Anatomia do BitTorrent

a Ciência da Computação no Transmission

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 Figuras

Lista de tarefas pendentes

Sobre este trabalho

Algumas observações devem ser feitas sobre este trabalho, para conhecimento antes da leitura.

Cores no texto

Em alguns momentos percebeu-se que poderia haver confusão semântica. Na tentativa de se resolver isso, um padrão de escrita foi adotado para o trabalho e, utilizando cores, dividiu-se em duas funções semânticas;

- `texto`: foi usado quando o significado do trecho destacado era de conteúdo não processado pelo programa, ou seja, como foi recebido; e
- `texto`: usado quando o texto destacado já foi processado, já sendo na forma de sequência de caracteres (*string*).

Essa diferença é notada na seção que trata de dicionários bencoded (capítulo 3, página 14), quando são mostrados conteúdos que representam um dicionário em duas formas diferentes: enquanto `d3:foo3:bar6:foobar6:bazbare` é o que se recebe em uma mensagem, possui representação diferente no computador após ser processado, que é `{"foo": "bar", "foobar": "bazbar"}`.

Termos em inglês

Foi preferido o uso dos termos técnicos de BitTorrent em inglês às suas traduções, para que o usuário se habitue com os originais, que são bastante utilizados na área. Por isso, estes não aparecem em itálico.

Para os outros termos, são escritos em *itálico*.

Trechos de código e comentários

Como o objetivo deste trabalho é apresentar código da linguagem C usado no Transmission, em alguns momentos, o código original que seria mostrado não era muito legível. Por causa disso, eles foram modificados apenas para melhorar sua ilustração, não perdendo funcionalidade. Essas alterações envolveram omissões de trechos de código irrelevantes (por exemplo, verificação de erros) e trocas de valores, pré-definidos como constantes, para seus valores absolutos.

Além disso, alguns comentários originais foram mantidos para garantir a essência do código apresentado. Todos esses originais estão comentados entre `/* ... */`.

Porém, existem momentos em que a leitura do código do Transmission não é suficiente, e, nesses casos, foram feitos comentários extras usando `// ...`.

```
1    /* Comentarios originais do codigo do Transmission. */
2
3    // comentario de codigo
4    // Comentarios extras adicionados posteriormente.
```

Capítulo 1

Introdução

Desde o início da história da computação, o compartilhamento de dados é uma ação naturalmente necessária, e que passou a ser mais comum com a criação dos dispositivos de armazenamento de dados sob a forma de arquivos, com a Internet anos depois e, mais recentemente, com a computação em nuvem.

A partir de 1999, com o surgimento do Napster, as redes peer-to-peer (P2P) passaram a ser mais populares, sendo frequentemente utilizadas para transferir dados. Essas redes têm como característica principal computadores transferindo dados entre si, ou seja, não existindo funções fixas de fonte e de consumo de dados, mas sim de ambas essas funções.

A comunicação P2P veio se desenvolvendo ao longo dos anos. Em 2003, esse desenvolvimento teve um grande impulso, quando Bram Cohen propôs o protocolo BitTorrent, lançando-o juntamente com um programa cliente, e incentivando o seu uso por "testadores" mediante o compartilhamento de material pornográfico. Com isso, pôde melhorar o seu funcionamento, se tornando popular rapidamente através de seus usuários.

Desde então, muitos programas de compartilhamento BitTorrent passaram a ser desenvolvidos, e o protocolo começou a ser estudado pela área acadêmica, passando por melhorias. Em 2013, o protocolo foi o responsável por aproximadamente 10% do tráfego total de Internet nos Estados Unidos [54], se tornando uma das formas mais eficientes e utilizadas de se compartilhar arquivos via Internet atualmente.

O BitTorrent contém conceitos de diversos tópicos em Ciência da Computação, tais como teoria dos jogos, estruturas de dados, tabelas de dispersão, etc, contando ainda com diversos estudos acadêmicos sobre topologias de rede formadas e otimizações de redes e algoritmos, por exemplo.

Neste trabalho, estudamos o protocolo BitTorrent, analisando em profundidade a sua apli-

cação pelo programa cliente Transmission [95], particularmente interessados nos elementos de Ciência de Computação presentes no código. Este texto contém uma descrição desses vários elementos encontrados, junto com trechos de código que consideramos ilustrativos para exemplicá-los.

No final, relacionamos os conceitos de Ciência da Computação encontrados no Transmission, e onde eles aparecem, e também se tais conceitos aparecem na grade curricular do Bacharelado em Ciência da Computação.

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, no fim da década dos anos 1990.

2.1 Período pré-torrent

Entre o final dos anos 80 e o início dos 90 [109, 126], 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 à Internet (ISPs) [114]. Com o advento do formato de áudio MP3 (MP3) [119], 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 [118], que era um site de divulgação de bandas independentes, e o Audiogalaxy.com [97, 104]. 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 de 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 três anos seguintes à criação desses dois sites foram muito produtivos ao mundo das redes peer-to-peer 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 [120], 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 dois 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 [60]. Em abril de 2000, foi a vez da banda Metallica também processá-lo, como retaliação à descoberta de que uma música ainda não lançada oficialmente já circulava na rede [51, 59]. 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 [31]. Isso fez com que o Napster recebesse atenção da mídia, ganhando popularidade e atingindo os 20 milhões de usuários em meados do ano 2000 [72].

Em 2001, esses imbróglios judiciais resultaram numa liminar federal que ordenou que o Napster retirasse o conteúdo protegido pelas 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 esta não conseguisse controlar o conteúdo que trafegava ali [120]. Em setembro, o Napster fez um acordo [102], 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 aos 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 [82].

O sucesso do Napster, mesmo que por curto período de 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 do Napster foi o Gnutella, que, em março de 2000 [111], surgiu como uma resposta de domínio público, feita com "gambiarras", 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 [14] , mas sem a limitação de ser 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 [110], 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 de melhoria no desempenho das pesquisas, as 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 [22] e, na mesma época, começou a testá-lo

usando como chamariz uma coleção de material pornográfico para atrair beta testers [94]. Assim, o software começou a ser usado imediatamente.

Nesse meio tempo, Bram ainda passou pela Valve [106], 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 [32, 117] 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 [15], 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 [70], 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 [27];
- VODO [99] 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 [20] e a holandesa VPRO [26], já disponibilizaram programas de sua grade para download. A norueguesa NRK o faz para conteúdos em HD [76] 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 [4], e distribuir os jogos e suas eventuais atualizações [17], respectivamente;
- o governo britânico distribuiu os detalhes de seus gastos [38], enquanto a Universidade do Estado da Flórida o utiliza para transmitir grandes conjuntos de dados científicos aos seus pesquisadores [53];
- Facebook [35] e Twitter [37] o usam para atualizar os seus sites, enviando de forma eficiente o código novo para seus servidores de aplicação [33].

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 [54].

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 compatilham 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 [80].

Estudos acadêmicos

Academicamente, o protocolo é bastante estudado desde o seu surgimento, sendo foco de pesquisas 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

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 contendo 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 mantém listas atualizadas dos peers que estão compartilhando aqueles arquivos, atualizadas em curtos períodos de tempo (usualmente trinta 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.

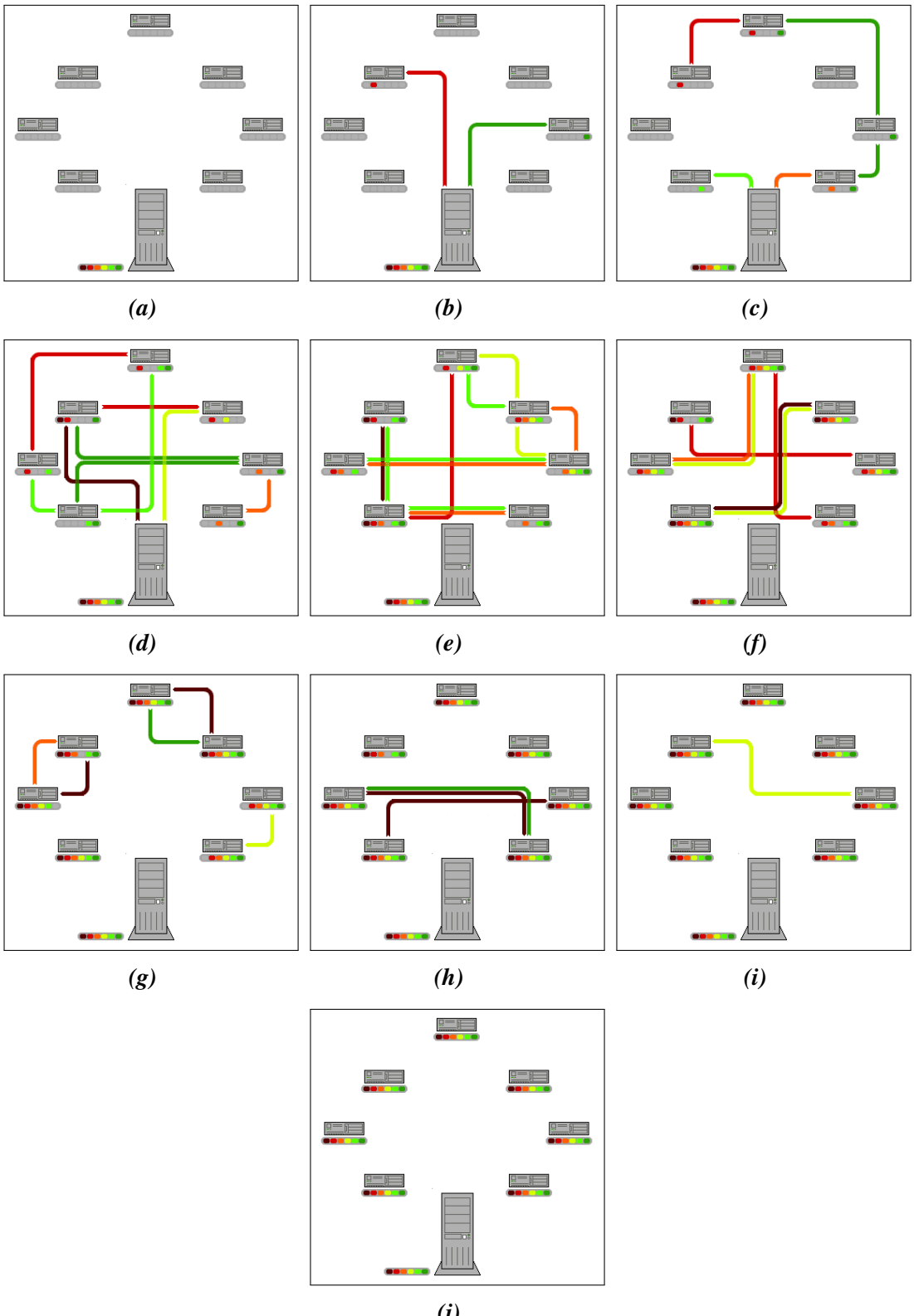

Figura 3.1: *simulação de uma transferência torrent: o seeder, na parte inferior das figuras, possui todas as cinco partes de um arquivo, que os outros computadores - os leechers - baixam de forma independente e paralela. Fonte: [128]*

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, os arquivo .torrent contêm informações sobre um único torrent (dentre elas o seu identificador único), 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 dividí-los em vários swarms, para fins de escalabilidade da rede formada.

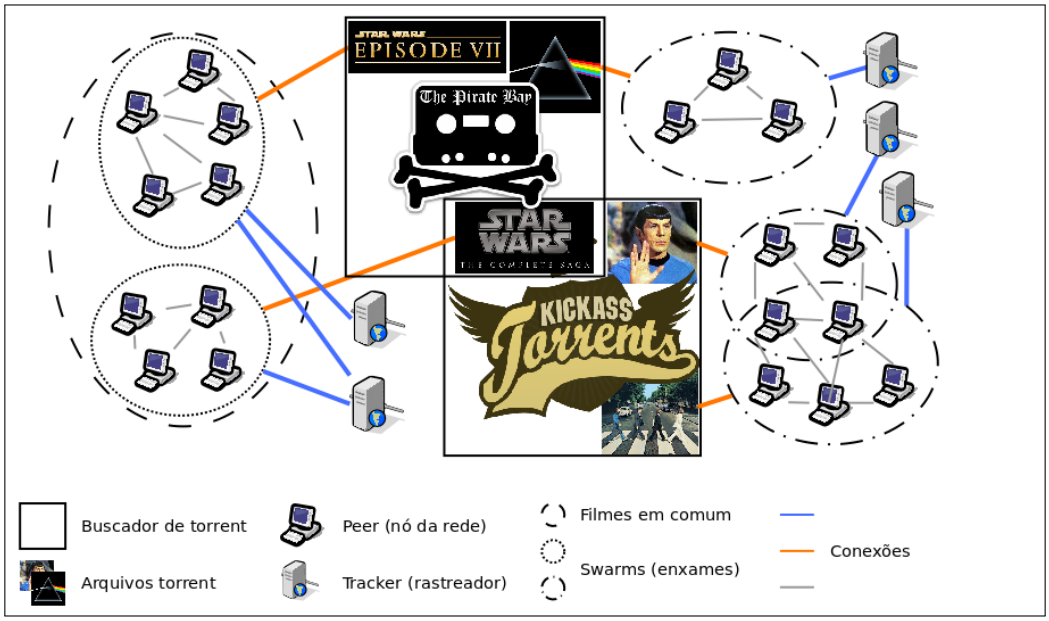

Figura 3.2: *amostra de uma rede de conexões BitTorrent*

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 [71], que é de domínio público e livre de direitos autorais.

Se abrirmos esse arquivo, veremos uma grande 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-listll36:http://bt1.
2  archive.org:6969/announceel36:http://bt2.archive.org:6969/announceee7:comment13:crea
3  tiondatei1343715473e4:infod5:filesld5:crc328:030208fe6:lengthi4127671704e3:md532:627
4  f5a428f9e454ccfcb29d31b87169a5:mtime10:10794024804:pathl29: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-listl28:http://archive.org/
11 download/39:http://ia600301.us.archive.org/22/items39:http://ia700301.us.archive.
12 org/22/itemsee
```

Conteúdo textual 3.1: *trecho do conteúdo do arquivo .torrent do filme "A Noite dos Mortos Vivos", de 1960 [71], com a parte binária truncada*

Esse conteúdo está organizado pela 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 [130]:

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 podem ser negativo), terminados por um `e`. Por exemplo, na linha 11, `i1343715473e` corresponde ao número `1343715473`.

listas são formadas por `l` (L minúsculo), seguidas por seus elementos (também no formato bencode), e então terminadas 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, `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"]}`.

```
d
    8:announce
    36:http://bt1.archive.org:6969/announce
    13:announce-list
    l
        l36:http://bt1.archive.org:6969/announcee
        l36:http://bt2.archive.org:6969/announcee
    e
    7:comment
    13:creation date
    i1343715473e
    4:info
    d
        5:files
        l
            d
                5:crc32
                8:030208fe
                6:length
                i4127671704e
                3:md5
                32:627f5a428f9e454ccfcb29d31b87169a
                5:mtime
                10:1079402480
                4:path
                l29:night_of_the_living_dead.mpege
                4:sha1
                40:5e44bb1b3f700240249a5287c64dc02dc56d034b
            e
        e
        4:name
        24:night_of_the_living_dead
        12:piece length
        i4194304e
        6:pieces
        23720:<binary>
    e
    6:locale
    2:en
    5:title
    24:night_of_the_living_dead
    8:url-list
    l
        28:http://archive.org/download/
        39:http://ia600301.us.archive.org/22/items
        39:http://ia700301.us.archive.org/22/items
    e
e
```

Conteúdo textual 3.2: *trechos formatados de forma legível do conteúdo do arquivo .torrent do filme "A Noite dos Mortos Vivos", de 1960 [71], 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 total de dados de um torrent. De tamanhos geralmente sendo potências de 2 definidos, para cada torrent, eles são escolhidos de forma a se otimizar o processo de transferência: partes muito grandes causariam ineficiência de banda de rede, enquanto partes muito pequenas aumentariam os trechos de valores hash nos arquivos .torrent.

Todas as partes possuem tamanhos iguais, exceto a parte final, que pode ser menor. No caso de um torrent de múltiplos arquivos, para efeito de divisão em partes, ele é considerado como um único arquivo contíguo, composto pela concatenação de todos esses arquivos que o compõem, na ordem em que forem listados no arquivo .torrent. Dessa forma, é possivel 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 $\lceil \frac{\text{tamanho total}}{\text{tamanho das partes}} \rceil$.

De toda essa sequência de partes, existirá uma sequência de valores hash SHA-1 de seus dados, formando uma grande string de um único valor 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, a preferência é manter-se 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, do filme “A Noite dos Mortos Vivos”, o site de origem do arquivo .torrent que estamos usando não fornece um magnet link oficialmente. Porém, o Transmission consegue construir um 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*)) [123], foi o seguinte:

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

Conteúdo textual 3.3: *link magnético do arquivo .torrent do filme "A Noite dos Mortos Vivos", de 1960 [71], com parâmetros divididos entre linhas para melhor visualização*

Esse endereço é composto por vários pares, constituídos 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; e

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 [105].

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 haja necessidade de se 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.

./libtransmission/metainfo.c:367

```
static const char*
tr_metainfoParseImpl(const tr_session * session, tr_info * inf,
bool * hasInfoDict, int * infoDictLength, const tr_variant * meta) {
  // variáveis temporárias
  int64_t i; size_t len; const char * str; const uint8_t * raw;
  tr_variant * infoDict = NULL;

  // flags
  bool b, isMagnet = false;

  /* info_hash: urlencoded 20-byte SHA1 hash of the value of the info key from the
   * Metainfo file. (...) */
  b = tr_variantDictFindDict(meta, TR_KEY_info, &infoDict);
  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`.

./libtransmission/metainfo.c:367

```
  if (!b) { // Não possui a chave "info" no dicionário.
    // "Será que não é um magnet link?"
    if (tr_variantDictFindDict(meta, TR_KEY_magnet_info, &d)) {
        isMagnet = true;

        if (!tr_variantDictFindRaw(d, TR_KEY_info_hash, &raw, &len) ||
            len != SHA_DIGEST_LENGTH) // Se tiver a chave "info_hash" válida, a usa.
          return "info_hash";

        memcpy(inf->hash, raw, len);
        tr_sha1_to_hex(inf->hashString, inf->hash);
        (...)
    }
    else return "info"; // Não é magnet link e não possui a chave "info".
  }
```

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.

./libtransmission/metainfo.c:367

```
  else {
    int len;
    char * bstr = tr_variantToStr(infoDict, TR_VARIANT_FMT_BENC, &len);
    tr_sha1(inf->hash, bstr, len, NULL);          // Calcula o hash SHA-1 do .torrent...
    tr_sha1_to_hex(inf->hashString, inf->hash);   // ...e converte de base2 para base16.
    (...)
  }
```

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 endereços web de contato do tracker (*announces*).

./libtransmission/metainfo.c:367

```
  if (!tr_variantDictFindInt(infoDict, TR_KEY_private, &i)) // privacidade do torrent
    if (!tr_variantDictFindInt(meta, TR_KEY_private, &i)) i = 0;
  inf->isPrivate = i != 0;

  if (!isMagnet) { // quantidade de partes
    if (!tr_variantDictFindInt(infoDict, TR_KEY_piece_length, &i) || (i < 1))
      return "piece length";
    inf->pieceSize = i;
  }

  if (!isMagnet) { // hashes das partes
    if (!tr_variantDictFindRaw(infoDict, TR_KEY_pieces, &raw, &len) ||
        len % SHA_DIGEST_LENGTH) return "pieces";

    inf->pieceCount = len / SHA_DIGEST_LENGTH;
    inf->pieces = tr_new0(tr_piece, inf->pieceCount);
    for (i = 0; i < inf->pieceCount; i++)
      memcpy(inf->pieces[i].hash, &raw[i * SHA_DIGEST_LENGTH], SHA_DIGEST_LENGTH);
  }

  if (!isMagnet) { // lista de arquivos
    if ((str = parseFiles(inf, tr_variantDictFind(infoDict, TR_KEY_files),
                               tr_variantDictFind(infoDict, TR_KEY_length))))
      return str;

    if (!inf->fileCount || !inf->totalSize ||
        (uint64_t) inf->pieceCount != (inf->totalSize+inf->pieceSize-1)/inf->pieceSize)
      return "files";
  }

  if ((str = getannounce(inf, meta))) return str; // announce(s)
  (...)
  return NULL;
}
```

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 [133].

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 o meio 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, transmitidos desde o momento em que o cliente enviou o primeiro aviso ao tracker;

downloaded: a quantidade total de dados, em bytes, recebidos desde o 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 passar 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 serão o host e os 2 bytes finais serão 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, desse modo, 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; e
- *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 aguardando (ou "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 números 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 na mesma sub-rede (no mesmo lado de um roteador) — com tradução de endereço de rede (*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; e

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

./libtransmission/announcer-common.h:127

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

./libtransmission/announcer.c:1200

```
static void announce_request_delegate(tr_announcer * announcer,
    tr_announce_request * request, tr_announce_response_func * callback,
    void * callback_data) {
  (...)
  if (!memcmp(request->url, "http", 4))        // announce começa com "http://"
    tr_tracker_http_announce(session, request, callback, callback_data);
  else if (!memcmp(request->url, "udp://", 6)) // announce começa com "udp://"
    tr_tracker_udp_announce(session, request, callback, callback_data);
  else
    tr_logAddError("Unsupported url: %s", request->url);
  (...)
}
```

Como resposta, é recebido 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 a causa da falha da requisição;

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 emtre 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 junto às 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; e

peers : pode ser uma das seguintes 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 descrita 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); e
   - **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*);

```
* About to connect() to exodus.desync.com port 6969 (#8)
*   Trying 208.83.20.164...
*
* Connected to exodus.desync.com (208.83.20.164) port 6969 (#8)
> 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
406288292&numwant=80&key=6ee99240&compact=1&supportcrypto=1&requirecrypto=1&event=
started HTTP/1.1
User-Agent: Transmission/2.82
Host: exodus.desync.com:6969
Accept: */*
Accept-Encoding: gzip;q=1.0, deflate, identity

< HTTP/1.1 200 OK
< Content-Type: text/plain
< Content-Length: 136
<
* Connection #8 to host exodus.desync.com left intact
Announce response:
< {
    "complete": 1,
    "downloaded": 11,
    "incomplete": 6,
    "interval": 1732,
    "min interval": 866,
    "peers": \"<binary>\"
}
```

Conteúdo textual 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 o peer passou a ser um seeder; e
- quando o peer 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 em suas 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 a chave:

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); e
- **name** (opcional): o nome interno do torrent, como especificado pelo respectivo arquivo .torrent, na sua seção `info` .

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

./libtransmission/announcer-common.h:37

```
typedef struct {
  // lista de info_hash de torrents para serem pesquisados por scrape
  uint8_t info_hash[TR_MULTISCRAPE_MAX][SHA_DIGEST_LENGTH];
  char * url;  // URL de scrape
  (...)
} tr_scrape_request;

struct tr_scrape_response_row { // usado para conter os dados respondidos do scrape
  uint8_t info_hash[SHA_DIGEST_LENGTH]; // info_hash do torrent
  int seeders;     // qtd de peers que são seeders deste torrent
  int leechers;    // qtd de peers baixando este torrent
  int downloads;   // qtd de vezes que este torrent foi baixado
  int downloaders; // qtd de leechers ativos no swarm (suportado por alguns trackers)
};
```

./libtransmission/announcer.c:1397

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

```
* About to connect() to www.mvgroup.org port 2710 (#1)
*   Trying 88.129.153.50...
* Connected to www.mvgroup.org (88.129.153.50) port 2710 (#1)
> GET /scrape?info_hash=\%83F\%24\%b62\%e5Q\%cc4\%10h\%ba\%1e8\%e2C\%f7\%80\%01\%87 HTTP/1.1
User-Agent: Transmission/2.82
Host: www.mvgroup.org:2710
Accept: */*
Accept-Encoding: gzip;q=1.0, deflate, identity

* HTTP 1.0, assume close after body
< HTTP/1.0 200 OK
<
* Closing connection 1
Scrape response:
< {
    "files": {
        \"<binary>\": {
            "complete": 9,
            "downloaded": 59074,
            "incomplete": 2
        }
    }
}
```

Conteúdo textual 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 de que o tracker não segue a convenção; e
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`; e
    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; e
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.

./libtransmission/announcer-http.c:288

```
void tr_tracker_http_announce(tr_session * session, const tr_announce_request * request,
        tr_announce_response_func response_func, void * response_func_user_data) {
    char * url = announce_url_new(session, request);
    struct announce_data * d = tr_new0(struct announce_data, 1);
    (...) // preenchimento dos campos da variável 'd'

    tr_webRun(session, url, on_announce_done, d);
    (...)
}
```

./libtransmission/announcer-http.c:189

```
static void on_announce_done(tr_session * session, bool did_connect, bool did_timeout,
    long response_code, const void * msg, size_t msglen, void * vdata) {
    tr_announce_response * response;
    struct announce_data * data = vdata;
    response = &data->response;
    (...)

    // lê os dados da resposta do tracker
    if (variant_loaded && tr_variantIsDict(&benc)) {
        int64_t i; size_t len; tr_variant * tmp; // variáveis ...
        const char * str; const uint8_t * raw;   // ... temporárias

        if (tr_variantDictFindStr(&benc, TR_KEY_failure_reason, &str, &len))
            response->errmsg = tr_strndup(str, len);

        if (tr_variantDictFindStr(&benc, TR_KEY_tracker_id, &str, &len))
            response->tracker_id_str = tr_strndup(str, len);

        if (tr_variantDictFindInt(&benc, TR_KEY_complete, &i))
            response->seeders = i;

        (...)

        if (tr_variantDictFindRaw(&benc, TR_KEY_peers, &raw, &len)) {
            response->pex = tr_peerMgrCompactToPex(raw, len,
            NULL, 0, &response->pex_count);
        }
        else if (tr_variantDictFindList(&benc, TR_KEY_peers, &tmp)) {
            response->pex = listToPex(tmp, &response->pex_count);
        }
    }

    tr_runInEventThread(session, on_announce_done_eventthread, data);
}
```

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, ou seja, a execução primária do programa, de que recebeu a resposta do tracker.

./libtransmission/announcer.c:1012

```
static void on_announce_done(const tr_announce_response * response, void * vdata) {
    struct announce_data * data = vdata;
    (...)
    tr_tier * tier = getTier(announcer, response->info_hash, data->tierId);
    const time_t now = tr_time();

    (...)

    if (tier != NULL) {
        tr_tracker * tracker;
        tier->lastAnnounceTime = now;
        tier->lastAnnounceTimedOut = response->did_timeout;
        tier->lastAnnounceSucceeded = tier->isAnnouncing = false;
        tier->manualAnnounceAllowedAt = now + tier->announceMinIntervalSec;

        (...)
        int i = scrape_fields = seeders = leechers = 0; const char * str;

        (...)

        if (response->seeders >= 0) {
            tracker->seederCount = seeders = response->seeders;
            (...)
        }
        if (response->leechers >= 0) {
            tracker->leecherCount = leechers = response->leechers;
            (...)
        }

        (...)

        if ((i = response->min_interval)) tier->announceMinIntervalSec = i;
        if ((i = response->interval)) tier->announceIntervalSec = i;

        if (response->pex_count > 0)
            publishPeersPex(tier, seeders, leechers, response->pex, response->pex_count);

        (...)
        tier->lastAnnounceSucceeded = true;
        (...)
    }
    (...)
}
```

./libtransmission/announcer.c:523

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

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.

./libtransmission/torrent.c:552

```
static void onTrackerResponse(tr_torrent * tor, const tr_tracker_event * event,
    void * unused UNUSED) {
    switch (event->messageType) {
        case TR_TRACKER_PEERS: {
            size_t i;
            (...)
            for (i = 0; i < event->pexCount; ++i)
                tr_peerMgrAddPex(tor, TR_PEER_FROM_TRACKER, &event->pex[i], seedProbability);
            break;
        }
        (...)
    }
}
```

./libtransmission/peer-mgr.c:2113

```
void tr_peerMgrAddPex(tr_torrent * tor, uint8_t from, const tr_pex * pex,
    int8_t seedProbability) {
    if (tr_isPex(pex)) { /* safeguard against corrupt data */
        tr_swarm * s = tor->swarm;
        managerLock(s->manager);

        if (!tr_sessionIsAddressBlocked(s->manager->session, &pex->addr))
            if (tr_address_is_valid_for_peers(&pex->addr, pex->port))
                ensureAtomExists(s, &pex->addr, pex->port, pex->flags, seedProbability, from);

        managerUnlock(s->manager);
    }
}
```

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

./libtransmission/peer-mgr.c:1852

```
static void ensureAtomExists(tr_swarm * s, const tr_address * addr, const tr_port port,
    const uint8_t flags, const int8_t seedProbability, const uint8_t from) {
    struct peer_atom * a = getExistingAtom(s, addr);

    (...)

    if (a == NULL) { // Se ainda não está na lista...
        const int jitter = tr_cryptoWeakRandInt(60 * 10);
        a = tr_new0(struct peer_atom, 1);
        a->addr = *addr;
        a->port = port;
        (...)

        // ...adiciona peer na lista de peers, de forma ordenada.
        tr_ptrArrayInsertSorted(&s->pool, a, compareAtomsByAddress);
    }
    (...)
}
```

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

As tabelas hash distribuídas (DHTs) surgiram quando buscas em redes peer-to-peer não eram eficientes, fosse pelos problemas da supercentralização dos sistemas, ou pela falta dela, de modo a se criar uma maneira híbrida de buscar e localizar recursos nessas redes.

Existem duas formas de se encontrar recursos: busca e endereçamento. A primeira se baseia no casamento de palavras-chave com as descrições dos recursos, sendo mais amigável para o usuário, pois não necessita de mecanismos complexos de identificação. Porém, é menos eficiente, já que necessita da conferência dos conteúdos dos recursos para saber se são os mesmos. As redes peer-to-peer descentralizadas utilizavam esta abordagem.

Quando uma rede tem estrutura descentralizada, ocorre o fenômeno chamado de inundação de mensagens, no qual estas são repassadas entre peers intermediários até o peer de destino. Esse repasse excessivo acarreta superconsumo de recursos (*overhead*) nos peers intermediários, além de gerar mensagens falso-negativas.

A segunda forma de se encontrar recursos utiliza endereços que possam identificá-los de forma única. Dessa maneira, um recurso é exclusivamente identificável, sendo possível encontrá-lo eficientemente. Contudo, é necessário algum procedimento para se obter seu nome único, além de uma estrutura organizada por endereços. Historicamente, as redes peer-to-peer de estrutura centralizada se baseiam neste método.

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, às vezes forçando o seu desligamento.

Essas formas de organizar esses recursos através da rede utilizam as conexões entre os diversos peers, formando redes superpostas (*overlay networks*), que são redes virtuais que figuram sobre as tradicionais redes IP. Assim, as redes peer-to-peer necessitam de funcionalidades eficientes para que um peer possa entrar e sair do *overlay*, assim como armazenar recursos e encontrá-los (usando endereçamento). Tal eficiência é conseguida através de DHTs, onde todas essas funcionalidades dependem da troca eficiente de mensagens entre peers.

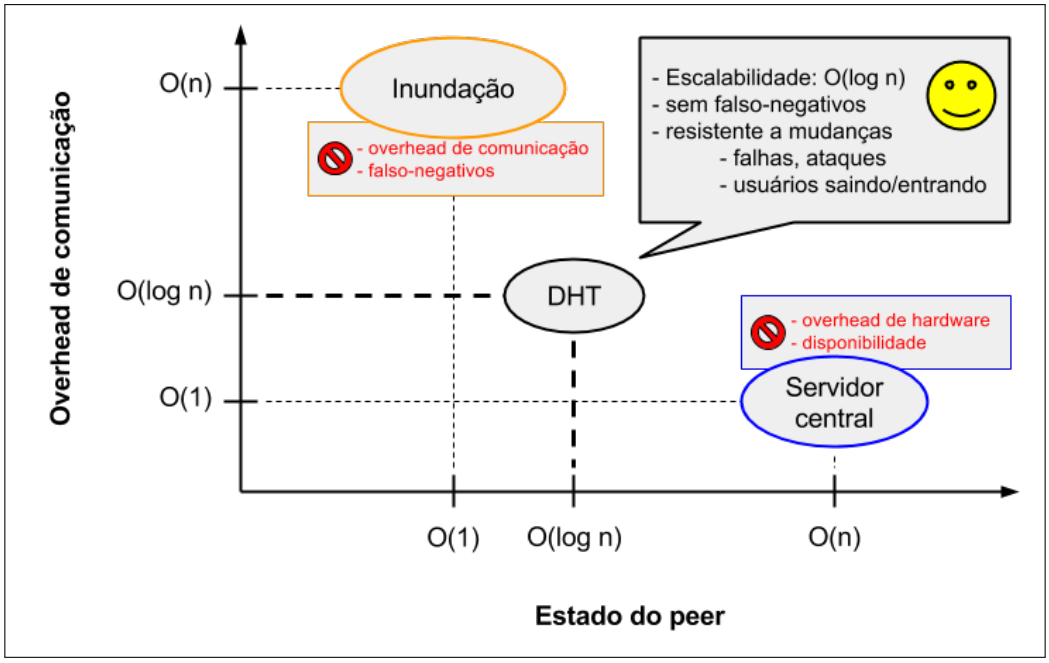

Figura 3.3: *comparação de overheads de redes centralizadas e descentralizadas, e a brecha onde a DHT se encaixa*

A DHT foi adotada pelos desenvolvedores de programas cliente BitTorrent, mas sempre como uma funcionalidade particular, sendo utilizada por alguns anos. Tendo comprovado seu desempenho, foi finalmente adicionada à especificação do BitTorrent no ano de 2008 [65].

Kademlia

O Kademlia é uma DHT, criada em 2002 [66], 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 [78]. 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 no formato 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 esteja em cada uma das subárvores, se estas contiverem algum nó.

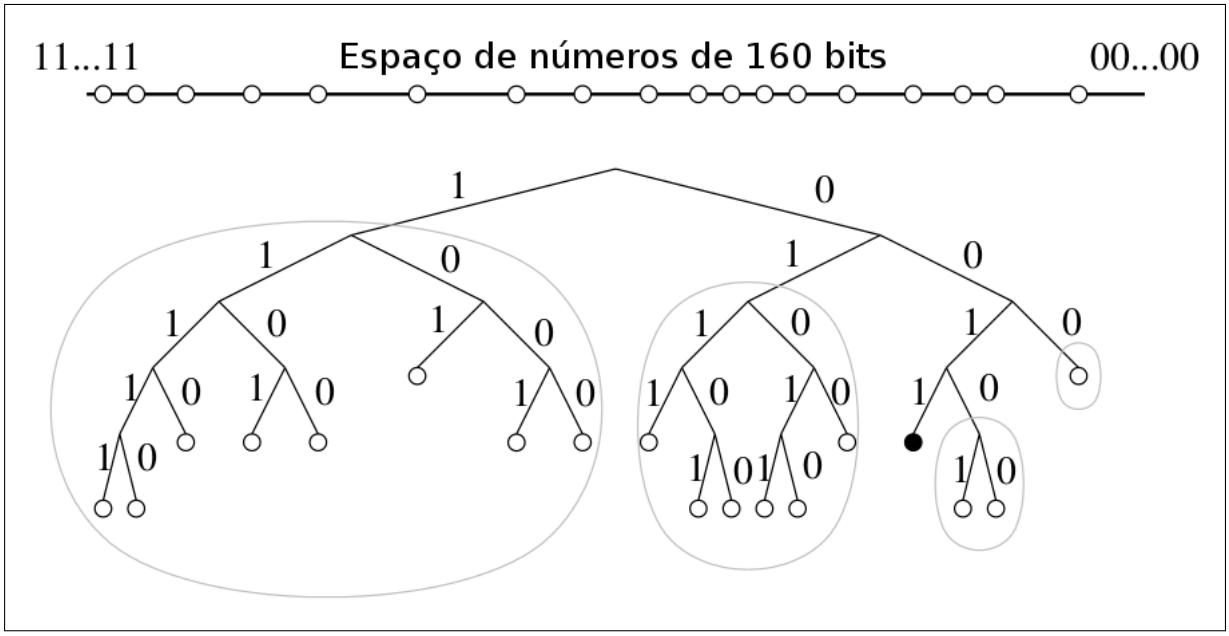

Figura 3.4: *Árvore binária do Kademlia. O nó preto é a posição do ID 0011...; os ovais cinzas são as subárvores onde o nó preto deve possuir nós conhecidos. Fonte: [66]*

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 associada ao objeto, ou seja, o ID, e explorar a rede em passos. A cada passo, encontrará nós mais próximos da chave, até chegar ao valor buscado ou até não existirem nós mais próximos que o atual. Dessa forma, para uma rede com n nós, o algoritmo visita apenas $\mathrm{O}(\log n)$ nós.

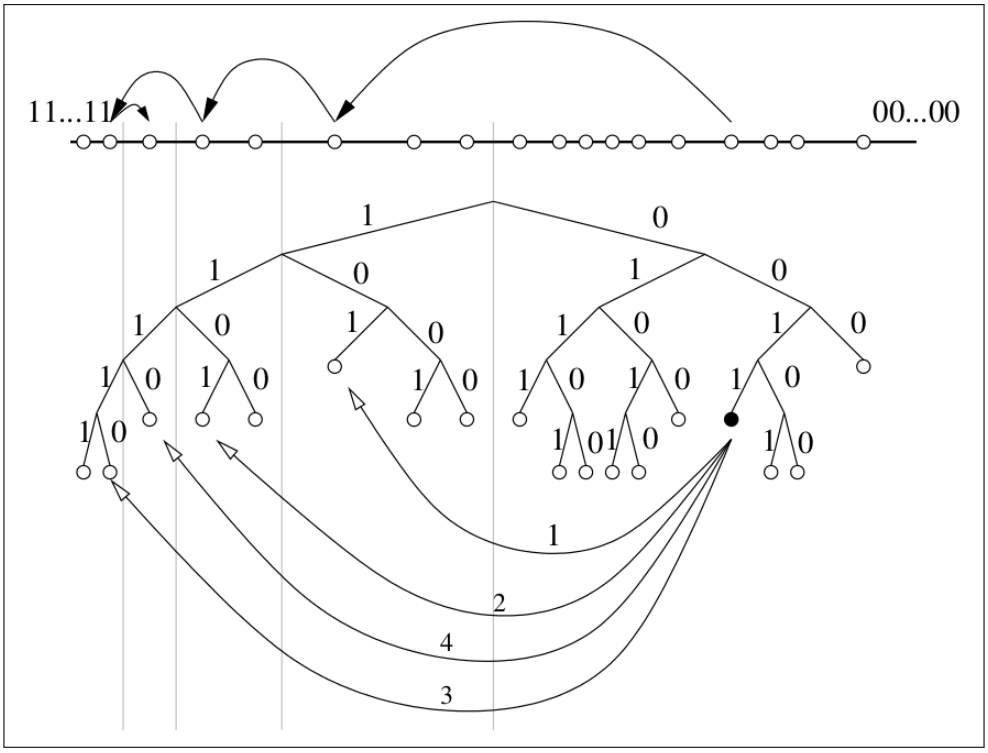

Figura 3.5: *Exemplo de uma busca na árvore de nós do Kademlia usando-se um ID. O nó preto, de prefixo 0011, encontra o nó de prefixo 1110 através de sucessivas buscas (setas numeradas inferiores). As setas superiores mostram a convergência da busca durante a execução. Fonte: [66]*

Para o conceito de proximidade, as distâncias são calculadas usando-se a função de distância baseada em XOR (operador ou-exclusivo lógico) bit a bit

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

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 convirjam 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 [73].

No Transmission, o Kademlia é usado a partir de uma biblioteca externa (não mantida pelos seus desenvolvedores), criada por Juliusz Chroboczek [21], sendo adicionada e adaptada para o uso no código do programa. Sua implementação não é na forma de árvore, mas sim de listas ligadas.

./third-party/dht/dht.c:129

```
struct node {
    // elemento de lista ligada de nós DHT
    (...)
    unsigned char id[20]; // ID do nó
    time_t time;          // horário da última mensagem recebida
    time_t reply_time;    // horário da última resposta recebida corretamente
    time_t pinged_time;   // horário da última requisição
    int pinged;           // quantidade de requisições feitas desde a última resposta
    struct node *next;    // ponteiro para o próximo elemento da lista ligada
};

struct bucket {
    // elemento de lista ligada de baldes
    (...)
    unsigned char first[20];        // ID do primeiro nó do balde
    int count;                      // quantidade de nós no balde
    int time;                       // horário da última resposta neste balde
    struct sockaddr_storage cached; // endereços de possíveis candidatos
    int cachedlen;                  // tamanho da lista de candidatos
    struct node *nodes;             // ponteiro para lista ligada de nós
    struct bucket *next;            // ponteiro para o próximo elemento da lista ligada
};
```

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-*bucket*s 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 que para distâncias maiores, poderão ser de tamanho k. Este valor, que é o de replicação do sistema, é escolhido de tal forma que haja grande probabilidade desses k nós 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 que ele se mantenha conectado por mais uma hora [87]. 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-*bucket*s se os antigos fossem excluídos.

Protocolo

O protocolo de mensagens DHT utiliza o formato KRPC, 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 três tipos de mensagem: consulta (*query*), resposta (*response*) e erro (*error*). Para o protocolo DHT, são quatro 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 duas chaves comuns a todos os quatro 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 dois 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) [116].

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`); e
- `a`: dicionário contendo parâmetros adicionais, dependendo do comando passado na chave `q`.

response:
- `y`: caractere `r`; e
- `r`: dicionário contendo valores da resposta, dependendo do comando passado na chave `q`.

error:

- `y`: caractere `e`;
- `e`: lista contendo dois 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; e
  - 204 (Method Unknown): comando não conhecido.
- exemplo de erro:
  `d1:eli201e23:A Generic Error Ocurrede1:t2:aa1:y1:ee` (bencode)
  `{"t":"aa", "y":"e", "e":[201, "A Generic Error Ocurred"]}` (string)

As informações retornadas podem ser sobre peers ou nós DHT: enquanto a primeiro é a "informação compacta de endereço IP/porta" (como na página 20) — string de 6 bytes (4 bytes iniciais para o endereço IP e bytes finais para a porta de comunicação usada) —, a 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 quatro comandos de *query* do protocolo 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`, ou seja, 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)

./third-party/dht/dht.c:2291

```c
/* We could use a proper bencoding printer and parser, but the format of DHT messages
is fairly stylised, so this seemed simpler. */
#define CHECK(offset, delta, size)                      \
    if(delta < 0 || offset + delta > size) goto fail

#define INC(offset, delta, size)                        \
    CHECK(offset, delta, size);                         \
    offset += delta

#define COPY(buf, offset, src, delta, size)             \
    CHECK(offset, delta, size);                         \
    memcpy(buf + offset, src, delta);                   \
    offset += delta;

#define ADD_V(buf, offset, size)                        \
    if(have_v) {                                        \
        COPY(buf, offset, my_v, sizeof(my_v), size);    \
    }
```

./third-party/dht/dht.c:2341

```c
int send_ping(const struct sockaddr *sa, int salen, const unsigned char *tid, int tid_len) {
    int i = 0, rc; // variáveis temporárias
    char buf[512]; // buffer

    rc = snprintf(buf + i, 512 - i, "d1:ad2:id20:"); INC(i, rc, 512);

    // copia o ID do peer (que está em "myid") no buffer
    COPY(buf, i, myid, 20, 512);

    rc = snprintf(buf + i, 512 - i, "e1:q4:ping1:t%d:", tid_len); INC(i, rc, 512);

    // copia o ID de 4 bytes para a transação do Transmission no buffer
    COPY(buf, i, tid, tid_len, 512); ADD_V(buf, i, 512);

    rc = snprintf(buf + i, 512 - i, "1:y1:qe"); INC(i, rc, 512);
    return dht_send(buf, i, 0, sa, salen); // envia a requisição de ping
    (...)
}

int send_pong(const struct sockaddr *sa, int salen, const unsigned char *tid, int tid_len) {
    int i = 0, rc; // variáveis temporárias
    char buf[512]; // buffer

    rc = snprintf(buf + i, 512 - i, "d1:rd2:id20:"); INC(i, rc, 512);

    // copia o ID do peer (que está em "myid") no buffer
    COPY(buf, i, myid, 20, 512);

    rc = snprintf(buf + i, 512 - i, "e1:t%d:", tid_len); INC(i, rc, 512);

    // copia o ID de 4 bytes para a transação do Transmission no buffer
    COPY(buf, i, tid, tid_len, 512); ADD_V(buf, i, 512);

    rc = snprintf(buf + i, 512 - i, "1:y1:re"); INC(i, rc, 512);
    return dht_send(buf, i, 0, sa, salen); // envia a resposta a um ping
    (...)
}
```

find_node

Este comando, que equivale à mensagem de `FIND_NODE` no artigo do Kademlia [66], é usado para encontrar as informações do nó, dado seu ID. Necessita enviar dois 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 mais próximos (que fizeram contato recentemente), e que estão englobados em sua tabela de roteamento, de um ou mais k-*bucket*s. O funcionamento do algoritmo da busca é explicado no trabalho [67]:

> "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-se 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 **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 a busca em largura usa uma fila FIFO (*First In First Out*) e também uma condição de parada, qual seja, a fila ficar vazia, no algoritmo do Kademlia temos uma fila de prioridade cuja métrica "de prioridade" é a distância para o nó de origem, e também temos uma condição de parada, qual seja, os k nós mais próximos serem 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 o que contém nós mais próximos.

./third-party/dht/dht.c:2536

```c
int send_closest_nodes(const struct sockaddr *sa, int salen, const unsigned char *tid,
    int tid_len, const unsigned char *id, int want, int af, struct storage *st,
    const unsigned char *token, int token_len) {
    unsigned char nodes[8 * 26]; unsigned char nodes6[8 * 38];  // variáveis...
    int numnodes = 0, numnodes6 = 0; struct bucket *b;          // ...temporárias

    (...)
    b = find_bucket(id, AF_INET); // Busca o bucket provável.
    if (b) {
        // Procura por nós no bucket encontrado e nos vizinhos anterior e/ou posterior,...
        // ...se possuir.
        numnodes = buffer_closest_nodes(nodes, numnodes, id, b);
        if (b->next) numnodes = buffer_closest_nodes(nodes, numnodes, id, b->next);
        b = previous_bucket(b);
        if (b) numnodes = buffer_closest_nodes(nodes, numnodes, id, b);
    }
    (...)

    // Envia os nós encontrados.
    return send_nodes_peers(sa, salen, tid, tid_len, nodes, numnodes * 26, nodes6,
        numnodes6 * 38, af, st, token, token_len);
}
```

A busca do *bucket* itera sobre a lista ligada de *bucket*s.

./third-party/dht/dht.c:464

```c
static int id_cmp(const unsigned char *restrict id1, const unsigned char *restrict id2) {
    /* Memcmp is guaranteed to perform an unsigned comparison. */
    return memcmp(id1, id2, 20);
}

static struct bucket * find_bucket(unsigned const char *id, int af) {
    struct bucket *b = af == AF_INET ? buckets : buckets6; // seletor de buckets IPv4/IPv6

    if (b == NULL) return NULL;
    while (1) {
        if (b->next == NULL) return b;
        if (id_cmp(id, b->next->first) < 0) // id tem valor binário menor que b->next->first,
            return b;                       // logo deve ficar neste bucket
        b = b->next;
    }
}
```

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

./third-party/dht/dht.c:2523

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

./third-party/dht/dht.c:2476

```c
static int insert_closest_node(unsigned char *nodes, int numnodes, const unsigned char *id,
    struct node *n) {
    int i, size; // variáveis temporárias

    (...)

    // Itera sobre os k (neste caso, 8) nós mais próximos atuais (nodes) e verifica
    // se algum é o nó procurado ou se é mais próximo que algum nó conhecido (n).
    for (i = 0; i < numnodes; i++) {
        if (id_cmp(n->id, nodes + size * i) == 0) return numnodes;
        if (xorcmp(n->id, nodes + size * i, id) < 0) break;
    }

    // Se tiver iterado por todos os nós próximos atuais, eles são os mais próximos; ...
    if (i == 8) return numnodes;

    if (numnodes < 8) numnodes++; // ... caso contrário, o nó atual (n) entra para a lista.

    if (i < numnodes - 1) // Abre espaço para o nó atual (n) na lista.
        memmove(nodes + size * (i + 1), nodes + size * i, size * (numnodes - i - 1));

    (...)
    // Adiciona o nó atual (n) para a lista, convertendo os dados de endereço IP e porta.
    struct sockaddr_in *sin = (struct sockaddr_in*) &n->ss;
    memcpy(nodes + size * i, n->id, 20);
    memcpy(nodes + size * i + 20, &sin->sin_addr, 4);
    memcpy(nodes + size * i + 24, &sin->sin_port, 2);
    (...)

    return numnodes;
}
```

Ao fim da busca, o Transmission envia a lista de nós que encontrou como resposta ao comando de `find_node` recebida, e também utiliza essa função para enviar os nós encontrados pelo comando `get_peers` (pág. 40).

./third-party/dht/dht.c:2409

```c
int send_nodes_peers(const struct sockaddr *sa, int salen, const unsigned char *tid,
    int tid_len, const unsigned char *nodes, int nodes_len, const unsigned char *nodes6,
    int nodes6_len, int af, struct storage *st, const unsigned char *token, int token_len) {
    char buf[2048]; int i = 0, rc, j0, j, k, len; // variáveis temporárias

    rc = snprintf(buf + i, 2048 - i, "d1:rd2:id20:");
    INC(i, rc, 2048);
    COPY(buf, i, myid, 20, 2048);
    if (nodes_len > 0) {
        // Monta a string bencode dos nós a serem enviados.
        rc = snprintf(buf + i, 2048 - i, "5:nodes%d:", nodes_len);
        INC(i, rc, 2048);
        COPY(buf, i, nodes, nodes_len, 2048);
    }

    (...)

    // find_node: token_len = 0
    // get_peers: token_len > 0
    if (token_len > 0) {
        // Adiciona o token na resposta da chamada.
        rc = snprintf(buf + i, 2048 - i, "5:token%d:", token_len);
        INC(i, rc, 2048);
        COPY(buf, i, token, token_len, 2048);
    }

    // find_node: st = NULL
    // get_peers: st != NULL
    if (st && st->numpeers > 0) {
        /* We treat the storage as a circular list, and serve a randomly
         chosen slice.  In order to make sure we fit within 1024 octets,
         we limit ourselves to 50 peers. */

        len = af == AF_INET ? 4 : 16;
        j0 = random() % st->numpeers;
        j = j0;
        k = 0;

        rc = snprintf(buf + i, 2048 - i, "6:valuesl");
        INC(i, rc, 2048);
        do {
            if (st->peers[j].len == len) {
                unsigned short swapped;
                swapped = htons(st->peers[j].port);
                rc = snprintf(buf + i, 2048 - i, "%d:", len + 2);
                INC(i, rc, 2048);
                COPY(buf, i, st->peers[j].ip, len, 2048);
                COPY(buf, i, &swapped, 2, 2048);
                k++;
            }
            j = (j + 1) % st->numpeers;
        } while (j != j0 && k < 50);
        rc = snprintf(buf + i, 2048 - i, "e");
        INC(i, rc, 2048);
    }

    rc = snprintf(buf + i, 2048 - i, "e1:t%d:", tid_len);
    INC(i, rc, 2048);
    COPY(buf, i, tid, tid_len, 2048);
    ADD_V(buf, i, 2048);
    rc = snprintf(buf + i, 2048 - i, "1:y1:re");
    INC(i, rc, 2048);

    return dht_send(buf, i, 0, sa, salen);
    (...)
}
```

Um exemplo de requisição e resposta para este comando é:

- exemplo de requisição:
  `d1:ad2:id20:abcdefghij01234567896:target20:mnopqrstuvwxyz123456e1:q 9: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`, e serve para buscar peers para um dado 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 serão 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 final 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`:

- 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)>"}`

./third-party/dht/dht.c:1866

```c
int dht_periodic(const void *buf, size_t buflen, const struct sockaddr *from, int fromlen,
    time_t *tosleep, dht_callback *callback, void *closure) {
    (...)
    // variáveis temporárias
    int message;
    unsigned char tid[16], id[20], info_hash[20], target[20];
    unsigned char nodes[256], nodes6[1024], token[128];
    int tid_len = 16, token_len = 128;
    int nodes_len = 256, nodes6_len = 1024;
    unsigned short port;
    unsigned char values[2048], values6[2048];
    int values_len = 2048, values6_len = 2048;
    int want;

    (...)
    // Processa a mensagem recebida e identifica seu tipo.
    message = parse_message(buf, buflen, tid, &tid_len, id, info_hash, target,
            &port, token, &token_len, nodes, &nodes_len, nodes6, &nodes6_len,
            values, &values_len, values6, &values6_len, &want);
    (...)

    // Realiza os procedimentos conforme o tipo de mensagem recebida.
    switch (message) {
        (...)
        case FIND_NODE:
            // find_node: só envia os nós mais próximos ao 'target'.
            debugf("Find node!\n");
            new_node(id, from, fromlen, 1);
            debugf("Sending closest nodes (%d).\n", want);
            send_closest_nodes(from, fromlen, tid, tid_len, target, want, 0, NULL, NULL, 0);
            break;
        case GET_PEERS:
            debugf("Get_peers!\n");
            new_node(id, from, fromlen, 1);
            if (id_cmp(info_hash, zeroes) == 0) {
                (...)
            }
            else {
                struct storage *st = find_storage(info_hash);
                unsigned char token[TOKEN_SIZE];
                make_token(from, 0, token);
                // Se conhecer nós para o hash de torrent dado, os indica na resposta...
                if (st && st->numpeers > 0) {
                    debugf("Sending found%s peers.\n",
                            from->sa_family == AF_INET6 ? " IPv6" : "");
                    send_closest_nodes(from, fromlen, tid, tid_len, info_hash, want,
                            from->sa_family, st, token, TOKEN_SIZE);
                }
                else {
                    // ..., senão procura nós próximos, como no comando find_node.
                    debugf("Sending nodes for get_peers.\n");
                    send_closest_nodes(from, fromlen, tid, tid_len, info_hash,
                        want, 0, NULL, token, TOKEN_SIZE);
                }
            }
            break;
        (...)
    }
    (...)
}
```

- 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:valuesl6: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:aoeusnthe1: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 quatro argumentos: o ID do nó consultante como valor da chave `id`; o valor hash identificador do torrent na chave `info_hash`; `port`, que contém um número inteiro de porta; e o `token` recebido como resposta de uma mensagem `get_peers` anterior. O nó consultado deve verificar se esse token foi enviado anteriormente para o mesmo endereço IP que o nó consultante, e então o nó consultado armazenará 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 sub-redes que possuem NAT, ou seja, que podem não saber quais são suas portas externas ao NAT, e que suportam o protocolo uTP, aceitando conexões na mesma porta que a 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 à uma chave secreta criada pelo programa cliente, que varia periodicamente.

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>"}`

./third-party/dht/dht.c:1277

```c
static int storage_store(const unsigned char *id, const struct sockaddr *sa,
    unsigned short port) {
    int i, len; struct storage *st; unsigned char *ip;

    (...)
    st = find_storage(id); // Encontra o banco de peers para o ID (hash do torrent) dado.

    if (st == NULL) {    // Se não existe banco, cria um para o torrent e adiciona na...
        (...)            // ... lista de bancos.
        st = calloc(1, sizeof(struct storage));
        (...)
        memcpy(st->id, id, 20);
        st->next = storage;
        storage = st;
        numstorage++;
    }

    // Procura pelo peer passado para a função.
    for (i = 0; i < st->numpeers; i++) {
        if (st->peers[i].port == port &&
            st->peers[i].len == len &&
            memcmp(st->peers[i].ip, ip, len) == 0) break;
    }

    // Se o peer já existir, só atualiza o momento da última notificação.
    if (i < st->numpeers) {
        /* Already there, only need to refresh */
        st->peers[i].time = now.tv_sec;
        return 0;
    }
    else {
        struct peer *p;
        // Se não tiver espaço para o peer novo, expande o banco.
        if (i >= st->maxpeers) {
            /* Need to expand the array. */
            struct peer *new_peers; int n;

            // Não tem mais espaço. (DHT_MAX_PEERS = 2048)
            if (st->maxpeers >= DHT_MAX_PEERS) return 0;

            // Expande o banco criando dois espaços ou dobrando de tamanho, limitando em...
            // ...DHT_MAX_PEERS espaços.
            n = st->maxpeers == 0 ? 2 : 2 * st->maxpeers;
            n = MIN(n, DHT_MAX_PEERS);
            new_peers = realloc(st->peers, n * sizeof(struct peer));
            if (new_peers == NULL) return -1;
            st->peers = new_peers; st->maxpeers = n;
        }
        // Adiciona o novo peer no banco.
        p = &st->peers[st->numpeers++];
        p->time = now.tv_sec; p->len = len;
        p->port = port; memcpy(p->ip, ip, len);
        return 1;
    }
}
```

- exemplo de requisição:
  `d1:ad2:id20:abcdefghij01234567899:info_hash20:mnopqrstuvwxyz123456`
  `4:porti6881e5:token8:aoeusnthe1: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 (o *bootstrap node*, ou nó de *bootstrap*). Ao entrar, o novo nó escolherá um ID aleatório, ainda não utilizado, que durará até que ele saia da rede. Feito isso, o novo nó adicionará o nó de *bootstrap* em um de seus k-*bucket*s e iniciará uma consulta `FIND_NODE` em si mesmo. Isso fará com que k-*bucket*s em outros nós sejam populados com o novo ID, assim como os k-*bucket*s do novo nó serão preenchidos com os nós entre ele e o nó de *bootstrap*. Em seguida, o novo nó atualizará todos os k-*bucket*s mais distantes que o do nó de *bootstrap*, para procurar por uma chave aleatória que estará nesse intervalo.

Vale notar que os *bucket*s 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 uma hora. Assim, deverá escolher um ID aleatório do intervalo correspondente e efetuar uma busca por esse ID.

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

Além dessa divisão normal, existe a possibilidade de que árvores muito desbalanceadas atrapalhem 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 às 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 *bucket*s que não contenham o próprio ID. Assim, quando o novo nó dividir *bucket*s, todos os nós de mesmo prefixo saberão de sua existência.

Do Transmission, vale destacar que, quando ele efetua a saída da rede, salva a tabela de nós num arquivo `dht.bootstrap`, cujos peers salvos serão utilizados na fase de *bootstrap* do próximo início da DHT. Caso esses arquivos não existam ou não tenham sido suficientes, o programa utilizará peers fornecidos pela DHT "oficial", localizado no endereço `dht.transmissionbt.com:6881`.

Otimizações

As atualizações periódicas de tabelas ocorrem para evitar dois problemas no *lookup* por chaves válidas: 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 com 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 de 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. Entretando, 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.

Em segundo lugar, no caso de árvores muito desbalanceadas, os nós dividirão k-*bucket*s de acordo com o necessário para garantir o conhecimento total de uma subárvore de tamando $\geq k$. Se antes de republicar pares de chave-valor, um nó x atualizar todos os k-*bucket*s 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 dois casos:

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

2. se a chave a ser atualizada estiver fora do intervalo da subárvore, mas o nó x for um dos k nós mais próximos da chave, então todos os k-*bucket*s de x, para intervalos mais próximos da chave do que a subárvore, terão menos que k nós.

Então, o nó x conhecerá todos os nós dentro desses k-*bucket*s, o que, juntamente do conhecimento da subávore, 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 da 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, ele não é necessário para o funcionamento de uma transmissão BitTorrent, sendo de uso opcional, com o usuário do programa cliente tendo a opção de escolher utilizar esse método habilitando-o em suas configurações.

Este mecanismo faz parte das extensões do protocolo BitTorrent [75], porém, ainda não é oficial (mesmo possuindo especificações de mensagem, sua definição mais forte é uma convenção) [129].

Existem dois protocolos de extensão: o *Azureus Messaging Protocol* (AZMP) e o *libtorrent Extension Protocol* (LTEP) [74]. Em ambos, 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 um minuto.

No Transmission, se o seu uso for habilitado pelo usuário, o programa executará periodicamente uma função de envio de mensagens PEX, seja quando um peer se conectar a ele ou quando receber um pedido de mensagem PEX.

./libtransmission/peer-msgs.c:2280

```c
// Envio de mensagem PEX
static void sendPex(tr_peerMsgs * msgs) {
    if (msgs->peerSupportsPex && tr_torrentAllowsPex(msgs->torrent)) {
        (...)
        // Seleciona os 50 nós conectados e que são úteis ao torrent sendo baixado.
        const int newCount = tr_peerMgrGetPeers(msgs->torrent, &newPex, TR_AF_INET,
            TR_PEERS_CONNECTED, MAX_PEX_PEER_COUNT);
        (...)

        /* build the diffs */
        diffs.added = tr_new(tr_pex, newCount); diffs.addedCount = 0;
        diffs.dropped = tr_new(tr_pex, msgs->pexCount); diffs.droppedCount = 0;
        diffs.elements = tr_new(tr_pex, newCount + msgs->pexCount); diffs.elementCount = 0;
        tr_set_compare(msgs->pex, msgs->pexCount, newPex, newCount, tr_pexCompare,
                sizeof(tr_pex), pexDroppedCb, pexAddedCb, pexElementCb, &diffs);
        (...)

        // Se tiverem novos peers a serem enviados, o faz.
        if (diffs.addedCount || diffs.droppedCount || !diffs6.addedCount ||
            diffs6.droppedCount) {
            int i; tr_variant val; uint8_t * tmp, *walk; // variáveis temporárias
            struct evbuffer * payload;                   // listas de peers do PEX
            struct evbuffer * out = msgs->outMessages;   // mensagem a ser enviada

            /* update peer */
            (...)
            msgs->pex = diffs.elements; msgs->pexCount = diffs.elementCount;
            (...)

            /* build the pex payload */
            tr_variantInitDict(&val, 3);

            if (diffs.addedCount > 0) {
                /* "added" */
                tmp = walk = tr_new(uint8_t, diffs.addedCount * 6);
                for (i = 0; i < diffs.addedCount; ++i) {
                    memcpy(walk, &diffs.added[i].addr.addr, 4); walk += 4;
                    memcpy(walk, &diffs.added[i].port, 2); walk += 2;
                }
                (...)
                tr_variantDictAddRaw(&val, TR_KEY_added, tmp, walk - tmp);
                (...)
            }
            if (diffs.droppedCount > 0) {
                /* "dropped" */
                tmp = walk = tr_new(uint8_t, diffs.droppedCount * 6);
                for (i = 0; i < diffs.droppedCount; ++i) {
                    memcpy(walk, &diffs.dropped[i].addr.addr, 4); walk += 4;
                    memcpy(walk, &diffs.dropped[i].port, 2); walk += 2;
                }
                (...)
                tr_variantDictAddRaw(&val, TR_KEY_dropped, tmp, walk - tmp);
                (...)
            }
            (...)

            /* write the pex message */
            payload = tr_variantToBuf(&val, TR_VARIANT_FMT_BENC);
            evbuffer_add_uint32(out, 2 * sizeof(uint8_t) + evbuffer_get_length(payload));
            evbuffer_add_uint8(out, BT_LTEP);
            evbuffer_add_uint8(out, msgs->ut_pex_id);
            evbuffer_add_buffer(out, payload);
            pokeBatchPeriod(msgs, HIGH_PRIORITY_INTERVAL_SECS);
            dbgmsg(msgs, "sending a pex message; outMessage size is now %zu",
                evbuffer_get_length(out));
            (...)
        }
        (...)
    }
}
```

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.

./libtransmission/peer-msgs.c:1159

```
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 }
```

3.4 Jogo da troca de arquivos

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 responderam 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 da lógica existente 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 no qual se buscará as melhores condições de velocidade 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 peers dessa lista e os algoritmos do BitTorrent dessas trocas.

Estados dos nós e informações

Existem duas características independentes que resultam nas 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 um 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 doze mensagens e dois tipos de assinaturas. Essas mensagens são enviadas entre peers e servem 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 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 [131], 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 (*) e 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$_{h}ash >$: 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; e

<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 do nome do programa cliente e de sua versão (o Transmission envia o prefixo `-TR2820-...`).

./libtransmission/handshake.c:191

```
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_FASTEXT(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 com o seu `peer_id`, assim que detectar o ID do torrent na seção de `info_hash` da mensagem. 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 da 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 determinado período de tempo (geralmente, dois minutos).

Assim como as mensagens que serão descritas a seguir, 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; e

<dados>: dados a serem enviados ao outro peer, dependente da mensagem.

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

./libtransmission/peer-msgs.c:1093

```
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 a mudança de estado de *choking* (ou estrangulamento) da conexão (a maneira que o peer remetente tratará o receptor). Ou seja, o receptor "estrangulado" (ou *choked*) não terá suas requisições atendidas, ao contrário de quando estiver com sua conexão livre (ou *unchoked*).

./libtransmission/peer-msgs.c:431

```
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>`

Estas duas mensagens também servem para indicar mudança de estado, mas no sentido de mudança de interesse que o peer remetente terá no receptor.

./libtransmission/peer-msgs.c:769

```
// bool b:  true ⟹ interested
//          false ⟹ not interested
static void sendInterest(tr_peerMsgs * msgs, bool b) {
    struct evbuffer * out = msgs->outMessages; // buffer de saída
    (...)
    dbgmsg(msgs, "Sending %s", b ? "Interested" : "Not Interested");
    evbuffer_add_uint32(out, sizeof(uint8_t)); // <tamanho> = 1 byte

    // <ID da mensagem>: BT_INTERESTED (2) ou BT_NOT_INTERESTED (3)
    evbuffer_add_uint8(out, b ? BT_INTERESTED : BT_NOT_INTERESTED);
    (...)
}
```

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.

./libtransmission/peer-msgs.c:399

```
// uint32_t index: índice i da parte adquirida
static void protocolSendHave(tr_peerMsgs * msgs, uint32_t index) {
    struct evbuffer * out = msgs->outMessages; // buffer de saída

    evbuffer_add_uint32(out, sizeof(uint8_t) + sizeof(uint32_t)); // <tamanho> = 1B + 4B
    evbuffer_add_uint8(out, BT_HAVE);                             // <ID da mensagem> = 4
    evbuffer_add_uint32(out, index);                              // <indice> = indice i

    dbgmsg(msgs, "sending Have %u", index);
    (...)
}
```

Uma implementação de algoritmo do jogo da troca pode fazer com que um peer, que acabou de adquirir a parte, não emita aviso para todos os peers vizinhos que já possuem. Isso diminui a quantidade total de mensagens enviadas, contribuindo para a redução do *overhead* 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 (em forma de bits sinalizadores) quais partes de um torrent já possui.

Esta mensagem 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 (*spare bits*) no seu final.

Bitfields de comprimento errado, ou sem bits de sobra no final, são considerados erros, e as respectivas conexões devem ser fechadas.

./libtransmission/peer-msgs.c:2120

```
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 16), 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.

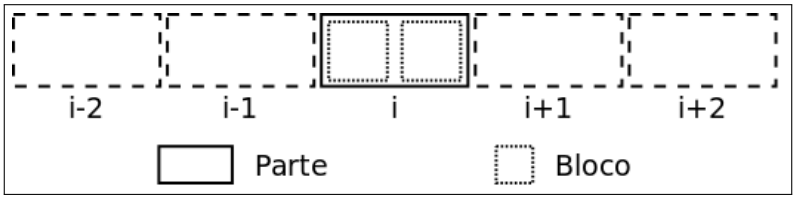

Figura 3.6: *trecho da seção de dados do torrent, com as divisões das partes e dos blocos*

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

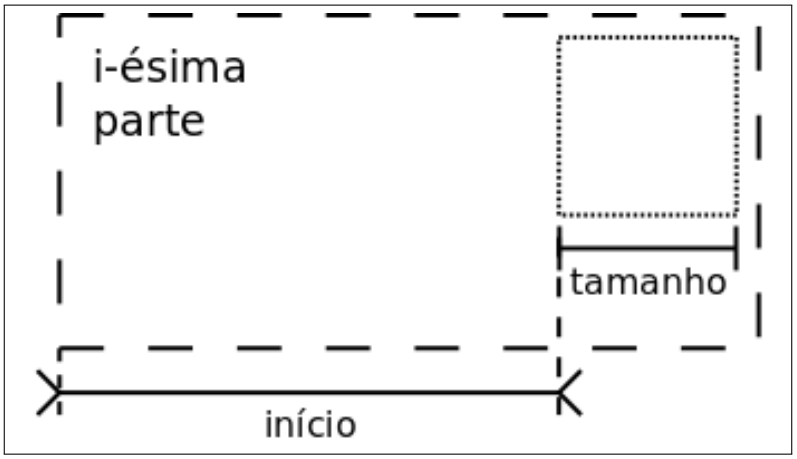

Figura 3.7: *parâmetros da mensagem* request *e seus significados*

índice: índice da parte base i;

início: deslocamento, em bytes, da posição do bloco da parte i pedida; e

tamanho: tamanho, em bytes, do bloco pedido.

./libtransmission/peer-msgs.c:356

```
// const struct peer_request * req: requisição a ser cancelada
static void protocolSendRequest(tr_peerMsgs * msgs, const struct peer_request * req) {
    struct evbuffer * out = msgs->outMessages; // buffer de saída

    // sizeof(uint8_t) + 3 * sizeof(uint32_t) = 1B + 3 * 4B = 13B
    evbuffer_add_uint32(out, sizeof(uint8_t) + 3 * sizeof(uint32_t)); // <tamanho=0013>
    evbuffer_add_uint8(out, BT_REQUEST);    // <ID da mensagem> = 6

    // <dados=<índice><início><tamanho>>
    evbuffer_add_uint32(out, req->index);  // <índice>
    evbuffer_add_uint32(out, req->offset); // <início>
    evbuffer_add_uint32(out, req->length); // <tamanho>

    dbgmsg(msgs, "requesting %u:%u->%u...", req->index, req->offset, req->length);
    (...)
}
```

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; e

bloco: conteúdo dos dados do bloco.

./libtransmission/peer-msgs.c:1093

```
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_peerIoGetWriteBufferSpace(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 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
50     return bytesWritten;
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 (página 59).

./libtransmission/peer-msgs.c:372

```c
// const struct peer_request * req: requisição a ser cancelada
static void protocolSendCancel(tr_peerMsgs * msgs, const struct peer_request * req) {
    struct evbuffer * out = msgs->outMessages; // buffer de saída

    // sizeof(uint8_t) + 3 * sizeof(uint32_t) = 1B + 3 * 4B = 13B
    evbuffer_add_uint32(out, sizeof(uint8_t) + 3 * sizeof(uint32_t)); // <tamanho=0013>
    evbuffer_add_uint8(out, BT_CANCEL);     // <ID da mensagem> = 8

    // <dados=<índice><início><tamanho>>
    evbuffer_add_uint32(out, req->index);  // <índice>
    evbuffer_add_uint32(out, req->offset); // <início>
    evbuffer_add_uint32(out, req->length); // <tamanho>

    dbgmsg(msgs, "cancelling %u:%u->%u...", req->index, req->offset, req->length);
    (...)
}
```

port

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

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

./libtransmission/peer-msgs.c:387

```c
// uint16_t port: número da porta a ser utilizada
static void protocolSendPort(tr_peerMsgs *msgs, uint16_t port) {
    struct evbuffer * out = msgs->outMessages; // buffer de saída

    dbgmsg(msgs, "sending Port %u", port);
    evbuffer_add_uint32(out, 3);       // <tamanho=0003>
    evbuffer_add_uint8(out, BT_PORT); // <ID da mensagem=9>
    evbuffer_add_uint16(out, port);   // <dados=porta>
}
```

Algoritmos de seleção de partes

Por se tratar de um protocolo de troca de dados em partes, uma escolha ruim sobre 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 [23]. 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 ajudar a difundir o conteúdo deste ao resto do swarm.

As estratégias a seguir relacionadas são medidas que um peer BitTorrent pode adotar, gerando efeitos em si mesmo, mas 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 então algum membro do swarm envia-no uma parte comum aleatoriamente (*Random First Piece*). Dessa forma, ele possuirá uma "moeda de troca", podendo com isso ajudar outros peers, e assim, conseguir melhores condições de ser atendido. É importante que a parte seja comum, pois dessa maneira 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 primeiro (*Rarest First*). 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 bifields, 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 deixará de ser interessante, por estar disponibilizando essas partes em um dado momento, 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, será melhor que cada leecher baixe partes diferentes dos outros, que maximizará o espalhamento dos dados e aliviará 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. Com isso, 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 (*Strict Priority*) 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 (*Endgame Mode*), a tendência é que os últimos blocos de dados demorem, chegando aos poucos. Para agilizar isso, o cliente entra no modo de fim de jogo, onde pede todos os blocos faltantes para todos os peers, enviando uma mensagem `cancel`, assim que o bloco for recebido, para todos aqueles que não tiverem respondido à requisição, de modo a evitar despedício 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.

./libtransmission/peer-mgr.c:172

```
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.

./libtransmission/peer-mgr.c:1699

```
// Função de callback para recepção de mensagens de peers.
static void peerCallbackFunc(tr_peer * peer, const tr_peer_event * e, void * vs) {
    tr_swarm * s = vs;

    swarmLock(s);
    switch (e->eventType) {
        (...)
        case TR_PEER_CLIENT_GOT_HAVE:                        // peer avisou que tem uma parte
            if (replicationExists(s)) {
                tr_incrReplicationOfPiece(s, e->pieceIndex); // replicação para uma parte
                (...)
            }
            break;

        case TR_PEER_CLIENT_GOT_HAVE_ALL: // peer avisou que tem todas as partes
            if (replicationExists(s)) {
                tr_incrReplication(s);    // replicação para todas as partes
                (...)
            }
            break;

        (...)

        case TR_PEER_CLIENT_GOT_BITFIELD:                       // peer enviou bitfield
            (...)
            if (replicationExists(s)) {
                tr_incrReplicationFromBitfield(s, e->bitfield); // replicação usando bitfield
                (...)
            }
            break;
        (...)
    }
    swarmUnlock(s);
}
```

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; e
4. *random first piece*: aleatoriedade.

./libtransmission/peer-mgr.c:948

```c
// Função que compara duas partes e calcula a importância de uma sobre a outra.
static int comparePieceByWeight(const void * va, const void * vb) {
    const struct weighted_piece * a = va;
    const struct weighted_piece * b = vb;
    int ia, ib, missing, pending;
    const tr_torrent * tor = weightTorrent;
    const uint16_t * rep = weightReplication;

    // 1o fator: peso (strict policy)
    // Uma parte que possui muitos blocos faltando e alguns pedidos tem prioridade menor...
    // ... do que uma parte que está próxima de terminar.
    missing = tr_cpMissingBlocksInPiece(&tor->completion, a->index);
    pending = a->requestCount;
    ia = missing > pending ?
            missing - pending : (tor->blockCountInPiece + pending);
    missing = tr_cpMissingBlocksInPiece(&tor->completion, b->index);
    pending = b->requestCount;
    ib = missing > pending ?
            missing - pending : (tor->blockCountInPiece + pending);
    if (ia < ib) return -1;
    if (ia > ib) return 1;

    // 2º fator: prioridade
    // O usuário pode ter assinalado para o Transmission que um dos arquivos do torrent...
    // ... deve ser baixado com prioridade maior do que outros.
    ia = tor->info.pieces[a->index].priority;
    ib = tor->info.pieces[b->index].priority;
    if (ia > ib) return -1;
    if (ia < ib) return 1;

    // 3º fator: raridade (rarest first)
    // Se um elemento da lista de partes tiver menos réplicas do que outro, então é...
    // ... mais raro, logo, tem maior prioridade.
    ia = rep[a->index];
    ib = rep[b->index];
    if (ia < ib) return -1;
    if (ia > ib) return 1;

    // 4º fator: aleatoriedade (random first piece)
    // Isso afeta o início do download, quando não se possui nenhuma parte e os peers...
    // ... vizinhos ainda não são bem conhecidos. Nesse caso, a aleatoriedade está no...
    // ... salt (valor aleatório entre 0 e 4096).
    if (a->salt < b->salt) return -1;
    if (a->salt > b->salt) return 1;

    return 0; // partes podem ser consideradas de prioridade igual
}
```

Já o vetor de blocos mantém a lista dos blocos pedidos no momento da requisição e para quem esta foi feita. Essa lista é usada para cancelar requisições que ficaram pendentes por muito tempo ou para evitar requisições duplicadas antes do modo de fim de jogo.

./libtransmission/peer-mgr.c:165

```c
// vetor de pedidos de blocos
struct block_request {
    tr_block_index_t block;
    tr_peer * peer;
    time_t sentAt;
};
```

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

./libtransmission/peer-mgr.c:187

```
// Estrutura de informações sobre o swarm de um torrent.
typedef struct tr_swarm {
    (...)
    tr_ptrArray peers; // vetor de objetos peer
    tr_torrent * tor;  // informações sobre o torrent

    struct block_request * requests; // vetor de pedidos de blocos
    int requestCount;                // tamanho do vetor de pedidos de blocos

    struct weighted_piece * pieces;       // vetor de partes desejadas
    int pieceCount;                       // tamanho do vetor de partes

    // estado do vetor de partes: {não ordenado, por índice, por importância}
    enum piece_sort_state pieceSortState;

    uint16_t * pieceReplication; // vetor sobre quantos peers possuem cada parte
    size_t pieceReplicationSize; // tamanho do vetor de índices de partes comuns

    // indicador do modo de fim de jogo
    // - antes do fim de jogo: endgame = 0
    // - durante o fim de jogo: endgame = média de qtd de requisições pendentes por peer
    // Somente peers "rápidos" serão utilizados no fim de jogo.
    int endgame;
} 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.

./libtransmission/peer-mgr.c:1324

```c
// Função que cria a lista de blocos para serem requisitados a partir da lista de partes.
void tr_peerMgrGetNextRequests(tr_torrent * tor, tr_peer * peer, int numwant,
    tr_block_index_t * setme, int * numgot, bool get_intervals)
{
    int i, got = 0;
    struct weighted_piece * pieces;
    tr_swarm * s = tor->swarm;
    const tr_bitfield * const have = &peer->have;
    (...)

    // Se não existir uma lista de partes, cria uma já ordenada.
    if (s->pieces == NULL) pieceListRebuild(s);

    // Caso já exista a lista de partes, verifica o estado da ordenação por importância.
    if (s->pieceSortState != PIECES_SORTED_BY_WEIGHT)
        pieceListSort(s, PIECES_SORTED_BY_WEIGHT);

    (...)
    // Confirma o estado do torrent, verificando se está no modo de fim de jogo.
    updateEndgame(s); pieces = s->pieces;

    // Para cada parte que falta ser baixada, ...
    for (i = 0; i < s->pieceCount && got < numwant; ++i) {
        struct weighted_piece * p = pieces + i;

        // ... se o peer tiver a parte desejada, ....
        if (tr_bitfieldHas(have, p->index)) {
            tr_block_index_t b, first, last;
            tr_ptrArray peerArr = TR_PTR_ARRAY_INIT;

            // ... calcula o intervalo de bytes da parte.
            tr_torGetPieceBlockRange(tor, p->index, &first, &last);

            // Dentro do intervalo de bytes da parte:
            for (b = first; b <= last && (...)) {
                int peerCount; tr_peer ** peers;

                // Ignore blocos já possuídos.
                if (tr_cpBlockIsComplete(&tor->completion, b)) continue;

                // Encontre peers novos para se requisitar o bloco,...
                // ... juntando às antigas requisições para ele.
                tr_ptrArrayClear(&peerArr);
                getBlockRequestPeers(s, b, &peerArr);
                peers = (tr_peer **) tr_ptrArrayPeek(&peerArr, &peerCount);
                if (peerCount != 0) {
                    // Não faça uma 2ª requisição antes do fim do jogo.
                    if (!s->endgame) continue;

                    // Não requisite o bloco para mais de um peer simultaneamente.
                    if (peerCount > 1) continue;

                    // Não refaça a requisição para o mesmo peer.
                    if (peer == peers[0]) continue;

                    // No modo de fim de jogo, permita requisitar para um 2° peer...
                    // ... somente se este estiver atendendo pedidos rapidamente.
                    if (peer->pendingReqsToPeer + numwant - got < s->endgame) continue;
                }
                (...)

                // Atualiza a lista de blocos pedidos.
                requestListAdd(s, b, peer);
                ++p->requestCount;
            }
            (...)
        }
    }
    (...)
}
```

Tendo criado a lista de blocos, os requisita.

./libtransmission/peer-msgs.c:1874

```c
// Verifica a possibilidade de comunicação com um dado peer e envia requisições de blocos.
static void updateBlockRequests(tr_peerMsgs * msgs) {
    (...)
    blocks = tr_new(tr_block_index_t, numwant); // Cria um vetor de blocos.

    // Monta a lista de quais blocos pedir.
    tr_peerMgrGetNextRequests(msgs->torrent, &msgs->peer, numwant, blocks, &n, false);

    for (i = 0; i < n; ++i) {
        struct peer_request req;
        blockToReq(msgs->torrent, blocks[i], &req); // Cria uma requisição para um bloco.
        protocolSendRequest(msgs, &req);            // Envia uma mensagem de request.
    }
}
```

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 calculada 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.

./libtransmission/peer-mgr.c:878

```c
static bool testForEndgame(const tr_swarm * s) {
    // Modo de fim de jogo acontece quando, em um dado momento:
    // qtd de bytes requisitados ≥ qtd de bytes para completar o torrent
    return (s->requestCount * s->tor->blockSize) >= tr_cpLeftUntilDone(&s->tor->completion);
}
```

./libtransmission/peer-mgr.c:887

```c
static void updateEndgame(tr_swarm * s) {
    (...)

    if (!testForEndgame(s)) { /* not in endgame */
        s->endgame = 0;
    }
    else if (!s->endgame) { /* only recalculate when endgame first begins */
        int i, numDownloading = 0;
        const int n = tr_ptrArraySize(&s->peers);

        // Soma os peers BitTorrent ativos...
        for (i = 0; i < n; ++i) {
            const tr_peer * p = tr_ptrArrayNth(&s->peers, i);
            if (p->pendingReqsToPeer > 0) ++numDownloading;
        }

        // ... com os possíveis seeders web.
        numDownloading += countActiveWebseeds(s);

        // Média instantânea de requisições pendentes por peers fornecedores.
        s->endgame = s->requestCount / MAX(numDownloading, 1);
    }
}
```

Algoritmos de enforcamento

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

O enforcamento (ou *choking*) não faz parte do protocolo de mensagens, mas é necessário para a boa performance do protocolo. Um bom algoritmo de *choking*, que se 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* [122] são aqueles onde duas entidades não podem fazer trocas e ficar mais "felizes" que antes. Em termos de Ciência da Computação, buscar a eficiência de Pareto é usar um algoritmo de otimização local onde as entidades procurem meios de melhorar mutuamente, que convergirão a uma situação ótima global. No contexto do BitTorrent, se dois peers não estão tendo vantagem recíproca por enviar partes, eles podem começar a trocar partes entre si, de modo a conseguir taxas de download melhores do que as anteriores.

Algoritmo de choking

Cada peer fará *unchoke* de quatro peers (geralmente), ou seja, sempre deixará livres de enforcamento cerca de quatro peers. Com isso, o problema passará a ser escolher quais deles fazer, deixando que o TCP controle o congestionamento de banda. Essa escolha é baseada estritamente na taxa de download, calculando-se a média dessa taxa durante vinte segundos. Antigamente, esse cálculo era feito sobre quantidades transferidas a longos prazos, mas notou-se que era uma medida fraca por causa das variações de largura da banda de rede.

Para evitar que recursos sejam desperdiçados pelo rápido *choke* e *unchoke* de peers, um cálculo é realizado a cada dez 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 não é tarefa fácil, pois não há meios de se descobrir conexões melhores. Para tentar contornar esse problema, um peer será escolhido para ser alvo de um "*unchoke* otimista" (*Optimistic Unchoking*), que é um *unchoke* independentemente da taxa de download que ele provê.

A escolha do peer ocorrerá a cada trinta segundos (após o terceiro ciclo de *choke*). Esse tempo é suficiente para que o peer escolhido tome a iniciativa de colaborar enviando partes, sendo recompensado com novas, e essas transmissões alcancem altas taxas de upload e download.

Este algoritmo objetiva, portanto, incentivar a cooperação entre peers.

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 um minuto sem receber nenhuma parte de um outro peer, o peer que não recebeu nada perceberá que foi censurado (*snubbed*), e retaliará deixando de enviar partes para quem o censurou *Anti-snubbing*, a menos que ele seja sorteado em um *unchoke* otimista. Dessa forma, vários *unchokes* otimistas serão executados simultaneamente, o que implicará na recuperação das taxas de download.

Upload Only

Enquanto um peer ainda não recebeu todas as partes do torrent, ele priorizará para quem enviará as que ele possui, de acordo com as taxas de download que ele atinge com os outros peers. Isso acelerará a disseminação do torrent. Porém, quando tiver recebido todas as partes, o peer manterá a rápida disseminação do torrent, priorizando os peers que alcançam altas taxas de download com ele, maximizando sua taxa de upload.

Implementação do Transmission

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

./libtransmission/peer-mgr.c:3167

```
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, verificará se será um turno de *unchoke* otimista, que durará quatro execuções. Além disso, o vetor de peers será analisado para se obter informações, guardadas em um vetor temporário, que ajudarão na decisão de quais desses peers passarão a ser *choked*.

./libtransmission/peer-mgr.c:3057

```c
static void rechokeUploads(tr_swarm * s, const uint64_t now) {
    int i, size, unchokedInterested;
    const int peerCount = tr_ptrArraySize(&s->peers);
    tr_peer ** peers = (tr_peer**) tr_ptrArrayBase(&s->peers);
    struct ChokeData * choke = tr_new0(struct ChokeData, peerCount);
    const tr_session * session = s->manager->session;
    const int chokeAll = !tr_torrentIsPieceTransferAllowed(s->tor, TR_CLIENT_TO_PEER);
    const bool isMaxedOut = isBandwidthMaxedOut(&s->tor->bandwidth, now, TR_UP);

    // Confere se existe um peer que está unchoked de forma otimista
    // (unchoke otimista dura por 4 execuções da função)
    if (s->optimisticUnchokeTimeScaler > 0) s->optimisticUnchokeTimeScaler--;
    else s->optimistic = NULL;

    // Ordena os peers por preferência e avaliação.
    for (i = 0, size = 0; i < peerCount; ++i) {
        tr_peer * peer = peers[i];
        tr_peerMsgs * msgs = PEER_MSGS(peer);
        (...)

        if (tr_peerIsSeed(peer)) {
            // Choke se peer é seeder total ou parcial (não tem o que baixar,...
            // ... só envia dados).
            tr_peerMsgsSetChoke(PEER_MSGS(peer), true);
        }
        else if (chokeAll) { // Choke todos se não está fazendo upload.
            tr_peerMsgsSetChoke(PEER_MSGS(peer), true);
        }
        else if (msgs != s->optimistic) { // Se o peer ainda não estiver no modo otimista.
            struct ChokeData * n = &choke[size++];
            n->msgs = msgs;
            n->isInterested = tr_peerMsgsIsPeerInterested(msgs);
            n->wasChoked = tr_peerMsgsIsPeerChoked(msgs);
            n->rate = getRate(s->tor, atom, now);
            n->salt = tr_cryptoWeakRandInt(INT_MAX); // valor aleatório entre 0 e INT_MAX
            n->isChoked = true;
        }
    }

    qsort(choke, size, sizeof(struct ChokeData), compareChoke);
    (...)
```

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*; e
3. aleatoriedade.

./libtransmission/peer-mgr.c:2993

```
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 restante permanecerá *choked*. Além disso, no caso de um rodada de *unchoke* otimista, sorteará um dos peers para ter a sua vez.

./libtransmission/peer-mgr.c:3057

```
    (...)
    /**
     * (...)
     * Peers which have a better upload rate (as compared to the downloaders) but
     * aren't interested get unchoked. If they become interested, the downloader with
     * the worst upload rate gets choked. If a client has a complete file, it uses its
     * upload rate rather than its download rate to decide which peers to unchoke.
     * (...)
     */

    // O vetor choke está ordenado do peer mais interessante para o menos interessante.
    // Sendo N = session->uploadSlotsPerTorrent, que é a qtd de peers que receberá
    // uploads simultaneamente, N peers serão unchoked. Se já tiver chegado ao limite
    // máximo, ele mantém o estado anterior.
    unchokedInterested = 0;
    for (i = 0; i < size && unchokedInterested < session->uploadSlotsPerTorrent; ++i) {
        choke[i].isChoked = isMaxedOut ? choke[i].wasChoked : false;
        if (choke[i].isInterested) ++unchokedInterested;
    }

    // Se ainda existirem peers na lista, e nenhum no modo otimista, monta-se o sorteio para
    // escolher um.
    if (!s->optimistic && !isMaxedOut && (i < size)) {
        int n; struct ChokeData * c;
        tr_ptrArray randPool = TR_PTR_ARRAY_INIT;

        for (; i < size; ++i) {
            if (choke[i].isInterested) {
                const tr_peerMsgs * msgs = choke[i].msgs;
                int x = 1, y;
                if (isNew(msgs)) x *= 3; // peers novos têm mais chance no sorteio
                for (y = 0; y < x; ++y)
                    tr_ptrArrayAppend(&randPool, &choke[i]);
            }
        }

        // Sorteia um peer aleatoriamente para o unchoke otimista.
        if ((n = tr_ptrArraySize(&randPool))) {
            c = tr_ptrArrayNth(&randPool, tr_cryptoWeakRandInt(n));
            c->isChoked = false;
            s->optimistic = c->msgs;
            s->optimisticUnchokeTimeScaler = OPTIMISTIC_UNCHOKE_MULTIPLIER;
        }
        (...)
    }

    // Envia mensagens choke.
    for (i = 0; i < size; ++i)
        tr_peerMsgsSetChoke(choke[i].msgs, choke[i].isChoked);
    (...)
}
```

Depois, trata os downloads: primeiro, calculará para quantos peers mandará mensagens `interested`, usando como base as estatísticas de pedidos de blocos e quais deles foram cancelados, dentro de limites de bom funcionamento.

./libtransmission/peer-mgr.c:2833

```c
/* determines who we send "interested" messages to */
static void rechokeDownloads(tr_swarm * s) {
    int i, maxPeers, rechoke_count = 0;
    struct tr_rechoke_info * rechoke = NULL;
    const int MIN_INTERESTING_PEERS = 5;
    const int peerCount = tr_ptrArraySize(&s->peers);
    const time_t now = tr_time();

    if (tr_torrentIsSeed(s->tor)) return; // Não é necessário se já somos um seeder.
    if (!tr_torrentIsPieceTransferAllowed(s->tor, TR_PEER_TO_CLIENT)) return;

    /* decide HOW MANY peers to be interested in */
    {
        int blocks = 0, cancels = 0; time_t timeSinceCancel;

        // Conta quantos blocos pedidos e cancelados cada peer tem. Isso é devido ao
        // fato de que o Transmission gastou toda a banda de rede disponível, e agora
        // precisa saber quantos peers utlizará. Peers não responsivos não serão
        // levados em conta.
        for (i = 0; i < peerCount; ++i) {
            const tr_peer * peer = tr_ptrArrayNth(&s->peers, i);
            const int b = tr_historyGet(&peer->blocksSentToClient, now, CANCEL_HISTORY_SEC);
            const int c = tr_historyGet(&peer->cancelsSentToPeer, now, CANCEL_HISTORY_SEC);

            if (b == 0) continue; // ignora peers não responsivos
            blocks += b; cancels += c;
        }

        if (cancels > 0) {
            /* cancelRate: of the block requests we've recently made, the percentage
             * we cancelled. higher values indicate more congestion. */
            const double cancelRate = cancels / (double) (cancels + blocks);
            const double mult = 1 - MIN(cancelRate, 0.5);
            maxPeers = s->interestedCount * mult;
            tordbg(s, "cancel rate is %.3f -- reducing the number of peers we're interested "
                "in by %.0f percent", cancelRate, mult * 100);
            s->lastCancel = now;
        }

        timeSinceCancel = now - s->lastCancel;
        if (timeSinceCancel) {
            const int maxIncrease = 15;
            const time_t maxHistory = 2 * CANCEL_HISTORY_SEC;
            const double mult = MIN(timeSinceCancel, maxHistory) / (double) maxHistory;
            const int inc = maxIncrease * mult;
            maxPeers = s->maxPeers + inc;
            tordbg(s, "time since last cancel is %li -- increasing the number of peers"
                "we're interested in by %d", timeSinceCancel, inc);
        }
    }

    /* don't let the previous section's number tweaking go too far... */
    if (maxPeers < MIN_INTERESTING_PEERS) maxPeers = MIN_INTERESTING_PEERS;
    if (maxPeers > s->tor->maxConnectedPeers) maxPeers = s->tor->maxConnectedPeers;
    s->maxPeers = maxPeers;
    (...)
```

Após saber o limite de peers, o Transmission procurará quais deles contatará, de acordo com as partes que lhe faltam. Para isso, montará um vetor com informações dos peers, ordenando-o pela função `qsort()`, e fornecerá uma função comparadora que avaliará os critérios:

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

./libtransmission/peer-mgr.c:2813

```c
typedef enum {
    RECHOKE_STATE_GOOD, RECHOKE_STATE_UNTESTED, RECHOKE_STATE_BAD
} tr_rechoke_state;
```

./libtransmission/peer-mgr.c:2820

```c
static int compare_rechoke_info(const void * va, const void * vb) {
    const struct tr_rechoke_info * a = va, * b = vb;
    if (a->rechoke_state != b->rechoke_state)
        return a->rechoke_state - b->rechoke_state;
    return a->salt - b->salt;
}
```

./libtransmission/peer-mgr.c:2833

```c
    if (peerCount > 0) {
        bool * piece_is_interesting;
        const tr_torrent * const tor = s->tor;
        const int n = tor->info.pieceCount;

        // Constrói um bitfield com as partes que ainda precisam ser baixadas.
        piece_is_interesting = tr_new(bool, n);
        for (i = 0; i < n; i++)
            piece_is_interesting[i] = !tor->info.pieces[i].dnd &&
                                      !tr_cpPieceIsComplete(&tor->completion, i);

        // Decide por quais peers se interesará, baseado na razão cancelamentos/pedidos
        for (i = 0; i < peerCount; ++i) {
            tr_peer * peer = tr_ptrArrayNth(&s->peers, i);

            if (!isPeerInteresting(s->tor, piece_is_interesting, peer)) {
                tr_peerMsgsSetInterested(PEER_MSGS(peer), false);
            }
            else {
                tr_rechoke_state rechoke_state;
                const int blocks = tr_historyGet(&peer->blocksSentToClient, now,
                    CANCEL_HISTORY_SEC);
                const int cancels = tr_historyGet(&peer->cancelsSentToPeer, now,
                    CANCEL_HISTORY_SEC);

                // Peers bons cancelam 10% ou menos dos pedidos.
                if (!blocks && !cancels) rechoke_state = RECHOKE_STATE_UNTESTED;
                else if (!cancels) rechoke_state = RECHOKE_STATE_GOOD;
                else if (!blocks) rechoke_state = RECHOKE_STATE_BAD;
                else if ((cancels * 10) < blocks) rechoke_state = RECHOKE_STATE_GOOD;
                else rechoke_state = RECHOKE_STATE_BAD;

                if (rechoke == NULL) rechoke = tr_new(struct tr_rechoke_info, peerCount);

                // valor aleatório entre 0 e INT_MAX
                rechoke[rechoke_count].salt = tr_cryptoWeakRandInt(INT_MAX);
                rechoke[rechoke_count].peer = peer;
                rechoke[rechoke_count].rechoke_state = rechoke_state;
                rechoke_count++;
            }
        }
        (...)
    }

    // Ordena os peers de acordo com o interesse.
    qsort(rechoke, rechoke_count, sizeof(struct tr_rechoke_info), compare_rechoke_info);
    s->interestedCount = MIN(maxPeers, rechoke_count);
    for (i = 0; i < rechoke_count; ++i) // Envia mensagens interested.
        tr_peerMsgsSetInterested(PEER_MSGS(rechoke[i].peer), i < s->interestedCount);
    (...)
}
```

Capítulo 4

Ciência da Computação no Transmission

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. Abordaremos de forma superficial pontos que são utilizados explicitamente no programa, comentando o seu funcionamento e utilização.

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 [24].

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 sequências 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 funções próprias onde encapsula o código mostrado.

Listas ligadas

Listas ligadas são estruturas de dados que organizam os objetos de forma linear, assim como os vetores. Porém, enquanto estes possuem índices que determinam a sua posição, em listas, cada elemento possui 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.

./libtransmission/list.h:28

```
typedef struct tr_list {
    void *  data;
    struct tr_list  * next; // ponteiro para o próximo elemento
    struct tr_list  * prev; // ponteiro para o elemento anterior
} 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.

./libtransmission/list.c:19

```
static tr_list * recycled_nodes = NULL;

static tr_list* node_alloc(void) {
    tr_list * ret;                  // ponteiro que aponta para a região alocada

    if (recycled_nodes == NULL) {   // Se não houver elementos reciclados,...
        ret = tr_new(tr_list, 1);   // ... aloque um novo.
    }
    else {                                  // Caso contrário, reutilize, reapontando...
        ret = recycled_nodes;               // ... os ponteiros para os elementos adjacentes.
        recycled_nodes = recycled_nodes->next;
    }

    *ret = TR_LIST_CLEAR;           // limpa campos do elemento
    return ret;                     // devolve o ponteiro para o elemento
}
```

Existem vários tipos de listas ligadas:

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

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

multiplamente ligada: possui ponteiros para 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 operações 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, como por exemplo a lista implementada pelo *framework* de criação de interfaces GTK+.

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 $O(1)$ [24].

verificar se referências do tcc estão na mesma linha.

Uma tabela hash generaliza a noção do *array* 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 em um valor único, chamado de chave, fornecendo um meio de se encontrar essa chave. Esse meio é uma função de hash.

Às vezes, funções de hash fazem com que dois 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 boas funções de hash, descritas a seguir.

A tabela hash usada pelo Transmission aparece na DHT, porém de uma forma mais simples: não existe "a função de hash da DHT", onde há uma função característica para uma modelagem de tabela. Ao invés disso, as chaves já estão calculadas, sendo os IDs dos torrents e dos peers do Kademlia.

Outra utilização de tabelas hash no BitTorrent (apesar de não ser usada no Transmission) é nas **árvores hash** ou **árvores de Merkle** [96]. Essas árvores são usadas para organizar o grande valor hash das partes do torrent, contido no arquivo .torrent, em uma árvore cujas folhas possuem

o valor hash de uma parte e cada nó que não é uma folha possui como valor os valores hash dos seus nós filhos. Dessa forma, o cálculo de um valor hash de um conjunto de partes contínuo pode ser adquirido em O$(\log n)$ [10].

4.2 Funções de hash

Funções de hash são funções matemáticas usadas para gerar conteúdo de comprimento fixo que referencia o conteúdo original.

Isso é útil quando existem grandes quantidades de dados a serem indexados. Por exemplo, numa busca em uma tabela de dados, ou tarefas de comparação de dados, tais 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 usadas para fins criptográficos, são construídas de forma que essa recuperação seja impossível sem que um imenso poder computacional seja utilizado. Por conta disso, é igualmente difícil fingir um valor hash para esconder dados maliciosos.

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, precisa do mesmo valor hash.

Outros usos para funções de hash são nas 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 ocorrida na prática, em um computador comum [138].

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, ainda que de difícil realização atualmente. Essa família é

caracterizada por possuir algoritmos iterativos, baseada no desenho do algoritmo MD4 [84].

O resultado dessa função é um valor hash de 160 bits (ou 20 bytes) na forma hexadecimal. 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 em passos simples de operações de números binários, utilizando alguns valores pré-definidos. Uma variável de encadeamento é usada como mensagem de entrada para a iteração seguinte e os blocos da mensagem expandida se tornam as novas chaves de iteração; e
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, a função de compressão será não-inversível na variável de encadeamento.

Uma das implementações conhecidas para a linguagem C é a biblioteca OpenSSL [77], 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 (*Application Programming Interface*) 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 a 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

```
typedef struct SHAstate_st {
    SHA_LONG h0, h1, h2, h3, h4;
    SHA_LONG Nl, Nh;
    SHA_LONG data[SHA_LBLOCK];
    unsigned int num;
} SHA_CTX;
```

./libtransmission/crypto.c:38

```
void tr_sha1(uint8_t * setme, const void * content1, int content1_len, ...) {
    va_list vl;
    SHA_CTX sha;
    const void * content;

    SHA1_Init(&sha);
    SHA1_Update(&sha, content1, content1_len);

    va_start(vl, content1_len);
    while ((content = va_arg(vl, const void*)))
        SHA1_Update(&sha, content, va_arg(vl, int));
    va_end(vl);

    SHA1_Final(setme, &sha);
}
```

4.3 Criptografia

A criptografia é o estudo e prática de técnicas que visam a segurança de informações de diversas formas, 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, no Egito antigo, a área se manteve em atividade devido a necessidade de se cifrar e quebrar mensagens, se tornando mais necessária com o advento da computação.

Os quatro objetivos principais da criptografia são oferecer [68]:

confidencialidade, para se manter um conteúdo de informação sob sigilo, com acesso somente àqueles autorizados a visualizá-lo. Suas inúmeras formas vão desde a proteção fisica 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 à 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; e

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 [107] 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 [28].

Já as criptografias que utilizam chaves públicas surgiram a partir de outro trabalho de Whitfield Diffie, Martin Hellman [29], 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) [28, 88]: 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 gerador;
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 $S_A = B^{X_A} \bmod p$. Bob faz o mesmo, ou seja, calcula $S_B = A^{X_B} \bmod p$.
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

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

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 dificuldade deste cálculo depende dos tamanhos de p, X_A e X_B. Para um cálculo de aritmética modular, $X \bmod p$ pode ser congruente a valores entre 0 e $p-1$, ou seja, p ter valores grandes já dificulta o trabalho do cálculo inverso. Porém, se forem utilizados X_A e X_B grandes, esse trabalho passa a ser ainda mais difícil. Por isso, 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 [89].

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 um segredo comercial, porém, em 1994, seu código foi publicado na lista de discussão de criptografia CypherPunks [55], se espalhando pela Internet rapidamente. Seu uso se tornou comum, sendo utilizado 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).

./libtransmission/crypto.c:258

```
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 }
```

./libtransmission/crypto.c:237

```
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, 44–46].

Porém, como é dito na especificação da extensão criptográfica no protocolo BitTorrent [50], "o objetivo não é criar um protocolo que consegue, criptograficamente, 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 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 atualmente usada de forma extraoficial pelos programas cliente em geral, inclusive o Transmission.

Comunicação com trackers

Quando a criptografia for habilitada, as comunicações com trackers, ou seja, as requisições de announces, não devem enviar o `info_hash` do torrent. Ao invés disso, devem enviar o `sha_ih`, 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 mantém 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`; e

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

./libtransmission/announcer-http.c:58

```
1  static char* announce_url_new(const tr_session * session, const tr_announce_request * req) {
2      (...)
3      tr_http_escape_sha1(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      // se criptografia estiver habilitada, avisa do seu uso
10     if (session->encryptionMode == TR_ENCRYPTION_REQUIRED)
11         evbuffer_add_printf(buf, "&requirecrypto=1");
12
13     (...)
14     return evbuffer_free_to_str(buf);
15 }
```

Comunicação com peers

A comunicação entre peers é criptografada usando RC4 e troca de chaves Diffie-Hellman-Merkle [134]. Para isso, o protocolo de *handshake* para mensagens entre peers é estendido, de forma a efetuar esses cinco procedimentos criptográficos:

./libtransmission/crypto.c:60-79

```
// Comentários retirados de
// http://wiki.vuze.com/w/Message_Stream_Encryption#Message_Stream_Encryption

// A é o remetente
// chave pública de A: Y_A = (g^X_A) mod p
// B é o receptor
// chave pública de B: Y_B = (g^X_B) mod p

/* P, S, Y_A and Y_B are 768bits long */
#define KEY_LEN 96

/* Prime p is a 768 bit safe prime,
"0xFFFFFFFFFFFFFFFFC90FDAA22168C234C4C6628B80DC1CD129024E088A67CC74
   020BBEA63B139B22514A08798E3404DDEF9519B3CD3A431B302B0A6DF25F1437
   4FE1356D6D51C245E485B576625E7EC6F44C42E9A63A36210000000000090563"*/
#define PRIME_LEN 96
static const uint8_t dh_P[PRIME_LEN] =
{
  0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xC9, 0x0F, 0xDA, 0xA2,
  0x21, 0x68, 0xC2, 0x34, 0xC4, 0xC6, 0x62, 0x8B, 0x80, 0xDC, 0x1C, 0xD1,
  0x29, 0x02, 0x4E, 0x08, 0x8A, 0x67, 0xCC, 0x74, 0x02, 0x0B, 0xBE, 0xA6,
  0x3B, 0x13, 0x9B, 0x22, 0x51, 0x4A, 0x08, 0x79, 0x8E, 0x34, 0x04, 0xDD,
  0xEF, 0x95, 0x19, 0xB3, 0xCD, 0x3A, 0x43, 0x1B, 0x30, 0x2B, 0x0A, 0x6D,
  0xF2, 0x5F, 0x14, 0x37, 0x4F, 0xE1, 0x35, 0x6D, 0x6D, 0x51, 0xC2, 0x45,
  0xE4, 0x85, 0xB5, 0x76, 0x62, 0x5E, 0x7E, 0xC6, 0xF4, 0x4C, 0x42, 0xE9,
  0xA6, 0x3A, 0x36, 0x21, 0x00, 0x00, 0x00, 0x00, 0x00, 0x09, 0x05, 0x63,
};

/* Generator g is "2" */
static const uint8_t dh_G[] = { 2 };

/* X_A and X_B are a variable size random integers.
They are the private key for each side and have to be discarded after the Diffie-Hellman
(DH) handshake is done. Minimum length is 128 bit. Anything beyond 180 bit is not
believed to add any further security and only increases the necessary calculation time.
You should use a length of 160bits whenever possible, lower values may be used when CPU
time is scarce. */
#define DH_PRIVKEY_LEN_MIN 16
#define DH_PRIVKEY_LEN 20

/* DH secret: S = (Y_A^X_B) mod p = (Y_B^X_A) mod P */
```

1) A envia Y_A para B:

O Transmission, nesse caso o peer A, envia sua chave pública (Y_A) com um trecho de dados aleatórios, com comprimento qualquer entre 0 e 512 bytes;

./libtransmission/handshake.c:313

```c
/* 1 A → B: Diffie Hellman Y_A, PadA */
static void sendYa(tr_handshake * handshake) {
    int len; const uint8_t * public_key;
    char outbuf[KEY_LEN + PadA_MAXLEN]; // 96 + 512 = 608 bytes
    char *walk = outbuf;

    /* add our public key (Y_A) */
    public_key = tr_cryptoGetMyPublicKey(handshake->crypto, &len);
    (...)
    memcpy(walk, public_key, len); walk += len;

    /* add some bullshit padding */
    len = tr_cryptoRandInt(PadA_MAXLEN);
    tr_cryptoRandBuf(walk, len); walk += len;

    /* send it */
    setReadState(handshake, AWAITING_YB);
    tr_peerIoWriteBytes(handshake->io, outbuf, walk - outbuf, false);
}
```

2) A recebe Y_B de B:

O outro peer, B, responde com sua chave pública (Y_B). Assim, a chave compartilhada S já pode ser calculada;

./libtransmission/handshake.c:391

```c
/* 2 A ← B: Diffie Hellman Y_B, PadB */
static int readYb(tr_handshake * handshake, struct evbuffer * inbuf) {
    int isEncrypted;
    const uint8_t * secret;
    uint8_t yb[KEY_LEN];
    struct evbuffer * outbuf;
    size_t needlen = HANDSHAKE_NAME_LEN;

    (...)
    isEncrypted = memcmp(evbuffer_pullup(inbuf, HANDSHAKE_NAME_LEN),
                         HANDSHAKE_NAME, HANDSHAKE_NAME_LEN);
    (...)
    tr_peerIoSetEncryption(handshake->io, isEncrypted ? PEER_ENCRYPTION_RC4
                                                      : PEER_ENCRYPTION_NONE);

    if (!isEncrypted) {
        setState(handshake, AWAITING_HANDSHAKE); // Handshake não encriptado acaba aqui.
        return READ_NOW;
    }

    (...)
    /* compute the secret S*/
    evbuffer_remove(inbuf, yb, KEY_LEN);
    secret = tr_cryptoComputeSecret(handshake->crypto, yb);
    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 das partes encripta 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 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;

./libtransmission/handshake.c:391

```
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_sha1(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_sha1(req2, "req2", 4, tr_cryptoGetTorrentHash(handshake->crypto),
22            SHA_DIGEST_LENGTH, NULL);                       // HASH ('req2', SKEY)
23        tr_sha1(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 };
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 };
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:

./libtransmission/handshake.c:587

```
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 }
```

./libtransmission/peer-io.c:1083

```
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 e o traffic shaping

No BitTorrent, os bitfields são usados como uma tabela de controle de quais partes de um torrent o programa cliente já recebeu de outros peers, permitindo conhecer quais são as partes mais raras. Com eles, o peer consegue utilizar a estratégia de *Rarest First* (58), priorizando partes raras antes das mais comuns.

Com o poder crescente da tecnologia de banda larga e a disseminação do uso do BitTorrent em todo o mundo, as ISPs têm tido trabalho em manter a qualidade de suas infraestruturas com a imensa quantidade de dados que trafegam pelos seus cabos. Para evitar que o tráfego em excesso cause perda de desempenho no oferecimento do serviço, as empresas realizam o chamado *traffic shaping* (modelagem de tráfego), que foi especificado em 1998 [16].

Para conseguir controlar o grande volume da sua rede, as ISPs tentam classificar os dados de acordo com seus protocolos, as portas utilizadas e as informações de cabeçalho dos pacotes. Feito isso, os pacotes entram na fila correspondente ao seu tipo, até esta fila ser enviada depois de seguir o contrato de tráfego da fornecedora da conexão. É dessa forma que os pacotes enviados durante o uso de BitTorrent se atrasam ou, até mesmo, se perdem, causando limitação das taxas de transmissão.

Especificamente, o controle do tráfego é admissível porque, durante a análise dos pacotes, é possível perceber o distinto protocolo de *handshake* do BitTorrent, onde ocorre o envio dos bitfields. Para se evitar isso, foi desenvolvida a técnica de *lazy bitfield* (bitfield preguiçoso): o peer envia um bitfield indicando que possui menos partes que a realidade, enviando a diferença em seguida, utilizando mensagens `have`. Assim, o peer finge não ser um potencial seeder, por exemplo, evitando ter sua rede regulada.

A proposta de extensão *Fast* do protocolo BitTorrent [49], que surgiu em 2008, permite o uso de uma mensagem simples para avisar outros peers, que também suportarem o protocolo, que o remetente completou o torrent sem a necessidade de enviar o bitfield. Por conta disso, a probabilidade da conexão ser controlada é menor.

4.5 Protocolos TCP e UDP

A Internet é o meio mais importante de comunicação que existe atualmente. Usamos de forma tão corriqueira que nem nos damos conta de quantas camadas e protocolos existem em uso em um único instante. Para chegar até o que é hoje, passou por uma grande evolução, que começou a partir da precursora da Internet, a ARPAnet, que em outubro de 1969 [61] conectou quatro universidades. Atualmente, seus protocolos são mantidos pela IETF (Internet Engineering Task Force).

Tecnicamente falando, a Internet é organizada em uma pilha de camadas de protocolo, que oferecem e consomem serviços às camadas adjacentes, permitindo que dados sejam roteados entre um computador emissor e outro receptor. Esses protocolos podem estar implementados tanto por *software*, por *hardware*, ou por uma combinação de ambos. A vantagem da modelagem da pilha é ela que provê um meio organizado de se discutir as partes do sistema, e até atualizá-las separadamente. Em contrapartida, uma camada pode necessitar de um valor presente em outra, ou ainda possuir alguma funcionalidade já implementada em outra.

As cinco camadas que representam a pilha de camadas de protocolo da Internet podem ser explicadas fazendo-se uma analogia com um serviço postal [79]:

aplicação: é onde existem as aplicações de rede e seus protocolos; ocorrem as traduções de endereços de Internet para endereços de rede (DNS); e transmissões de documentos de Internet (HTTP), de mensagens de e-mail (SMTP) e de arquivos (FTP). Os pacotes de dados dessa camada são chamados de **mensagens**.

Nessa camada, seria onde uma pessoa escreve uma carta a uma amiga e a deposita na caixa de correio. A pessoa amiga recebe a carta na sua caixa de correio e a lê. Ambas as pessoas não sabem dos processos e rotas que a carta tomou. Bastou uma enviar a carta e a outra recebê-la;

transporte: é onde atuam os protocolos TCP e UDP, que recebem as mensagens através de sockets de rede e as transformam em **segmentos**. Além disso, também criam conexões entre cliente e servidor (multiplexação e demultiplexação dos dados) e as monitora, prevenindo erros.

No caso do serviço postal, o remetente será avisado se enviar a carta para um endereço incorreto (por exemplo, se tiver errado o Estado), ou ainda se uma carta registrada não puder ser entregue. Nesses casos, a carta é devolvida, ficando a cargo da pessoa decidir o que fazer após esse problema;

rede: é responsável por transportar pacotes, conhecidos como **datagramas**, para outro computador. Ele recebe da camada de transporte um segmento e um endereço de destino. Assim, funciona como um serviço de entrega, que sabe quais rotas o datagrama deve tomar para chegar ao destino. Também é onde atua o protocolo IP nas versões IPv4 e IPv6, que todo componente de Internet deve possuir, e que define alguns dados no datagrama, da mesma forma que equipamentos roteadores fazem.

O serviço de entrega de correio usaria aviões para transportar suas cartas entre as cidades, porém seu piloto não saberia quem as enviou, para quem levando ou quais seus conteúdos;

enlace: responsável pela transmissão dos dados que recebe da camada de rede. A cada nó da rota, a camada de rede repassa o datagrama para a camada de enlace, que então o transforma em **quadros** e o entrega para o próximo nó da rede. Esse nó recebe os quadros na sua camada de enlace e os repassa a sua camada de rede. Dos protocolos que atuam nesta camada, se incluem o Ethernet, os vários de conexões wifi, os de ponto a ponto (PPP), etc.

Para o serviço postal, seu equivalente seria a frota de caminhões e entregadores, que distribui os pacotes dentro de uma cidade; e

física: responsável pela transmissão de cada bit dos quadros, de um nó para outro. Os protocolos dessa camada dependem do tipo de meio pelo qual os nós estão ligados. Por exemplo, o protocolo Ethernet possui uma especificação para cabeamentos coaxiais, outra para cabos de par trançado, outra para fibra óptica, etc.

Na analogia das cartas, são as canetas e papéis usados em sua escrita, ou a luz acesa para sua leitura.

Uma característica bastante forte do BitTorrent é o uso perceptível dos protocolos TCP e UDP, onde ele implementa sempre a mesma funcionalidade para cada um desses protocolos. Ambos têm suas características , que podem ser explorados dependendo da aplicação, para uma melhor utilização dos recursos.

UDP

O protocolo UDP é um protocolo de conexão que pertence à camada de transporte e que especifica conexões quase que diretas (se comparado com o TCP) entre as camadas de aplicação e de rede.

Quando mensagens são enviadas por uma aplicação, elas são recebidas através dos sockets. Depois, são anexadas aos endereços IP e número de porta de origem e de destino, e aos comprimentos e *checksum* do cabeçalho e do corpo de dados UDP, para enfim serem repassadas como segmentos para a camada de rede. Por sua vez, a camada de rede encapsula os segmentos em datagramas e faz o melhor possível para entregá-los ao destinatário.

Já quando datagramas são recebidos, o protocolo UDP utiliza a porta de destino contida no cabeçalho para entregar os dados do segmento para o processo correto de aplicação. Como não existe nenhum protocolo de *handshake* entre as camadas de transporte de ambos remetente e destinatário, o protocolo UDP é dito “sem conexão” ou “não orientado a conexão”. Além disso, é considerado não confiável, já que não há garantia de entregas dos pacotes, muito menos a entrega na ordem correta.

Apesar de ser considerado não confiável, existem vantagens em se escolher o UDP ao invés do TCP:

- controle avançado de quais dados são enviados e quando: como o TCP possui controle de congestionamento e confirmação de recebimento de segmentos, pode ser que a aplicação seja comprometida pelo atraso de algum datagrama. No caso de aplicações em tempo real, é possível suportar alguma perda de dados, ou ainda implementar o seu próprio método de verificação de integridade;

- dispensa conexões: enquanto o TCP utiliza protocolos de *handshake* antes da transferência dos dados, o UDP já os envia sem a necessidade de contatos anteriores. Essa agilidade é o principal motivo pelo qual servidores de DNS (*Domain Name Service*) utilizam UDP;
- não mantém estado da conexão: essas informações de estado são necessárias para se conseguir uma conexão de dados confiável, como faz o TCP, que usa buffers de entrada e saída, gerencia congestionamento de dados e possui parâmetros de confirmação. Por conta disso, um servidor dedicado a uma aplicação consegue suportar muito mais conexões do que se fosse usado TCP;
- pouco *overhead* de cabeçalho de pacotes: enquanto o TCP possui *overhead* de cabeçalho de 20 bytes por segmento, o UDP possui 8 bytes.

Originalmente, os trackers utilizavam o protocolo TCP, porém, com o tempo, percebeu-se que com o protocolo UDP eles se tornariam mais eficientes, reduzindo o consumo da largura de rede pela metade [137]. Por esses motivos, os servidores dos trackers utilizam prioritariamente o protocolo UDP.

No Transmission, trackers podem ser adicionados por arquivos .torrent ou manualmente. Quando o programa precisa atualizar as informações sobre cada um desses trackers, procura pelos respectivos endereços de announce. Caso os possua com esquemas URI para TCP e UDP, o Transmission escolhe aquele em UDP.

./libtransmission/announcer.c:555

```c
// Classifica dois trackers quaisquer.
static int filter_trackers_compare_func(const void * va, const void * vb) {
    const struct ann_tracker_info * a = va;
    const struct ann_tracker_info * b = vb;

    if (a->info.tier != b->info.tier) // ordem de ocorrência (ou seja, peers diferentes)
        return a->info.tier - b->info.tier;

    return -strcmp(a->scheme, b->scheme); // protocolo usado: UDP é preferido a HTTP (TCP)
}
```

TCP

O protocolo TCP é um protocolo de conexão que pertence à camada de transporte, que determina um meio de transmissão confiável de dados e orientado a conexão.

Quando as mensagens chegam pelos sockets, o protocolo TCP também as anexa aos endereços IP e número de porta de origem e de destino, e também aos comprimentos e *checksum* do cabeçalho e do corpo de dados TCP, para enfim serem repassadas como segmentos para a camada de rede. Porém, acrescenta também outras informações:

- números de sequência e de confirmação de recebimento (*acknowledgement*), para uso na implementação do serviço de entrega confiável de dados;

- quantidade de bytes que o destinatário deseja receber (*receive window*);
- campos de opção, usados na negociação do tamanho da janela de dados ou do tamanho máximo do segmento (MSS);
- campos de *flags* sinalizadores, onde 6 bits indicam estados da conexão ou do segmento:

  URG: indicador de urgência do segmento enviado;

  SYN: número de sincronização de sequência, que deve ser enviado sempre no primeiro pacote de uma remessa de dados por um nó;

  ACK: indica que o valor contido no campo de confirmação de recebimento é válido. Ele deve ser enviado depois do primeiro pacote com SYN;

  PSH: para indicar ao destinatário que repasse os dados para a camada de aplicação imediatamente ao receber o pacote;

  RST: campo de redefinição de conexão, para indicar o erro de que não possui um socket na porta indicada no cabeçalho; e

  FIN: indica fim do envio dos dados pelo nó.

- ponteiro de dados urgentes, que indica o fim do trecho de dados urgentes.

Na prática, PSH, URG e o ponteiro de dados urgentes não são utilizados [61].

Munido de todos esses dados, o TCP implementa um protocolo de *handshake* de quatro passos que precede o envio dos dados de fato, e então passa a enviar o conjunto de dados de forma ordenada, com checagem de erros de transmissão e de recebimento, e controles de congestionamento de envio e de recebimento, utilizando confirmações e limites de tempo. É um protocolo grande se comparado ao UDP e, por consequência, utiliza maior quantidade de dados e de banda de conexão.

BitTorrent prefere UDP

Em 2008, a substituição do TCP pelo UDP, como protocolo utilizado na troca de partes no BitTorrent, foi desenvolvida pioneiramente pelo programa cliente μTorrent. Essa mudança instantaneamente causou discussões acaloradas pela Internet.

Richard Bennett, arquiteto de redes que escreveu o primeiro padrão Ethernet para cabos de par trançado e ajudou no desenvolvimento de protocolos wifi, publicou um artigo [11] que dizia que a substituição para o protocolo UDP poderia causar um colapso da Internet como um todo, pois os ISPs não teriam como controlar o tráfego de dados BitTorrent por ele, já que necessitavam que os dados fossem transmitidos pelo protocolo TCP. Esse colapso afetaria usuários de outros serviços UDP (na sua maioria, aplicações de tempo real), como jogos online ou sistemas de comunicação VoIP.

Enquanto isso, desenvolvedores de programas cliente responderam dizendo que não existiam motivos para preocupações. Simon Morris, que na época era chefe da gestão de produto da empresa BitTorrent, afirmou [18] que a substituição de protocolo permitiria um melhor controle de congestionamentos das transmissões de dados. Com o TCP, esses congestionamentos só seriam conhecidos depois de suas ocorrências. Por outro lado, com o uso do UDP, seria possível prevê-los medindo-se as taxas de transmissão entre peers.

Outro depoimento foi o de um gerente de comunidade do μTorrent, chamado Firon, que explicou a um site de notícias relacionadas a BitTorrent [58] o mesmo argumento dado por Simon Morris. Além disso, acrescentou que o uso do TCP no BitTorrent significaria o uso de dois protocolos de *handshake* e, portanto, a mudança para UDP eliminaria essa redundância, reduzindo o tráfego na Internet.

Essa grande discussão ainda causa reflexos nos dias atuais, pois ISPs ainda tentam controlar o fluxo de dados BitTorrent no seu fornecimento de Internet. Porém, conforme os dados de uma pesquisa recente sobre a quantidade de tráfego na Internet durante horários de pico [54], 60,47% do tráfego de download e 54,97% do tráfego total norte-americanos são gerados pelos serviços de *streaming* de vídeo Netflix e Youtube e por navegação usual de sites. Enquanto isso, o BitTorrent consome apenas 5,57% e 9,23%, respectivamente. Portanto, não se pode afirmar com confiança que o BitTorrent seja um grande responsável pelo congestionamento de dados nos Estados Unidos [57].

Outra evidência de que o protocolo UDP pode ter importância em aplicações onde o TCP seria preferido, é que o Google está trabalhando atualmente no QUIC (*Quick UDP Internet Connections*), que fará parte da especificação do protocolo HTTP 2.0 [85]. O QUIC, apesar de aumentar o consumo de banda, reduzirá o tempo de resposta de confirmação de recebimento de pacotes, entre outras melhorias.

4.6 Multicast

Na área de redes de computadores, *multicast* é um conceito de entrega de informação simultânea para um grupo de computadores a partir de uma única fonte, utilizando algoritmos específicos para seu roteamento. Geralmente, é implementado sobre os protocolos da camada de rede ou de enlace, e é usado em aplicações que necessitam de distribuição de "um para muitos" ou "muitos para muitos", como por exemplo as de distribuição de dados em massa (atualização de servidores), transmissão de áudio e vídeo contínuo (transmissões de vídeo ao vivo), aplicações de dados compartilhados (teleconferência), fontes de dados (bolsa de valores), atualização de cache de Internet e jogos multi-jogadores interativos [61].

Os endereços de *multicast* são definidos pelo protocolo IP como a sub-rede formada entre os endereços IP `224.0.0.0` até `239.255.255.255`, antigamente chamada de **rede classe D**, onde cada trecho desse intervalo é reservado a um uso específico, controlado pela IANA (*Internet Assigned Numbers Authority*) [86]. Dentre estes, endereços contidos no intervalo de `239.0.0.0` a `239.255.255.255`, chamados de **endereços de multicast de escopo administrativo**, são destinados a uso privado dentro dos limites de organizações, porém, podendo não ser globalmente únicos [69]. Cada endereço deles é de um grupo específico de *multicast*, cujas mensagens encaminhadas a esse endereço serão repassadas a todos os nós de rede que estiverem conectados a ele.

Um tipo de mensagem enviada em *multicast* é a que utiliza o padrão SSDP (*Simple Service Discovery Protocol*), que serve para anúncio e descoberta de equipamentos conectados a uma rede, sem a necessidade de mecanismos de configuração baseados em servidor. Esse protocolo envia mensagens semelhantes às HTTP utilizando o protocolo UDP. Apesar de não ser especificado por nenhum órgao técnico, o SSDP é amplamente utilizado e faz parte da especificação do UPnP (*Universal Plug and Play*) [98].

O BitTorrent utiliza o *multicast* de forma não oficial (que não possui especificação formal) para a **descoberta de peers local**. Inicialmente desenvolvido pelo programa μTorrent e adotado por outros, um peer se conecta a um grupo de *multicast* no endereço `239.192.152.143:6771` e espera por mensagens SSDP [64], onde estão indicadas o endereço IP e porta do peer remetente, adicionando à sua lista de peers.

./libtransmission/tr-lpd.c:425

```c
/**
 * @brief Process incoming unsolicited messages and add the peer to the announced
 * torrent if all checks are passed.
 *
 * @param[in,out] peer Adress information of the peer to add
 * @param[in] msg The announcement message to consider
 * @return Returns 0 if any input parameter or the announce was invalid, 1 if the peer
 * was successfully added, -1 if not; a non-null return value indicates a side-effect to
 * the peer in/out parameter.
 *
 * @note The port information gets added to the peer structure if tr_lpdConsiderAnnounce
 * is able to extract the necessary information from the announce message. That is, if
 * return != 0, the caller may retrieve the value from the passed structure.
 */
static int tr_lpdConsiderAnnounce(tr_pex* peer, const char* const msg) {
    enum {
        maxValueLen = 25,
        maxHashLen = lengthof(lpd_torStaticType->info.hashString)
    };
    (...)
    char value[maxValueLen] = { }, hashString[maxHashLen] = { };
    int res = 0, peerPort = 0;

    if (peer != NULL && msg != NULL) {
        tr_torrent* tor = NULL;

        // retira BT-SEARCH * HTTP/1.1 do cabeçalho da mensagem
        const char* params = lpd_extractHeader(msg, &ver);

        (...)

        // lê a porta enviada na mensagem e a carrega em 'value'
        if (lpd_extractParam(params, "Port", maxValueLen, value) == 0) return 0;

        // processa o número da porta e o guarda em 'peerPort'
        if (sscanf(value, "%d", &peerPort) != 1 || peerPort > (in_port_t) -1) return 0;

        peer->port = htons(peerPort);
        res = -1; /* signal caller side-effect to peer->port via return != 0 */

        // lê o hash do torrent anunciado
        if (lpd_extractParam(params, "Infohash", maxHashLen, hashString) == 0)
            return res;

        // Se o Transmission estiver participando do torrent anunciado, adiciona o...
        // ... endereço do remetente à lista de peers.
        tor = tr_torrentFindFromHashString(session, hashString);
        if (tr_isTorrent(tor) && tr_torrentAllowsLPD(tor)) {
            /* we found a suitable peer, add it to the torrent */
            tr_peerMgrAddPex(tor, TR_PEER_FROM_LPD, peer, -1);
            tr_logAddTorDbg(tor, "Learned %d local peer from LPD (%s:%u)", 1,
                    tr_address_to_string(&peer->addr), peerPort);
            (...)
            return 1;
        }
        (...)
    }

    return res;
}
```

Da mesma forma, ele envia mensagens frequentemente para o grupo, para avisar da sua existência na rede.

./libtransmission/tr-lpd.c:425

```
/**
 * @brief Announce the given torrent on the local network
 *
 * @param[in] t Torrent to announce
 * @return Returns true on success
 *
 * Send a query for torrent t out to the LPD multicast group (or the LAN, for that
 * matter). A listening client on the same network might react by adding us to his
 * peer pool for torrent t.
 */
bool tr_lpdSendAnnounce(const tr_torrent* t) {
    size_t i;
    const char fmt[] = "BT-SEARCH * HTTP/1.1\r\n" // BT-SEARCH * HTTP/1.1
    "Host: %s:%u\r\n"                             // Host: 239.192.152.143:6771
    "Port: %u\r\n"                                // Port: <porta do Transmission>
    "Infohash: %s\r\n"                            // Infohash: <hash do torrent>
    "\r\n"
    "\r\n";

    // info_hash do torrent
    char hashString[lengthof(t->info.hashString)];

    // mensagem de announce multicast montada
    char query[lpd_maxDatagramLength + 1] = { };

    (...)

    /* prepare a zero-terminated announce message */
    tr_snprintf(query, lpd_maxDatagramLength + 1, fmt, (...), "239.192.152.143", 6771,
        lpd_port, hashString); // porta de conexão do Transmission e hash do torrent

    /* actually send the query out using [lpd_socket2] */
    {
        const int len = strlen(query);

        /* destination address info has already been set up in tr_lpdInit (),
         * so we refrain from preparing another sockaddr_in here */
        int res = sendto(lpd_socket2, query, len, 0,
                (const struct sockaddr*) &lpd_mcastAddr, sizeof lpd_mcastAddr);

        if (res != len) return false;
    }
    tr_logAddTorDbg(t, "LPD announce message away");
    return true;
}
```

4.7 Configuração e roteamento de pacotes em rede

Atualmente, o uso de roteadores de rede para uso doméstico é bastante comum, servindo para distribuição de uma conexão de Internet para vários equipamentos eletrônicos que conseguem acessá-la. Para que ocorra essa distribuição de dados de Internet é necessário que, no roteador, funcione um protocolo chamado NAT (*Network Address Translator*).

Um roteador em que funciona um serviço de NAT parece, externamente (para a rede exterior), que é um único dispositivo com o seu endereço IP [61], enquanto internamente, é visto como

o responsável por rotear os dados e abstrair a conexão de Internet externa. Do ponto de vista do roteador, ele conhece os endereços IP de cada dispositivo da rede interna e o endereço IP do modem do ISP. Assim, ele contrói uma tabela de tradução de endereços onde, para cada endereço da rede interna, associa uma porta de rede interna e outra externa.

Então, quando um dos dispositivos envia dados para a Internet, esse pacote passa pelo roteador, que troca o endereço IP e porta do dispositivo de origem, contidos no datagrama, pela respectiva tradução da rede externa, e então repassa o pacote para a Internet. Analogamente, quando um pacote da Internet chega ao roteador, este verifica a porta de conexão de destino, contida no datagrama, e a procura em sua tabela de tradução. Se houver um dispositivo da rede interna associado a essa porta, o roteador troca o endereço e porta de destinos do datagrama e repassa o pacote para tal dispositivo.

Enquanto o NAT parece solucionar um problema, ele cria outro: peers de redes P2P necessitam saber as portas com que estão se comunicando, para informar a outros peers. Porém, um programa cliente que esteja sendo executado em um dispositivo, que está numa sub-rede atendida por NAT, deve saber informar qual porta externa está associada a ele na tabela de NAT, e não a que utiliza no dispositivo.

Para resolver esse problema, existem dois protocolos diferentes de configuração de portas: o NAT PMP (NAT *Port Mapping Protocol*), que atualmente é chamado de PCP (*Port Control Protocol*) [135], e o UPnP (*Universal Plug and Play*) [19]. Ambos os protocolos têm a mesma função: configurar um NAT e conhecer as portas externas que ele lhes reservou, fazendo a travessia de NAT (*NAT Traversal*).

O UPnP [127] é um conjunto de protocolos que possibilita a comunicação entre dispositivos variados, a partir da conexão destes a uma rede, estabelecendo serviços. Entre os diversos protocolos, está o IGDP (*Internet Gateway Device Protocol*), no qual é possível realizar várias ações, desde o conhecimento do endereço IP externo do roteador, conhecer o mapeamento de portas internas e externas, e adicionar ou remover entradas nesse mapeamento. Ao adicionar uma entrada, é possível realizar a travessia de NAT. O UPnP utiliza as portas 1900 para o pacotes UDP, e 2869 para portas TCP.

Já o PCP [121], introduzido em 2005 pela Apple como alternativa ao IGDP, serve somente para configuração de travessia de um NAT e conhecimento do seu endereço externo, automatizando a configuração de redirecionamento de portas em roteadores. Para isso, utiliza a porta 5351 para pacotes UDP.

Essa praticidade de não precisar se configurar manualmente um roteador tem um preço. Ambos os protocolos não são totalmente seguros, fazendo com que um roteador que permita configurações através deles possa oferecer meios de se invadir essas redes por conexões externas. Assim, para efeitos de segurança, um NAT não substitui *firewalls*.

O Transmission utiliza duas bibliotecas, uma para cada protocolo: o MiniUPnP [13] e o

libnatpmp [12]. Ambas são desenvolvidas por Thomas Bernard, e de código aberto.

4.8 IPv6

Com o desenvolvimento da ARPAnet, a quantidade de computadores conectados na rede cresceu, chegando a 562 unidades, em 1983. A quarta versão do protocolo TCP/IP, o IPv4, criado em 1981 [90], foi utilizado para organizar aquelas redes já formadas e ordenar o crescimento posterior.

O IPv4 tem dois objetivos: prover a fragmentação dos datagramas maiores em partes menores, para que pudessem ser enviadas pela camada de enlace; e regras de endereçamento, para que os datagramas tivessem os endereços de origem e de destino armazenados em seus cabeçalhos [56]. Apesar de ser considerada robusta e de fácil implantação e interoperabilidade, o projeto original não previu a ocorrência de problemas, tais como o crescimento das redes e das tabelas de roteamento, segurança de dados, prioridade de entrega de tipos específicos de pacote e, o mais grave, o esgotamento de endereços IP.

O endereçamento do IPv4 é feito com 4 bytes, geralmente representado na forma decimal (4 números de 0 a 255), separados por pontos, o que permite 2^32 endereços possíveis. Esses endereços, que são distribuídos pela IANA (*Internet Assigned Numbers Authority*) globalmente para os cinco RIRs (*Regional Internet Registry*, ou Registro Regional de Internet), que então os distribuem localmente para clientes, que incluem provedores de Internet (ISPs) e outras organizações, que então repassam endereços a usuários finais. Esses endereços são os que estão esgotando atualmente. Para resolver esse problema e alguns outros adquiridos com a experiência operacional do IPv4, o IPv6 foi publicado em 1998 [25].

O IPv6 possui várias alterações desde o IPv4, muitas delas evidentes no formato do seu datagrama:

espaço de endereço maior: o tamanho dos endereços passa a ter 128 bits, representados em 8 grupos de 4 dígitos hexadecimais. Com isso, a quantidade total de endereços possíveis passa de 2^{32} para 2^{128}. Isso também permitiu que fossem definidas três metodologias de roteamento: o *unicast* (para um endereço em específico), o *multicast* (para vários endereços simultaneamente) e o *anycast* (para que, dentre um grupo de potenciais destinatários, apenas um receba);

cabeçalho de 40 bytes fixos: o cabeçalho do IPv4 possui alguns campos opcionais, porém, alguns deles foram retirados ou sofreram alterações quando migrados para o IPv6. Isso agiliza o processamento dos datagramas; e

classificação de fluxo e prioridade: permite que dados que sejam de fluxo contínuo, como áudio ou vídeo, possam ter tratamento diferenciado no redirecionamento de pacotes e melhor manipulação da qualidade para serviços específicos.

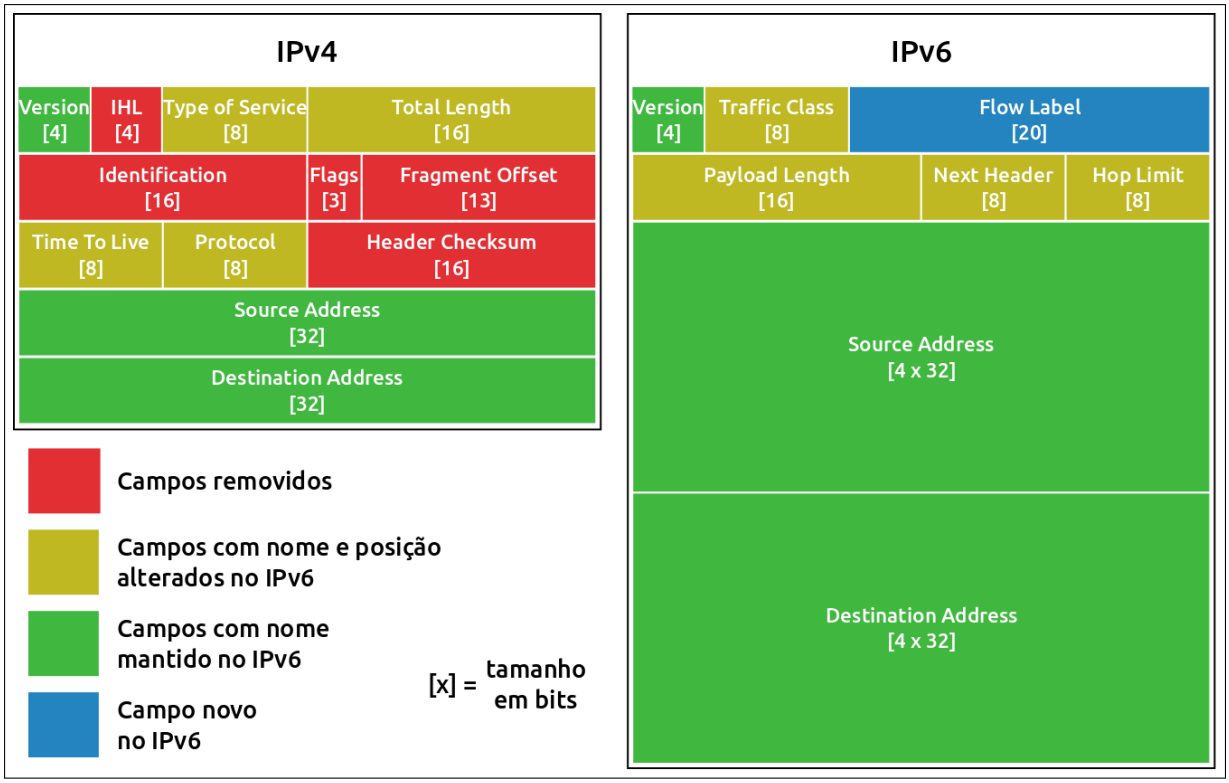

Figura 4.1: *formatos dos datagramas dos protocolos IPv4 e IPv6. Ambos possuem largura de 32 bits.*

Os campos do cabeçalho do IPv6 são:

Version: traz a versão do protocolo utilizado (no caso do IPv6, o valor 6). Porém, o simples envio do valor 4 não implica na utilização do IPv4, não possuindo compatibilidade anterior;

Traffic Class: campo de 8 bits que possui a mesma finalidade do campo *Type of Service* no IPv4, que especifica que o datagrama possui dados que requerem tráfego com baixo atraso, alta vazão ou confiabilidade;

Flow Label: campo de 20 bits usado na classificação de fluxo de datagramas;

Payload Length: campo de 16 bits que representa um inteiro positivo para o tamanho do conjunto de dados anexado após o cabeçalho;

Next Header: identifica o protocolo usado para entregar os dados (TCP ou UDP). É usado da mesma forma que no IPv4;

Hop Limit: quantidade máxima de roteadores pelos quais o datagrama poderá passar, sendo decrescido de 1 cada vez que passa por algum. Quando chega a 0, o datagrama é descartado; e

Source Address e Destination Address: endereços IPv6 de origem e de destino do datagrama, em alguma das várias representações possíveis definidas no RFC4291 [52].

Duas ausências significativas foram as dos campos que definem a fragmentação dos datagramas e do *checksum* do cabeçalho. No primeiro, a funcionalidade foi movida dos roteadores, a fim de reduzir o trabalho deles e melhorar a velocidade do encaminhamento IP pela rede, tornando os sistemas finais responsáveis pela escolha do tamanho dos datagramas. Já no *checksum*, os protocolos das camadas de transporte (TCP e UDP) e de enlace (Ethernet) já realizam a verificação de erros por soma. Por isso, a redundância de trabalho foi retirada, eliminando a necessidade de recalcular essa soma a cada roteador e, assim, melhorando a velocidade de processamento de pacotes IP.

Apesar de não ser retrocompatível com o IPv4, o IPv6 consegue ser transmitido utilizando **tunelamento automático** por **mecanismos de transição**, que proporciona uma camada transparente para que pacotes IPv6 transitem em uma rede IPv4. Entre esses mecanismos estão o 6to4, 6in4 e Teredo.

Uso do IPv6

O IPv6 tem duas datas marcantes: o Dia Mundial e o Dia do Lançamento Mundial. O Dia Mundial do IPv6, ou *World IPv6 Day*, organizado pelo *Internet Society* (ISOC) [5], ocorreu no dia 8 de junho de 2011, e contou com a participação de grandes empresas estrangeiras (Facebook, Google, Yahoo!, Akamai, Lamelight Networks, etc) e brasileiras (Terra, Campus Party, IG), num teste de 24 horas de oferecimento de conexão e conteúdo utilizando o protocolo IPv6. O evento foi considerado um sucesso pela ISOC.

Já o Dia de Lançamento Mundial do IPv6, em dia 6 de junho de 2012, foi um evento semelhante ao Dia Mundial, com o mesmo objetivo e com mais participantes, porém, mantendo o IPv6 habilitado após o evento [136]. Desde então, empresas do mundo todo têm corrido contra o relógio para que o oferecimento de conexões de Internet não seja estagnado pela falta de endereços.

O IPv6 precisa ser suportado pelas pontas das redes (aplicações em clientes e servidores) e pelas rotas entre eles (provedores de acesso, empresas de telecomunicações). Enquanto o primeiro é simples, pois bastam configurações de conectores dos respectivos sistemas operacionais, servidores de aplicação e roteadores internos que suportem o IPv6, o caminhos entre ambos é mais complicado. Para as empresas provedoras de acesso, envolve custos de troca de equipamentos de infraestrutura de rede, além do treinamento da mão de obra. Por conta disso, a transição para o novo protocolo ainda é complexa e lenta.

A situação ainda está devagar no contexto brasileiro do IPv6. Segundo o SindiTelebrasil, em novembro de 2013 [92], as operadoras brasileiras ainda estavam adquirindo equipamentos. O cenário era considerado preocupante e, devido ao atraso da migração tecnológica, medidas paliativas estão sendo tomadas.

Uma delas é o compartilhamento de endereços IPv4 por usuários, com restrição de portas, o que pode afetar a experiência de uso de alguns tipos de aplicações. Esse compartilhamento é feito numa estrutura com dois NATs, usando o modelo chamado de NAT444.

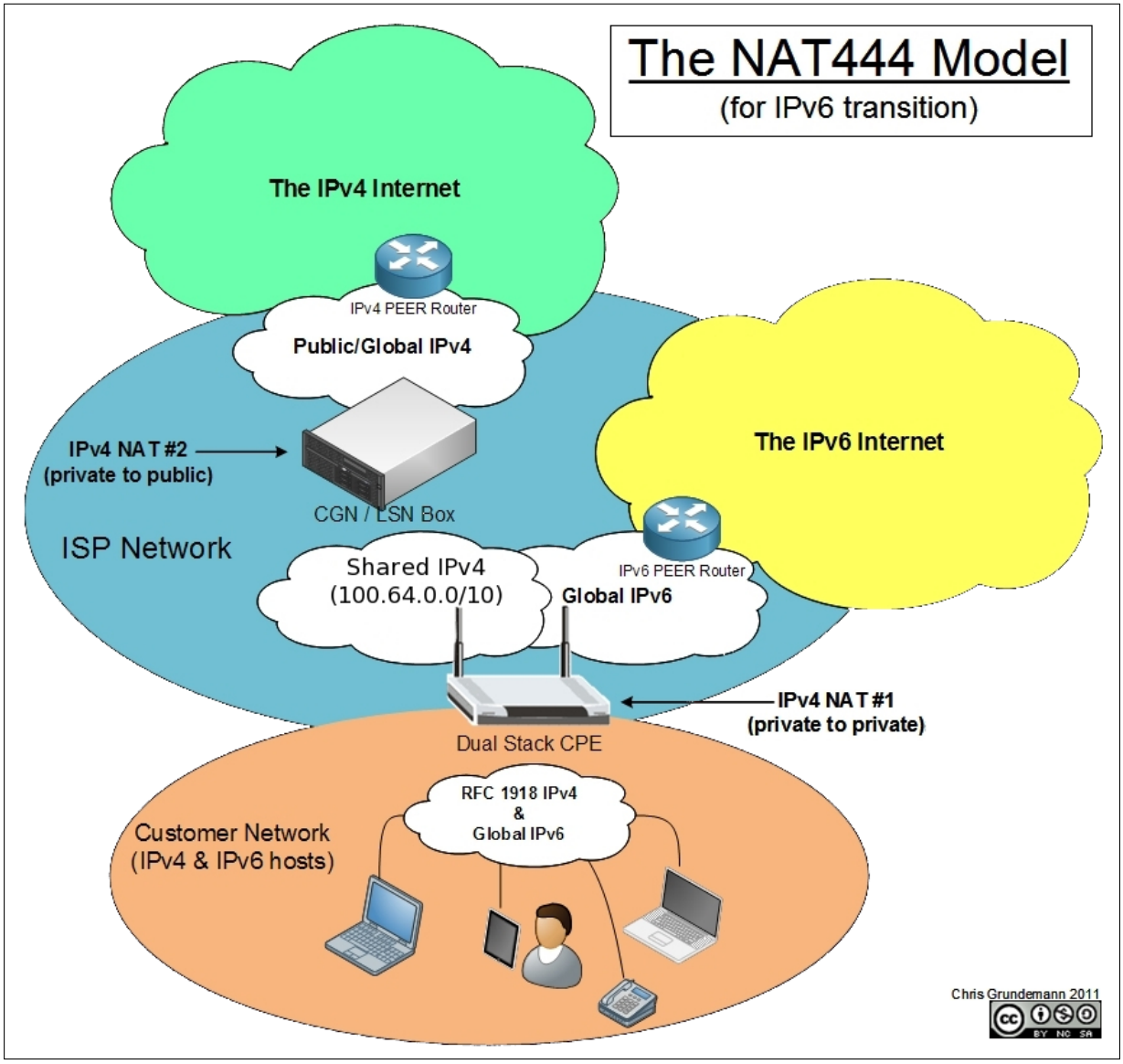

Figura 4.2: *Esquema de uso do NAT444. Fonte:[47]*

No NAT444 [47], dois NATs são usados, e é possível perceber que:

- NAT444 de IPv4 funcionando em pilha dupla (*dual stacked*; para ambos os protocolos) com IPv6 global (público); e
- NAT444 adiciona uma segunda camada de NAT, ou seja, uma segunda área de endereçamento "interno" (privada).

Ao adicionar a segunda camada, vários usuários podem ter um mesmo endereço, ou seja, o

ISP deverá tomar medidas de como registrar os acessos com as respectivas portas de entrada e saída, por questões legais. Outro fato é que deve-se aceitar que o segundo NAT não será configurável por UPnP, por NAT-PMP ou outros protocolos de travessia de NAT. Um ISP não permitirá que ocorra essa configuração, pois existirá o risco de um usuário afetar serviços de outros. Por esse motivo, é possível prever aplicações que podem ser afetadas com problemas de desempenho, ou mesmo incapacidade de execução [30]:

- grandes downloads por FTP;
- *seeding* em Limewire e BitTorrent;
- jogos online (Xbox, Playstation, etc);
- *streaming* de vídeo (Hulu, Netflix, etc);
- acesso remoto de webcam;
- tunelamento para IPv6 (6to4, Teredo, etc);
- VPN e criptografia (IPSec, SSL);
- VoIP.

4.9 Conexão com a Internet

Programas que se conectam à redes ou à Internet são comuns atualmente. Para isso, a linguagem C permite escrever programas que criam conexões de Internet via sockets [48].

Através de um socket, a aplicação consegue enviar mensagens para a camada de transporte, que as transforma em segmentos e as repassa para as camadas inferiores, que transmitem os dados.

Quando o protocolo TCP for usado, os passos para se criar uma conexão serão:

1. configuração do socket em uma variável **sadd** do tipo `struct sockaddr_in` (para IPv4) ou `struct sockaddr_in6` (para IPv6), que conterá informações do computador de destino da conexão.

   Enquanto isso, ocorre a criação de um socket usando a função `socket()`. Essa função, passando o parâmetro `SOCK_STREAM`, pede para o SO (sistema operacional) a criação de um descritor de arquivo especial para socket TCP, que será usado para o fluxo de dados da conexão;

2. ligação do socket à variável **sadd** usando a função `bind()`, que serve para registrar no SO que ele deve manipular os dados que chegam na porta indicada em **sadd**, usando o socket indicado;

3. aguardar conexões no socket usando a função `listen()`; e

4. receber dados pelo socket usando a função `accept()`.

./libtransmission/net.c:317

```c
static int tr_netBindTCPImpl(const tr_address * addr, tr_port port, bool suppressMsgs,
    int * errOut)
{
    static const int domains[NUM_TR_AF_INET_TYPES] = { AF_INET, AF_INET6 };
    struct sockaddr_storage sock; int fd, addrlen, optval;
    (...)

    // criação do descritor de arquivo para socket para TCP (SOCK_STREAM)
    fd = socket(domains[addr->type], SOCK_STREAM, 0);
    (...)

    // parte principal do ./libtransmission/net.c:196
    // É executada neste ponto, durante a função tr_netBindTCPImpl, avaliando o protocolo...
    // ...de conexão a ser usado (IPv4 ou IPv6)
    if (addr->type == TR_AF_INET) {
        // configura os dados do computador de destino para conexão por IPv4

        struct sockaddr_in  sock4;
        memset (&sock4, 0, sizeof (sock4));
        sock4.sin_family      = AF_INET;         // família de endereçamento
        sock4.sin_addr.s_addr = addr->addr.addr4.s_addr; // endereço IPv4
        sock4.sin_port        = htons(port);     // porta de conexão em "big endian"
        memcpy (sock, &sock4, sizeof (sock4));   // grava os dados em 'sock'
        addrlen =  sizeof (struct sockaddr_in);
    }
    else {
        // configura os dados do computador de destino para conexão por IPv6
        struct sockaddr_in6 sock6;
        memset (&sock6, 0, sizeof (sock6));
        sock6.sin6_family   = AF_INET6;          // família de endereçamento
        sock6.sin6_port     = htons(port);       // porta de conexão em "big endian"
        sock6.sin6_flowinfo = 0;                 // info de fluxo IPv6
        sock6.sin6_addr     = addr->addr.addr6;  // endereço IPv6
        memcpy (sock, &sock6, sizeof (sock6));   // grava os dados em 'sock'
        addrlen = sizeof (struct sockaddr_in6);
    }
    // fim da parte principal do ./libtransmission/net.c:196

    // Ligação do socket com a porta indicada
    if (bind(fd, (struct sockaddr *) &sock, addrlen)) {
        (...) // erro de ligação
    }

    (...)
    if (listen(fd, 128) == -1) { // aguarda conexões na porta indicada
        (...) // erro de escuta
    }

    return fd;
}
```

./libtransmission/net.c:417

```
int tr_fdSocketAccept(tr_session * s, int sockfd, tr_address * addr, tr_port * port) {
    int fd; unsigned int len;
    struct tr_fdInfo * gFd; struct sockaddr_storage sock;
    (...)
    len = sizeof(struct sockaddr_storage);

    // recebe dados do socket e armazena os dados lidos em 'sock'
    fd = accept(sockfd, (struct sockaddr *) &sock, &len);
    if (fd >= 0) {
        (...)
    }

    return fd;
}
```

Já quando o protocolo UDP for usado, os passos para se criar uma conexão são exatamente os mesmos que os do protocolo TCP, exceto pelo parâmetro `SOCK_DGRAM` na função `socket()`.

./libtransmission/tr-udp.c:190

```
static void event_callback(int s, short type UNUSED, void *sv) {
    int rc; socklen_t fromlen; unsigned char buf[4096];
    struct sockaddr_storage from; tr_session *ss = sv;
    (...)

    // recebimento de dados do socket 's'
    rc = recvfrom(s, buf, 4096 - 1, 0, (struct sockaddr*) &from, &fromlen);
    (...)

    if (rc > 0) {
        (...) // uso do conteúdo de 'buf'
    }
}

void tr_udpInit(tr_session *ss) {
    (...)
    ss->udp_socket = socket(PF_INET, SOCK_DGRAM, 0); // configuração de socket para UDP
    (...)

    // quando recebe pacotes UDP, chama a função de retorno 'event_callback'
    ss->udp_event = event_new ( (...), event_callback, ss);
}
```

As funções `send()` e `recv()`, que enviam e recebem dados, respectivamente, possuem as suas versões mais complexas `sendto()` e `recvfrom()`, recebendo outro socket como parâmetro de entrada.

4.10 Threads

Threads são procedimentos que são executados de forma independente do programa principal, sujeitos ao escalonamento de processos do sistema operacional. Suas implementações dependem do sistema, porém as linguagens de programação provêm meios de um programador criar e terminar *threads*. Especificamente em sistemas UNIX, uma implementação é a *POSIX* (*Portable Operating System Interface*), que é uma família de padrões da IEEE (*Institute of Electrical and Electronics Engineers*) [9].

Processos contêm várias informações sobre recursos e estados de execução de um programa:

- IDs de processo, de grupo de processo, de grupo de usuário e de usuário;
- ambiente;
- diretório de trabalho;
- registradores;
- pilha de chamada;
- *heap*;
- descritores de arquivos;
- ações de sinal;
- bibliotecas compartilhadas;
- comunicações inter-processos.

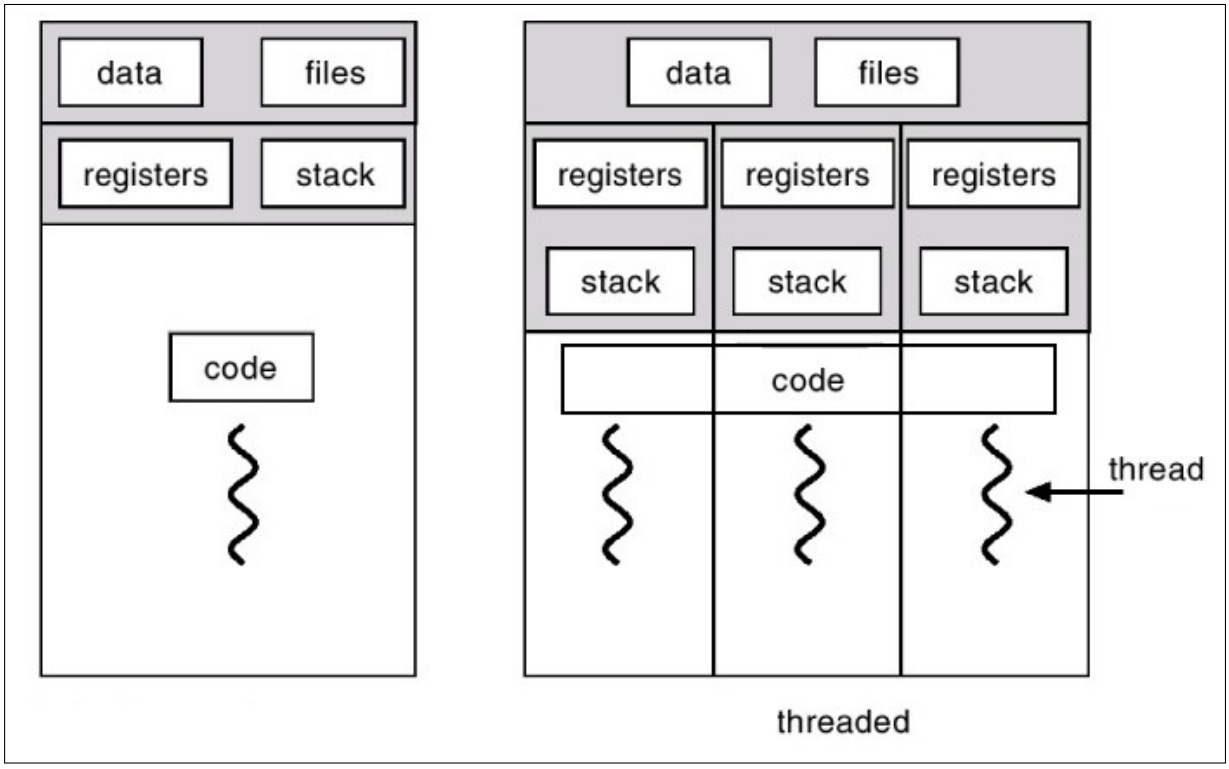

Figura 4.3: *estruturas de processos UNIX, sem e com threads. Fonte:[41]*

Quando uma thread é criada pelo sistema operacional, ela passa a existir dentro dos recursos do processo. Esses recursos poderão ser usados por ela, inclusive de forma compartilhada com outras *threads*. Na sua criação, alguns desses recursos são duplicados para uso próprio, o que permite sua execução independentemente do processo pai, enquanto este existir.

Apesar da praticidade e poder computacional que permite, um problema surge, que é o uso de memória compartilhada. Assim, *threads* precisam ter a leitura e escrita de dados sincronizada, para que se garanta que esses dados já tenham sido preparados por uma thread quando outra quiser acessá-los, por exemplo. O estudo dessas sincronizações e de processamentos independentes entre *threads* é o objetivo da área de Computação Paralela e Concorrente.

O Transmission controla as chamadas de funções a serem executadas em *threads*, usando uma estrutura que contém a função passada e seus parâmetros de entrada.

```
1  // Estrutura de thread do Transmission
2  // ./libtransmission/platform.c:80
3  struct tr_thread {
4      void             (* func)(void *);
5      void              * arg;
6      tr_thread_id       thread;
7  #ifdef WIN32
8      HANDLE             thread_handle;
9  #endif
10 };
```

Então, encapsula a chamada de novas *threads* com a criação do respectivo objeto. Assim, consegue executar tarefas mais complexas sem que a execução do programa tenha seu desempenho prejudicado por travamento. Dois exemplos de uso de criação de *threads* são: para o *bootstrap* da DHT, e para efetuar requisições a trackers.

./libtransmission/platform.c:156

```
1  // Recebe uma função e um ponteiro para seus argumentos, e executa a função dada em...
2  // ...uma nova thread.
3  tr_thread * tr_threadNew(void (*func)(void *), void * arg) {
4      tr_thread * t = tr_new0(tr_thread, 1);  // aloca espaço para uma estrutura 'tr_thread'
5
6      t->func = func;
7      t->arg = arg;
8
9  #ifdef WIN32    // Se o Transmission for utilizado em um sistema operacional Windows...
10 {                // ...usa os comandos de thread específicos para ele.
11     unsigned int id;
12     t->thread_handle = (HANDLE) _beginthreadex (NULL, 0, &ThreadFunc, t, 0, &id);
13     t->thread = (DWORD) id;
14 }
15 #else           // ...caso contrário, use pthread, que é padrão do Mac OS e Linux
16     pthread_create(&t->thread, NULL, (void* (*)(void*)) ThreadFunc, t);
17     pthread_detach(t->thread);
18 #endif
19     return t;
20 }
```

Outro recurso que o Transmission utiliza são as travas (*locks*), que servem para controlar o acesso a uma variável durante um programa de várias *threads*. Com isso, garante que a escrita em uma variável não ocorra simultaneamente à uma leitura.

./libtransmission/platform.c:139

```
/** @brief portability wrapper around OS-dependent thread mutexes */
struct tr_lock {
    int depth;
#ifdef WIN32
    CRITICAL_SECTION lock;
    DWORD lockThread;
#else
    pthread_mutex_t lock;
    pthread_t lockThread;
#endif
};

tr_lock* tr_lockNew(void) {
    tr_lock * l = tr_new0(tr_lock, 1);

#ifdef WIN32
    InitializeCriticalSection (&l->lock); /* supports recursion */
#else
    pthread_mutexattr_t attr;
    pthread_mutexattr_init(&attr);
    pthread_mutexattr_settype(&attr, PTHREAD_MUTEX_RECURSIVE);
    pthread_mutex_init(&l->lock, &attr);
#endif
    return l;
}

void tr_lockFree(tr_lock * l) {
#ifdef WIN32
    DeleteCriticalSection (&l->lock);
#else
    pthread_mutex_destroy(&l->lock);
#endif
    tr_free(l);
}

void tr_lockLock(tr_lock * l) {
#ifdef WIN32
    EnterCriticalSection (&l->lock);
#else
    pthread_mutex_lock(&l->lock);
#endif
    (...)
    l->lockThread = tr_getCurrentThread();
    ++l->depth;
}
```

4.11 Engenharia de Software

O Transmission é um programa bastante complexo e grande. Por exemplo, todos os arquivos que são usados na versão GTK+ somam pouco mais de 86 mil linhas. Desenvolvido por cinco programadores em nove anos, possui versões de vários tipos e sistemas operacionais. Um programa desse porte poderia usar conhecimentos de Engenharia de Software para questões de manutenção de código, melhoria da qualidade de desenvolvimento, enfim, todos os aspectos da produção do programa. Alguns desses aspectos são notáveis pelos seus recursos, entre eles o site, fórum de usuários, código do programa, etc.

modularização: o Transmission possui o código organizado de forma que a funcionalidade

está separada e independente da implementação da interface gráfica (*front end*). Isso permite que o programa funcione da mesma forma para todas as versões desenvolvidas, já que possui o mesmo núcleo de funcionamento (*back end*).

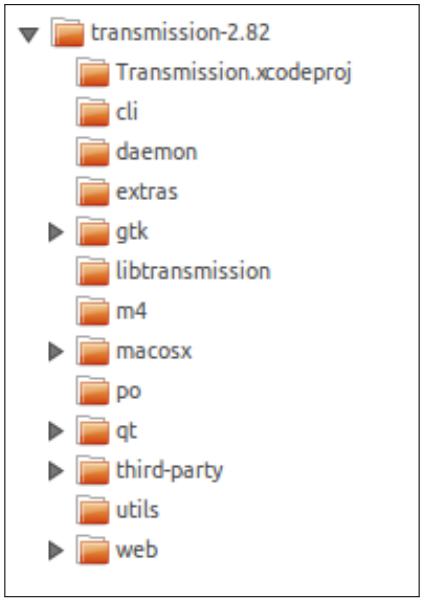

Figura 4.4: *estruturas de pastas do Transmission*

As implementações visuais do Transmission utilizam os *frameworks* GTK+ [93], Qt [81] e ncurses [3], além do modo de linha de comando;

multiplataforma: tendo seu código modularizado, basta uma ferramenta que saiba compilá-lo de acordo com as ferramentas que o computador possui. Para isso, utiliza as ferramentas: Autoconf [6], que gera os arquivos de configuração da compilação, dependendo do sistema operacional e dos softwares básicos instalados; e Automake [7], que gera o arquivo de compilação utilizando o de configuração gerado pelo Autoconf;

comentários de código: boa parte do código do Transmission não possui comentários, o que muitas vezes dificultou o entendimento de seu funcionamento;

testes unitários: são códigos que executam verificações da corretude de uma parte específica de código, geralmente no nível de funções. Existem alguns testes unitários escritos, mas não para toda implementação de funcionalidade. Por exemplo, o código de leitura e decodificação de magnet link;

./libtransmission/metainfo-test.c

```
// Arquivo completo de teste de leitura e decodificação de links magnéticos.
#include "transmission.h"

#include "libtransmission-test.h"

static int test1(void) {
    tr_info inf;
    tr_ctor * ctor;
    const char * magnet_link;
    tr_parse_result parse_result;

    /* background info @ http://wiki.theory.org/BitTorrent_Magnet-URI_Webseeding */
    magnet_link = "magnet:?"
            "xt=urn:btih:14FFE5DD23188FD5CB53A1D47F1289DB70ABF31E"
            "&dn=ubuntu+12+04+1+desktop+32+bit"
            "&tr=http%3A%2F%2Ftracker.publicbt.com%2Fannounce"
            "&tr=udp%3A%2F%2Ftracker.publicbt.com%3A80"
            "&ws=http://transmissionbt.com ";
    ctor = tr_ctorNew(NULL);
    tr_ctorSetMetainfoFromMagnetLink(ctor, magnet_link);
    parse_result = tr_torrentParse(ctor, &inf);
    check_int_eq(inf.fileCount, 0); /* cos it's a magnet link */
    check_int_eq(parse_result, TR_PARSE_OK);
    check_int_eq(inf.trackerCount, 2);
    check_streq("http://tracker.publicbt.com/announce", inf.trackers[0].announce);
    check_streq("udp://tracker.publicbt.com:80", inf.trackers[1].announce);
    check_int_eq(inf.webseedCount, 1);
    check_streq("http://transmissionbt.com", inf.webseeds[0]);

    /* cleanup */
    tr_metainfoFree(&inf);
    tr_ctorFree(ctor);
    return 0;
}

MAIN_SINGLE_TEST(test1)
```

internacionalização: de forma semelhante ao que é feito atualmente com o desenvolvimento de aplicativos para celulares, o Transmission é desenvolvido com seus textos escritos em arquivos externos, um para cada linguagem, facilitando o processo de tradução do programa;

código aberto e livre: o código do Transmission é acessível a todos, podendo ser portado a outras plataformas. Além disso, qualquer um pode participar do seu desenvolvimento, bastando entrar em contato com os atuais programadores.

Capítulo 5

Transmission e o BCC

Neste texto, descrevemos que são encontrados vários elementos de Ciência da Computação no programa cliente Transmission. A intenção agora é encontrar quais disciplinas da grade curricular do Bacharelado em Ciência da Computação (BCC) tem conhecimentos utilizados no Transmission.

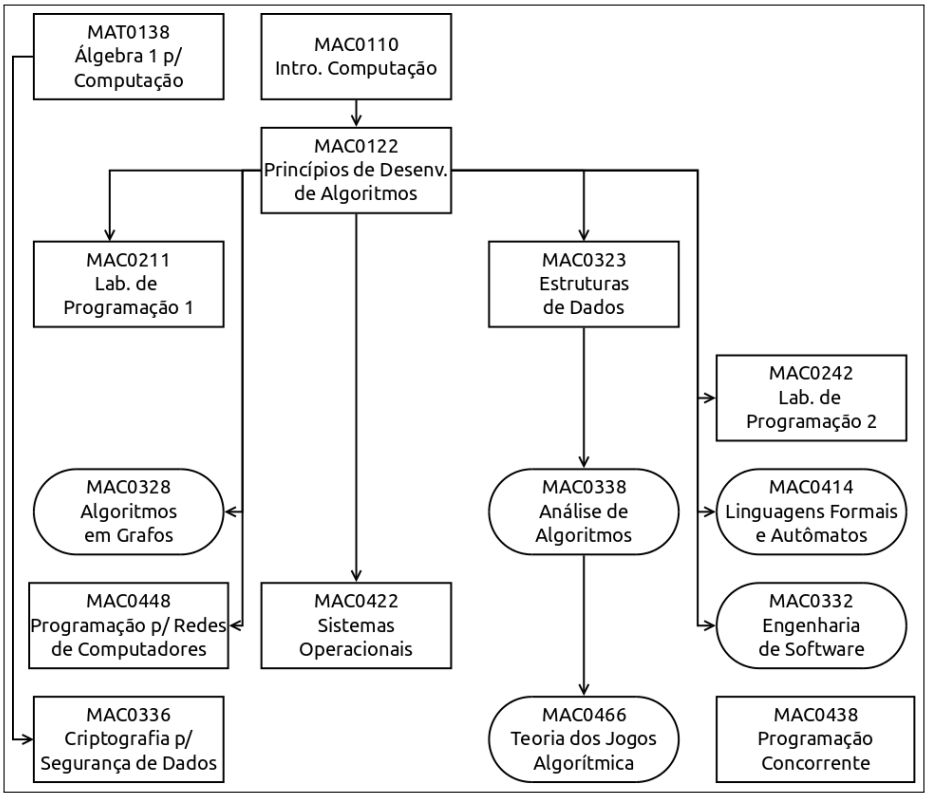

Figura 5.1: *disciplinas do BCC utilizadas diretamente na programação do Transmission, ligadas pelos seus pré-requisitos. As bordas arredondadas indicam que o conhecimento auxilia, mas não é necessário.*

Das disciplinas do BCC, as reconhecidas como necessárias para o entendimento do BitTorrent e o Transmission foram:

- Introdução à Computação (MAC0110), Princípios de Desenvolvimento de Algoritmos (MAC0122), Laboratórios de Programação 1 (MAC0211): linguagem C, organização de código, trabalho gerenciado em grupos de desenvolvedores, portabilidade de código entre plataformas (Autoconf e Automake);
- Estruturas de Dados (MAC0323): utilização e manipulação de estruturas de dados, como vetores e listas ligadas;
- Algoritmos em Grafos (MAC0328): entendimento do algoritmo de busca de nós no DHT;
- Análise de Algoritmos (MAC0328) e Teoria dos Jogos Algorítmica (MAC0446): algoritmo da troca das partes entre peers;
- Álgebra 1 (MAT0138) e Criptografia para Segurança de Dados (MAC0336): noções de álgebra modular e seus usos nos métodos criptográficos;
- Laboratório de Programação 2 (MAC0242) e Engenharia de Software (MAC0332): desenvolvimento de programas complexos;
- Programação para Redes (MAC0448): desenvolvimento de código de conexão via Internet, como TCP e UDP, multicast, NAT, IPv6, UPnP e NAT-pmp;
- Linguagens Formais e Autômatos (MAC0414): uso de autômatos para entendimento dos estados dos peers e suas transições com as trocas de mensagens [62]; e
- Sistemas Operacionais (MAC0422) e Programação Concorrente (MAC0438): uso de *threads* e métodos para computação com seus usos, como acesso à memória compartilhada (*mutex*) e travas, e entendimento de caches de memória na leitura e escrita de dados.

Capítulo 6

Comentários Finais

Este trabalho abordou o protocolo BitTorrent, entendendo a sua especificação, e usar o código do Transmission foi um excelente exemplo, servindo como ponte entre teoria e prática. Esse tipo de estudo foi muito esclarecedor em vários aspectos, permitindo conhecer melhor o protocolo e saber um pouco mais de cada um dos tópicos abordados, que ainda são bastante atuais. Entre estes, estão desde a própria linguagem C até as teorias relacionadas, como a teoria dos jogos, métodos criptográficos utilizados, estruturas de dados e os conhecimentos de redes.

As disciplinas da grade curricular que foram mais importantes para o entendimento do BitTorrent em sua totalidade foram:

- MAC0211 - Laboratório de Programação 1
- MAC0242 - Laboratório de Programação 2
- MAC0323 - Estruturas de Dados
- MAC0332 - Engenharia de Software
- MAC0336 - Criptografia para Segurança de Dados
- MAC0438 - Programação Concorrente
- MAC0448 - Programação para Redes

A disciplina Estruturas de Dados (MAC0323), apesar de sua importância, não é necessária para compreender o protocolo, pois este independe da linguagem de programação. Essa disciplina, por ter foco em linguagem C, auxilia somente programas escritos nessa linguagem, assim como Laboratório de Programação 1 (MAC0211).

Paralelamente, as disciplinas de Laboratórios de Programação 1 (MAC0211) e 2 (MAC0242) e Engenharia de Software (MAC0332) são fundamentais, pois desenvolvem a habilidade de

construir programas com muitos componentes e abstrações. Especificamente, as duas últimas geralmente são ministradas em linguagens orientadas a objeto, como Java ou ActionScript que justificam o fato da disciplina de Estruturas de Dados não ser requerida.

Apesar dos estudos que ocorreram antes do início do trabalho, de artigos que analisaram o protocolo, não foram feitas as análises dos resultados desses artigos e o impacto deles no desenvolvimento do BitTorrent.

tentar melhorar isso aqui

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 fornece uma lista de outros peers conectados naquele momento. 18, 19, 21, 23–25, 83, 92, 118

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. 7

API
do inglês *Application Programming Interface*, Interface de Programação de Aplicativos; especificação de como um conjunto de tipos e de procedimentos devem interagir com outras partes de um software. 78

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. 11, 13, 14, 16–19, 23, 30, 40, 42, 49, 50, 76, 78, 92, 119

Audiogalaxy
Rede P2P de compartilhamento de músicas MP3, criado em 1998. 5, 6

bencode
"codificação B", pronunciado *bê encode*; formato de codificação compacta de arquivos torrent para transmissão de metadados. 14, 19, 21–23, 33, 34, 40, 42, 44

beta tester
usuários de uma versão beta de um software. 9

bootstrap
objeto que serve para ajudar a calçar botas. Termo é usado como metáfora com o significado de execução de um procedimento sem ajuda externa. 44, 45, 107

bucket
. 32, 36, 37, 44, 45

escrever durante a revisão

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, `FOO`, `Foo` e `foo` seriam strings dadas como iguais. 120

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). 77, 91, 92, 101, 115

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. 8, 28, 29, 33, 34, 42, 45, 46, 51, 57, 76, 78, 107, 112, 117

eDonkey
lançado em 6 de setembro de 2000, é um protocolo que foi inaugurado juntamente com o software que o utilizava — o eDonkey2000 — mas inúmeros softwares cliente para diferentes plataformas surgiram nos dias seguintes ao lançamento. 7, 8

endian
extremidade; ordem de armazenamentos dos bytes em memória. *Big endian*, também chamado de ordem de dados de rede (*network byte order*), é quando o bit mais significativo é guardado no menor endereço, ou seja, em primeiro. *Little endian*, pelo contrário, é quando o bit menos significativo é guardado no menor endereço. 22, 52, 54

função de hash
é uma função ou algoritmo matemático que mapeia um dado de comprimento variável em outro de comprimento fixo. 17, 19, 50, 76–78, 120

Gnutella
software de compartilhamento P2P, desenvolvido por três programadores da empresa Nullsoft, recém adquirida da AOL Inc., lançado no ano 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. 7, 8, 10, 29

HTTP GET
método de requisição do protocolo HTTP, que permite uso de query strings para troca de informações. 19

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. 5, 82, 89, 93, 94, 98, 99, 103, 117

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. 8

***k*-bucket**
. 32, 33, 36, 44–46

escrever durante a revisão

leecher
em português, sugador; nome dado ao peer que ainda não terminou um download de um torrent. 11, 21, 23, 58

magnet link
em português, link magnético; padrão aberto, definido por convenção, de esquema de URI, utilizado para localizar recursos de rede BitTorrent para download. 16–18, 109

metadado
dados sobre outros dados; informação sobre outra informação. 11, 16, 115

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. 5, 6, 115

NAT
do inglês *Network Address Translation*; tradução de endereço de rede, é um protocolo empregado 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. 20, 42, 97, 98, 102, 103, 112

overhead
custo de recursos mais elevado do que o normal, sobrecarga. Na Ciência da Computação, esse custo geralmente é de tempo de processamento ou de consumos de memória ou de banda de rede. 28, 53, 92

P2P
do inglês *peer-to-peer*; redes de arquitetura descentralizada e distribuída, onde cada nó (peer) fornece e consome recursos. 3, 5–7, 9–11, 28, 98, 115–117

peer
em português, significa par, colega; nome que se dá a cada nó da rede, ou seja, a um computador conectado. 7, 8, 10, 11, 13, 15, 17, 19, 20, 22, 23, 25–30, 34, 40, 42, 44–46, 48–55, 57–59, 64–71, 76, 78, 82–86, 89, 94, 95, 98, 112, 115, 117–119

pool
em português, poço de recursos; conjunto de recursos alocados em memória que são pré-computados, a fim de estarem prontos para serem utilizados. 26

proxy
servidor que funciona como intermediário de acessos, repassando requisições de um computador cliente para o servidor de destino. 20

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`. 17, 116

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. 6

RPC
do inglês *Remote Procedure Call*; chamada de procedimento remoto, é uma comunicação interprocessual de um programa de computador que permite a um processo executar procedimentos em outros endereços de rede ou de memória, sem a necessidade de conhecimento detalhadado do programador. 33, 36, 40

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. 23, 24, 26

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. 11, 20–23, 58, 89

socket
soquete; portas de comunicação de dados da camada de aplicação para a camada de rede, que permitem a comunicação de programas com o envio e recebimento de dados transmitidos pela rede para a camada de rede. 90–93, 103–105

string
sequência de caracteres. 1, 14, 16, 17, 19–22, 33, 34, 36, 40, 42, 44, 50, 51, 116, 118–120

swarm
em português, enxame; grupo de peers que estão compartilhando dados de um mesmo torrent num determinado momento. 11–13, 16, 17, 19, 22, 26, 42, 46, 49, 58, 61

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. 8

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*. 8, 28–30, 73, 76, 77, 116

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. 19, 23, 57, 65, 90–94, 98, 100, 101, 103, 105, 112

thread
em português, linha de execução; procedimentos ou funções que são executados de forma independente do programa principal, sujeitos ao escalonamento de processos do sistema operacional. São utilizadas em processamentos paralelos. 26, 105–107, 112

torrent
conjunto de um ou mais arquivos definidos por um arquivo .torrent. 11–13, 15–18, 20, 23, 27, 29, 30, 42, 49–51, 53–55, 58, 59, 61, 64, 66, 76, 78, 82, 83, 89, 115, 118

tracker
em português, rastreador; servidor que funciona como um ponto de encontro de peer. 9, 11, 15, 17–21, 23, 24, 26, 49, 51, 82, 83, 92, 107, 115, 118

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. 19, 23, 32, 33, 42, 90–95, 98, 100, 101, 105, 112

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 à uma página de Internet (nome de esquema `http`). 16, 92, 117, 120

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. 115, 118, 120

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. 17, 19, 50, 83

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. 17

valor hash
: ou *hash*; valores gerados por uma função de hash. 8, 16–19, 23, 30, 40, 42, 50, 53, 76–78, 83, 86

XOR
: operador lógico ou-exclusivo ou disjunção exclusiva, cujo resultado é a diferença entre dois operandos. Na criptografia, é usado em operações bit a bit de ou-exclusivo lógico, que é equivalente à operação de adição módulo 2, onde o resultado da operação entre duas sequências de bits é onde há diferença entre bits na mesma posição. 31, 81

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/ (ver p. 82).

[2] *Amazon Simple Storage Service (Amazon S3) — Amazon S3 Functionality*. [Online; acessado em 7-outubro-2013]. URL: http://aws.amazon.com/s3/#functionality (ver p. 9).

[3] *Announcing ncurses 5.9*. [Online; accessed 30-January-2014]. URL: http://www.gnu.org/software/ncurses/ (ver p. 109).

[4] 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 (ver p. 10).

[5] *Archive: 2011 World IPV6 Day*. [Online; accessed 28-January-2014]. URL: http://internetsociety.org/ipv6/archive-2011-world-ipv6-day (ver p. 101).

[6] *Autoconf - GNU Project - Free Software Foundation (FSF)*. [Online; accessed 30-January-2014]. URL: http://www.gnu.org/software/autoconf/ (ver p. 109).

[7] *Automake - GNU Project - Free Software Foundation (FSF)*. [Online; accessed 30-January-2014]. URL: http://www.gnu.org/software/automake/ (ver p. 109).

[8] 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.

[9] Blaise Barney. *POSIX Threads Programming*. [Online; accessed 30-January-2014]. 1 de nov. de 2013. URL: https://computing.llnl.gov/tutorials/pthreads/ (ver p. 105).

[10] Georg Becker. “Merkle Signature Schemes, Merkle Trees and Their Cryptanalysis”. Em: (18 de jul. de 2008). [Online; accessed 26-January-2014]. URL: http://www.emsec.rub.de/media/crypto/attachments/files/2011/04/becker_1.pdf (ver p. 77).

[11] Richard Bennett. *Bittorrent declares war on VoIP, gamers*. [Online; accessed 25-January-2014]. 1 de dez. de 2008. URL: http://www.theregister.co.uk/2008/12/01/richard_bennett_utorrent_udp/ (ver p. 93).

[12] Thomas Bernard. *Github - libnatpmp*. [Online; accessed 27-January-2014]. URL: https://github.com/miniupnp/libnatpmp (ver p. 99).

[13] Thomas Bernard. *Github - MiniUPnP*. [Online; accessed 27-January-2014]. URL: https://github.com/miniupnp/miniupnp (ver p. 98).

[14] 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 (ver p. 7).

[15] *BitTorrent*. URL: http://www.bittorrent.com/ (ver p. 9).

[16] S. Blake et al. *RFC2475: An Architecture for Differentiated Services*. Dez. de 1998. URL: http://tools.ietf.org/html/rfc2475 (ver p. 89).

[17] *Blizzard Downloader — Wowpedia*. [Online; acessado em 5-outubro-2013]. 2013. URL: http://wowpedia.org/index.php?title=Blizzard_Downloader&oldid=3198520 (ver p. 10).

[18] Karl Bode. *UDP BitTorrent Will Destroy The Interwebs! ..says Richard Bennett. 'Sensationist nonsense,' BitTorrent tells us...* [Online; accessed 25-January-2014]. 1 de dez. de 2008. URL: http://www.dslreports.com/shownews/UDP-BitTorrent-Will-Destroy-The-Interwebs-99400 (ver p. 94).

[19] M. Boucadair e R. Penno. *RFC6970: Universal Plug and Play (UPnP) Internet Gateway Device - Port Control Protocol Interworking Function (IGD-PCP IWF)*. Jul. de 2013. URL: http://tools.ietf.org/html/rfc6970 (ver p. 98).

[20] *CBC to BitTorrent Canada's Next Great Prime Minister*. URL: http://archive.is/VYmFD (ver p. 9).

[21] Juliusz Chroboczek. *BitTorrent DHT library*. [Online; accessed 3-January-2014]. URL: http://github.com/jech/dht (ver p. 32).

[22] *Codecon 2002 — Schedule*. 2002. URL: http://web.archive.org/web/20021012072819/http://codecon.org/2002/program.html#bittorrent (ver p. 8).

[23] Bram Cohen. *Incentives Build Robustness in BitTorrent*. 2003 (ver p. 58).

[24] T. H. Cormen et al. *Introduction to Algorithms*. 3rd. The MIT Press, 2009. ISBN: 978-0-262-03384-8 (ver pp. 73, 76).

[25] S. Deering e R. Hinden. *RFC2460: Internet Protocol, Version 6 (IPv6) Specification*. [Online; accessed 28-January-2014]. Dez. de 1998. URL: http://tools.ietf.org/html/rfc2460 (ver p. 99).

[26] 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 (ver p. 9).

[27] *DGM Live — FAQ*. [Online; acessado em 7-outubro-2013]. URL: http://www.dgmlive.com/help.htm#whatisbittorrent (ver p. 9).

[28] 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. URL: http://dx.doi.org/10.1109/TIT.1976.1055638 (ver pp. 79, 80).

[29] 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. URL: http://doi.acm.org/10.1145/1499799.1499815 (ver p. 79).

[30] C. Donley et al. *RFC7021: Assessing the Impact of Carrier-Grade NAT on Network Applications*. [Online; accessed 28-January-2014]. Set. de 2013. URL: http://tools.ietf.org/html/rfc7021 (ver p. 103).

[31] *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 (ver p. 6).

[32] Ernesto. *BitTorrent And MPAA join forces*. 23 de nov. de 2005. URL: http://torrentfreak.com/BitTorrent-and-mpaa-join-forces/ (ver p. 9).

[33] 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/ (ver p. 10).

[34] Ernesto. *BitTorrent Sync 7 Times Faster Than Dropbox*. 5 de dez. de 2013. URL: http://torrentfreak.com/bittorrent-sync-7-times-faster-than-dropbox-131205/ (ver p. 135).

[35] 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/ (ver p. 10).

[36] Ernesto. *Judge Understands BitTorrent, Kills Mass Piracy Lawsuits*. 30 de jan. de 2014. URL: http://torrentfreak.com/judge-understands-bittorrent-kills-mass-piracy-lawsuits-140130/ (ver p. 135).

[37] Ernesto. *Twitter Uses BitTorrent For Server Deployment*. 10 de fev. de 2013. URL: http://torrentfreak.com/twitter-uses-bittorrent-for-server-deployment-100210/ (ver p. 10).

[38] 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/ (ver p. 10).

[39] Roy T. Fielding et al. *RFC2616: Hypertext Transfer Protocol - HTTP/1.1*. Jun. de 1999. URL: http://tools.ietf.org/html/rfc2616.

[40] 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.

[41] Ph.D. Carlos G. Gallo. *CSC322 - C Programming and UNIX — UNIX Threads*. [Online; accessed 3-February-2014]. URL: http://web.cs.miami.edu/home/gallo/CSC322/Content/UNIXProgramming/UNIXThreads.shtml (ver p. 106).

[42] 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/.

[43] 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/.

[44] 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 (ver p. 82).

[45] 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 (ver p. 82).

[46] 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 (ver p. 82).

[47] Chris Grundemann. *NAT444 (CGN/LSN) and What it Breaks*. [Online; accessed 28-January-2014]. 14 de fev. de 2011. URL: http://chrisgrundemann.com/index.php/2011/nat444-cgn-lsn-breaks/ (ver p. 102).

[48] Brian "Beej Jorgensen" Hall. *Beej's Guide to Network Programming*. [Online; accessed 29-January-2014]. 3 de jul. de 2012. URL: http://beej.us/guide/bgnet/output/html/multipage/index.html (ver p. 103).

[49] David Harrison e Bram Cohen. *BitTorrent Enhancement Proposals — Fast Extension*. 25 de set. de 2008. URL: http://www.bittorrent.org/beps/bep_0006.html (ver p. 89).

[50] David Harrison et al. *BitTorrent Enhancement Proposals — Tracker Peer Obfuscation*. 29 de abr. de 2008. URL: http://www.bittorrent.org/beps/bep_0008.html (ver p. 82).

[51] 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/ (ver p. 6).

[52] R. Hinden e S. Deering. *RFC4291: IP Version 6 Addressing Architecture*. [Online; accessed 28-January-2014]. Fev. de 2006. URL: http://tools.ietf.org/html/rfc4291 (ver p. 100).

[53] *HPC Data Repository*. [Online; acessado em 7-outubro-2013]. URL: http://www.hpc.fsu.edu/index.php?option=com_wrapper&view=wrapper&Itemid=80 (ver p. 10).

[54] 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 (ver pp. 3, 10, 94).

[55] *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 (ver p. 81).

[56] *IPv6 — Introdução*. [Online; accessed 27-January-2014]. Set. de 1981. URL: http://ipv6.br/entenda/introducao/ (ver p. 99).

[57] Ben Jones. *BitTorrent says Netflix is hogging bandwidth - not 'beating' it*. [Online; accessed 25-January-2014]. 15 de nov. de 2013. URL: http://www.theguardian.com/technology/2013/nov/15/bittorrent-says-netflix-is-hogging-bandwidth-not-beating-it/ (ver p. 94).

[58] Ben Jones. *Will uTorrent Really Kill the Internet?* [Online; accessed 25-January-2014]. 2 de dez. de 2008. URL: http://torrentfreak.com/will-utorrent-really-kill-the-internet-081201/ (ver p. 94).

[59] 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 (ver p. 6).

[60] 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/ (ver p. 6).

[61] James F. Kurose e Keith Ross. *Computer Networking: A Top-Down Approach*. 5nd. Addison-Wesley, 2009. ISBN: 0136079679 (ver pp. 89, 93, 95, 97).

[62] 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 (ver p. 112).

[63] 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.

[64] *Local Peer Discovery Documentation*. [Online; accessed 26-January-2014]. 30 de out. de 2009. URL: https://forum.utorrent.com/viewtopic.php?pid=433785#p433785 (ver p. 95).

[65] Andrew Loewenstern e Arvid Norberg. *BitTorrent Enhancement Proposals — DHT Protocol*. 29 de fev. de 2008. URL: http://www.bittorrent.org/beps/bep_0005.html (ver p. 29).

[66] 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 (ver pp. 29–31, 36).

[67] 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 (ver p. 36).

[68] 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 (ver p. 79).

[69] D. Meyer. *RFC2365: Administratively Scoped IP Multicast*. Jul. de 1998. URL: http://tools.ietf.org/html/rfc2365 (ver p. 95).

[70] *Miro*. [Online; acessado em 7-outubro-2013]. URL: http://getmiro.com (ver p. 9).

[71] *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 (ver pp. 14, 15, 17).

[72] *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 (ver p. 6).

[73] Arvid Norberg. *BitTorrent Tech Talks: DHT*. 22 de jan. de 2013. URL: http://engineering.bittorrent.com/2013/01/22/bittorrent-tech-talks-dht/ (ver p. 31).

[74] Arvid Norberg. *libtorrent*. URL: http://www.libtorrent.org/ (ver p. 46).

[75] 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 (ver p. 46).

[76] *NRKbeta*. [Online; acessado em 7-outubro-2013]. URL: http://nrkbeta.no/bittorrent/ (ver p. 9).

[77] *OpenSSL Cryptography and SSL/TLS Toolkit*. [Online; accessed 17-January-2014]. URL: http://www.openssl.org/ (ver p. 78).

[78] 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. URL: http://doi.acm.org/10.1145/258492.258523 (ver p. 29).

[79] Larry Press. *Analogy between the postal network and TCP/IP*. [Online; accessed 24-January-2014]. URL: http://som.csudh.edu/fac/lpress/471/hout/netech/postofficelayers.htm (ver p. 90).

[80] *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 (ver p. 10).

[81] *Qt Project*. [Online; accessed 30-January-2014]. URL: http://qt-project.org/ (ver p. 109).

[82] *Rhapsody.com*. [Online; acessado em 30-setembro-2013]. URL: http://www.rhapsody.com (ver p. 6).

[83] 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.

[84] R. Rivest. *RPF1320: The MD4 Message-Digest Algorithm*. Rel. téc. 1320. Internet Engineering Task Force, abr. de 1992. URL: http://tools.ietf.org/html/rfc1320 (ver p. 78).

[85] Jim Roskind. *Experimenting with QUIC*. [Online; accessed 25-January-2014]. 27 de jun. de 2013. URL: http://blog.chromium.org/2013/06/experimenting-with-quic.html (ver p. 94).

[86] Jim Roskind. *IPv4 Multicast Address Space Registry*. [Online; accessed 26-January-2014]. 8 de jan. de 2014. URL: http://www.iana.org/assignments/multicast-addresses/multicast-addresses.xhtm (ver p. 95).

[87] Stefan Saroiu, P. Krishna Gummadi e Steven D. Gribble. "A Measurement Study of Peer-to-Peer File Sharing Systems". Em: 2002 (ver p. 33).

[88] Bruce Schneier. *Applied Cryptography (2Nd Ed.): Protocols, Algorithms, and Source Code in C*. New York, NY, USA: John Wiley & Sons, Inc., 1995. ISBN: 0-471-11709-9 (ver p. 80).

[89] 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. URL: http://dx.doi.org/10.1137/S0097539795293172 (ver p. 81).

[90] Information Sciences Institute — University of Southern California. *RFC791: Internet Protocol — DARPA Internet Program Protocol Specification*. [Online; accessed 27-January-2014]. Set. de 1981. URL: http://tools.ietf.org/html/rfc791 (ver p. 99).

[91] *TCP vs UDP*. [Online; accessed 25-January-2014]. URL: http://www.diffen.com/difference/TCP_vs_UDP.

[92] ENTELCO TELECOM. *IPv6 ainda é a minoria no Brasil, mas por pouco tempo*. [Online; accessed 28-January-2014]. 19 de nov. de 2013. URL: http://www.nic.br/imprensa/clipping/2013/midia1712.htm (ver p. 101).

[93] *The GTK+ Project*. [Online; accessed 30-January-2014]. URL: http://www.gtk.org/ (ver p. 109).

[94] Clive Thompson. *The BitTorrent Effect*. Jan. de 2005. URL: http://www.wired.com/wired/archive/13.01/bittorrent.html (ver p. 9).

[95] *Transmission — A Fast, Easy, and Free BitTorrent Client*. [Online; acessado em 31-janeiro-2014]. 2014. URL: http://www.transmissionbt.com/ (ver p. 4).

[96] *Tree Hash EXchange format (THEX)*. [Online; accessed 26-January-2014]. URL: http://web.archive.org/web/20060409093503/www.open-content.net/specs/draft-jchapweske-thex-02.html (ver p. 76).

[97] “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 (ver p. 5).

[98] *UPnP™ Device Architecture 1.1*. [Online; accessed 26-January-2014]. 15 de out. de 2008. URL: http://www.upnp.org/specs/arch/UPnP-arch-DeviceArchitecture-v1.1.pdf (ver p. 95).

[99] *VODO — About*. [Online; acessado em 7-outubro-2013]. URL: http://www.dgmlive.com/help.htm#whatisbittorrent (ver p. 9).

[100] 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.

[101] 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.

[102] 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 (ver p. 6).

[103] 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.

[104] 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 (ver p. 5).

[105] Wikipedia. *BitTorrent — Wikipedia, The Free Encyclopedia*. [Online; accessed 28-October-2013]. 2013. URL: http://en.wikipedia.org/w/index.php?title=BitTorrent&oldid=578877679 (ver p. 17).

[106] 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 (ver p. 9).

[107] Wikipedia. *Cryptography — Wikipedia, The Free Encyclopedia*. [Online; accessed 19-January-2014]. 2014. URL: http://en.wikipedia.org/w/index.php?title=Cryptography&oldid=591122478 (ver p. 79).

[108] 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.

[109] 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 (ver p. 5).

[110] 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 (ver p. 7).

[111] 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 (ver p. 7).

[112] 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.

[113] 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.

[114] 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 (ver p. 5).

[115] 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.

[116] 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 (ver p. 33).

[117] 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 (ver p. 9).

[118] 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 (ver p. 5).

[119] 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 (ver p. 5).

[120] 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 (ver p. 6).

[121] Wikipedia. *NAT Port Mapping Protocol — Wikipedia, The Free Encyclopedia*. [Online; accessed 27-January-2014]. 2014. URL: http://en.wikipedia.org/w/index.php?title=NAT_Port_Mapping_Protocol&oldid=589867548 (ver p. 98).

[122] 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 (ver p. 65).

[123] 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 (ver p. 17).

[124] 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.

[125] 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.

[126] 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 (ver p. 5).

[127] Wikipedia. *Universal Plug and Play — Wikipedia, The Free Encyclopedia*. [Online; accessed 27-January-2014]. 2014. URL: http://en.wikipedia.org/w/index.php?title=Universal_Plug_and_Play&oldid=592063092 (ver p. 98).

[128] Wikipédia. *BitTorrent — Wikipédia, a enciclopédia livre*. [Online; accessed 22-outubro-2013]. 2013. URL: http://pt.wikipedia.org/w/index.php?title=BitTorrent&oldid=36953538 (ver p. 12).

[129] 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 (ver p. 46).

[130] 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 (ver p. 14).

[131] 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 (ver p. 50).

[132] Theory.org Wiki. *BitTorrentSpecification — Theory.org Wiki,* [Online; accessed 6-January-2014]. 2013. URL: https://wiki.theory.org/BitTorrentSpecification.

[133] 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 (ver p. 19).

[134] 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 (ver p. 83).

[135] D. Wing et al. *RFC6887: Port Control Protocol (PCP)*. Abr. de 2013. URL: http://tools.ietf.org/html/rfc6887 (ver p. 98).

[136] *World IPV6 Launch*. [Online; accessed 28-January-2014]. URL: http://www.worldipv6launch.org/ (ver p. 101).

[137] *XBT software — UDP Tracker protocol*. [Online; accessed 25-January-2014]. URL: http://xbtt.sourceforge.net/udp_tracker_protocol.html (ver p. 92).

[138] Tao Xie, Fanbao Liu e Dengguo Feng. *Fast Collision Attack on MD5*. [Online; accessed 17-January-2014]. 2013. URL: http://eprint.iacr.org/2013/170.pdf (ver p. 77).

Visão Pessoal

Aqui, apresento a minha visão sobre a experiência obtida neste trabalho, relacionando-a o curso do BCC.

Desafios e frustrações

Devo dizer que o tema BitTorrent não foi um assunto que eu desejei estudar a princípio. Na verdade, eu pretendia mesmo era aplicar algum estudo de redes sociais usando grafos, mas, quando eu contei meu plano para o professor Coelho, ele teve a (feliz e brilhante) idéia de me sugerir este assunto. Eis que abracei o tema e o escolhi como orientador.

Após ter definido o tema, o primeiro desafio (que acredito ser comum a todos que cursam esta disciplina) foi decidir o que escrever. Tive a mesma sensação que alguém deve ter quando lhe dão uma folha em branco e pedem para desenhar algo que não conhece. Afinal, como você vai fazer isso? Com que detalhes? Qual a abordagem? Foram muitas questões de uma só vez.

Uma idéia inicial foi procurar na Internet trabalhos acadêmicos sobre o BitTorrent. Foi aí que eu percebi que a área é abrangente, com artigos sobre muitos aspectos do protocolo. Então, tracei como objetivo escrever um trabalho de análise do BitTorrent e incluir um desses estudos já realizados. Durante o ano essa idéia caiu por terra, pois percebi que a análise requeria muito mais tempo do que eu pensava, juntamente com contratempos que foram ocorrendo, principalmente no segundo semestre.

Logo precisei decidir como escrever o trabalho. Uma idéia, que surgiu em uma das muitas reuniões que tive com o professor Coelho, foi o de desenvolver o tema apresentando código como se fizesse parte do texto. A sugestão soou estranha no começo mas, conforme o trabalho foi crescendo, se consolidou como boa idéia.

Como o objetivo era apresentar código, avaliei os de alguns programas cliente BitTorrent de código aberto. Não demorou muito para eu pensar no Transmission e adotá-lo, dado que é o programa oficial da distribuição de Linux Ubuntu, que atualmente é bastante usado.

Outro desafio que surgiu foi saber trabalhar com o LaTeX. Eu não sabia usá-lo muito, por isso tinha baixado um modelo preparado. Além disso, utilizava um editor online, hospedado em um site, que logo mostrou que não era tão bom, quando percebi que o pacote de formatação visual de código para LaTeX não funcionava. Tive que decidir entre manter o modelo de documento ou começar outro do arquivo em branco. Escolhi o segundo, e até hoje não me arrependo, pois consegui deixar mais organizado e ao me gosto.

Durante minhas pesquisas, percebi que encontrava material sobre o BitTorrent na Internet. Toda vez que buscava uma informação boa somente no Wikipedia, uma voz ecoava dizendo que Wikipedia não é referência. Com o tempo, essa voz foi morrendo, pois cada vez mais percebi que não era bem assim. Acredito que, com a rápida evolução tecnológica que temos atualmente, é impossível depender somente de fontes estáticas, como livros. Além disso, tive a oportunidade de tentar editar alguns artigos no Wikipedia, mais precisamente melhorando-os com referências, e até para isso foi difícil. Foi aí que percebi também que o antigo argumento de que "qualquer um pode escrever qualquer coisa no Wikipedia" não é válido, e comecei a conviver com o fato de que eu não teria opções melhores.

Um último desafio foi o de escrever o trabalho, dividindo o tempo entre ele e o estágio, em um período bastante curto. Escrevo esta seção no dia anterior ao da entrega, satisfeito com o resultado e com a sensação de dever cumprido.

Disciplinas relevantes ao trabalho

Eu avaliei as disciplinas do BCC no capítulo 5, na página 111.

Planos futuros

A cada passo que eu dava no desenvolvimento do tema, percebi o quão fascinante é o BitTorrent. Fiquei bastante contente com o fato dele se manter atual, mesmo com a certa "mesmice" que possui hoje em dia. Especificamente, a notícia de que é 7 vezes mais rápido do que o Dropbox [34] mostra como ele ainda é relevante. Além disso, notícias como a que uma juíza americana estudou o protocolo antes de tomar uma decisão em um julgamento envolvendo direitos autorais [36] me fazem ter esperança de que, um dia, o BitTorrent deixe de ser sinônimo de pirataria.

A fascinação pelo assunto me fez querer entrar na comunidade BitTorrent e colaborar de alguma forma, provavelmente no próprio Transmission, já que agora conheço boa parte do seu código. Após algum descanso nas próximas semanas, espero entrar em contato com os desenvolvedores e tentar ser voluntário no projeto.

Agradecimentos

Aqui vão alguns agradecimentos a pessoas que tornaram este trabalho possível.

Em especial, agradeço imensamente ao meu orientador, professor Coelho, pela paciência, atenção e dedicação, me atendendo sempre que precisei de reuniões para este trabalho, e até mesmo durante a graduação, quando achei que tudo estava distante. E obrigado por ter sugerido o tema. Não acho que podia ter sido melhor! =)

Agradeço ao meu pai, minha mãe e minha irmã por terem compreendido mais uma fase de ausência durante o curso, me incentivando em todas as decisões importantes que tive que tomar durante toda a minha passagem pela USP, que se iniciou em 2004.

Agradeço à Tatiana, minha namorada, e à sua mãe Valéria, por me acolherem e me acompanharem neste trabalho, sempre ao meu lado e de forma tão dedicada e paciente, me apoiando nos momentos mais difíceis.

Aos meus gestores de trabalho: Prof. Mardel de Conti (LabNumeral - Poli-USP); Jason Dyett (Harvard DRCLAS); Daniel Creão (UpLexis); Bruno Yoshimura e Allan Kajimoto (Kekanto), pelas horas de trabalho flexíveis, que me permitiram chegar neste ponto do curso do BCC, e em especial aos meus atuais colegas de LabArq da FAU-USP, Anderson Valtriani, Ricardo Couto e amigos desenvolvedores pelas ausências permitidas para que eu pudesse terminar este trabalho.

Aos meus amigos da turma de BCC 2009: Jackson, Renato, Samuel, Susana, Felipe, Rafael, Thiago, Diogo, Gustavo Katague, Fernando, Henrique, Wilson, Gustavo Coelho, Wallace, Jéssica, Jefferson, Tiago e Nádia (ex-IME), que nos momentos de desespero e de calmaria me acompanharam por muitos dias durante o curso e, com certeza, suas amizades vão me acompanhar pelo resto da vida.

Aos meus muitos amigos do BCC dos outros anos, em especial Luciano, Lucas, Marcos e Roberto, cuja amizade transcendeu o IME e hoje já fazem parte do meu cotidiano.

E aos **muitos** outros nomes que conheço e com quem tive momentos no IME ou no IF, nessa minha história pela USP, se estiver lendo este texto, saiba que também foi muito importante! =)