Reserved_Bytes.
- [91] Theory.org Wiki. *BitTorrentSpecification* --- *Theory.org Wiki*, [Online; accessed 6-January-2014]. 2013. URL: <https://wiki.theory.org/BitTorrentSpecification>.
- [92] Theory.org Wiki. *BitTorrentSpecification: Tracker Response* --- *Theory.org Wiki*, [Online; accessed 17-October-2013]. 2013. URL: https://wiki.theory.org/index.php?title=BitTorrentSpecification&oldid=2527#Tracker_Response.
- [93] Vuze Wiki. *BMessage Stream Encryption* --- *Vuze Wiki*, [Online; accessed 20-January-2014]. 2014. URL: http://wiki.vuze.com/w/Message_Stream_Encryption#Message_Stream_Encryption_.28aka_PHE.29_format_specification.
- [94] Tao Xie, Fanbao Liu e Dengguo Feng. *Fast Collision Attack on MD5*. Cryptology ePrint Archive, Report 2013/170. [Online; accessed 17-January-2014]. 2013. URL: <http://eprint.iacr.org/2013/170.pdf>.

Visão Pessoal