UNIVERSIDADE FEDERAL DO PARANÁ
CIÊNCIA DA COMPUTAÇÃO

BRUNO BERTONI

SISTEMA DE ARQUIVO BASEADO EM TORRENT COM CONTROLE DE COTA DE ARMAZENAMENTO

TRABALHO DE CONCLUSÃO DE CURSO

CURITIBA
2017

BRUNO BERTONI

SISTEMA DE ARQUIVO BASEADO EM TORRENT COM CONTROLE DE COTA DE ARMAZENAMENTO

Trabalho apresentado ao Bacharelado em Ciência da Computação da Universidade Federal do Paraná, como requisito parcial à obtenção do título de Bacharel em Ciência da Computação.
Orientador:
Prof. Dr. Luiz Carlos Erpen de Bona

CURITIBA
2017

Resumo

Este trabalho apresenta uma alteração do BTFS, um programa criador de sistemas de arquivos a partir de um arquivo .torrent. O BTFS mostrará toda a estrutura de arquivos dentro do arquivo .torrent como um sistema de arquivos inteiro. O BTFS utiliza o FUSE, ou Filesystem in Userspace, para criar um sistema de arquivos real e em nível de usuário. Para controlar o protocolo de torrent do programa, o BTFS utiliza a biblioteca Libtorrent, uma implementação completa do protocolo bittorrent focada em eficiência e escalabilidade. O BTFS fará o *download* dos arquivos assim que criar o sistema de arquivos. O BTFS também consegue atender a demanda do usuário, aumentando a prioridade do download de arquivos requisitados pelo usuário. A alteração feita nesse trabalho visa adicionar um controlador de cota de armazenamento do sistema de arquivos do BTFS, no intuito de limitar a utilização de espaço em disco. A alteração apagará o espaço ocupado pelo sistema de arquivos quando este chega em um certo limite estipulado pelo usuário, removendo o espaço ocupado pelos pedaços de torrent menos utilizados pelo usuário.

Lista de Figuras

Sumário

1 — Introdução

Atualmente o serviço de computação em nuvem tem crescido rapidamente por sua capacidade de oferecer capacidades computacionais flexíveis, baratas e oferecem amplo armazenamento de dados (Patil et al. (2017)). A computação em nuvem oferece uma maneira de reduzir os custos de compra, manutenção e armazenamento de equipamentos pois o usuário apenas contrata o serviço. Apesar de ser um bom serviço, segurança e privacidade são dois pontos cruciais a serem cuidados com a adoção da computação em nuvem. Casos recentes de vazamento de informações como o do Dropbox (Hunt (2017)) representam uma falha perigosa, onde existe a possibilidade de todos os dados de um usuário serem roubados ou deletados.

Uma opção de compartilhar capacidades computacionais são as redes *peer-to-peer*, ou P2P. Redes P2P de compartilhamento de recursos como o Gnutella (Ripeanu et al. (2002)) têm recebido muita atenção na última década por oferecerem uma grande disponibilidade para uso sem ter o custo significativo de ter uma entidade centralizada e muito redundante (Yin et al. (2016)). Diferente do cliente-servidor, que é o modelo de rede mais popular entre as aplicações mais famosas (Mitchell (2017)), todos os computadores dentro da rede P2P realizam as tarefas de cliente e servidor. Na rede P2P, os computadores compartilham seus recursos para todos os outros computadores, como memória, poder de processamento ou largura de banda. Apesar de precisar se preocupar com manutenção e equipamentos, o custo e desempenho de uma rede P2P consegue ser um meio termo entre a computação em nuvem e o modelo cliente-servidor (Infosec (2017), Patil et al. (2017)).

Um das opções para distribuir recursos, nesse caso arquivos, de maneira simples e usando redes P2P são sistemas de arquivos criados a partir de torrents, conhecidos como Bittorrent Filesystem ou BTFS (Gunnarsson (2017), Norberg (2017a)). Estes sistemas usam torrents, que usam o protocolo bittorrent para redes P2P (Cohen (2008)), para criar sistemas de arquivos e distribuir dados. Esses sistemas de arquivos são criados em espaço de usuário usando o FUSE (Oseberg (2017)), retirando a necessidade de ser administrador do sistema para criá-lo.

1.1 Motivação e Justificativa

Apesar do BTFS criar com sucesso um sistema de arquivos a partir de um torrent, o programa continua ocupando o espaço total do torrrent. Aproveitando o controle que o programa tem sobre os pedaços dos arquivos e as leituras dentro do sistema de arquivos, é possível manter somente os dados que são pertinentes ao usuário, economizando espaço em disco.

1.2 Objetivo Geral

Este projeto tem como objetivo implementar um sistema para controlar o armazenamento de dados em disco do BTFS, limitando o espaço ocupado pelo sistema de arquivos, porém mantendo sua capacidade de ler todos os arquivos.

1.3 Objetivos Específicos

Os objetivos específicos são:

- Implementar uma política de controle dos dados usados pelo usuário;
- Deletar dados dentro do sistema de arquivos sem interferir com sua funcionalidade;

1.4 Estrutura do Documento

O documento está dividido em seis capítulos. O capítulo presente mostra a motivação, justificativa e objetivos do desenvolvimento. Os três capítulos subsequentes explicam as teorias e algoritmos utilizados como base durante o desenvolvimento. O capítulo cinco é focado no desenvolvimento, escolhas e testes do projeto. O sexto e último capítulo faz considerações finais e conclui o projeto.

2 — Redes peer-to-peer(P2P)

Uma rede *peer-to-peer* é um sistema auto organizado com entidades autônomas e iguais que almejam o uso compartilhado de recursos distribuídos num ambiente de computadores em rede sem o uso de um serviço central (Oram (2001)).

Os computadores conectados nessa rede chamam-se *peers*. Diferente do modelo cliente-servidor (Sinha (1992)), cada *peer* controla suas conexões, roteando mensagens para outros *peers* e procurando por conteúdo (Vlachou et al. (2012)). Essa característica também aparece na distribuição de recursos, pois todo *peer* pode ceder seus recursos para serem usados por outros *peers*, seja memória, poder de processamento ou largura de banda.

Diferente de aplicações com servidores centralizados, onde a localização dos recursos é conhecida, sistemas descentralizados como a rede P2P guardam as informações em múltiplos e possivelmente distantes computadores dentro do sistema(Steinmetz and Wehrle (2005)). Os modelos mais comuns para organizar as redes P2P são a *estruturadas*, *não-estruturadas* e o de *super-peers*.

Redes P2P têm o problema de buscar e organizar os dados dentro da rede. Essa dificuldade é chamada de *Lookup Problem*. O *Lookup Problem* é como encontrar ou guardar um dado específico dentro de um conjunto de *peers*. Desde que a rede P2P foi desenvolvida, várias soluções foram criadas para cada tipo de topologia tentar resolver esse problema.

Peers em redes não-estruturadas precisam se conectar com outros *peers* de maneira aleatória e não há como saber onde estão os dados dentro da rede. *Peers* têm quase ou nenhum conhecimento sobre os recursos dos outros *peers*, então buscas dentro da rede podem chegar ao ponto de entrar em contato com todos os *peers* da rede (Vlachou et al. (2012)). Um método usado para a busca envolve um servidor central que contém a localização dos dados dentro da rede. Outro método, usado por redes como o *GNutella* (Ripeanu et al. (2002)), faz uma busca na rede inteira.

Usar um servidor central para controlar os dados tem uma excelente busca na ordem de O(1), porém ela não consegue manter esses resultados em redes grandes e ela tem o problema de ter um ponto de falha central, que é uma das características que a rede P2P

deveria evitar. Já o método de procurar pela rede inteira, chamado de *flooding*, consegue resolver os problemas da rede, é simples de implementar e manter, mas não é útil em redes grandes, fazendo com que a rede perca sua escalabilidade. O *flooding* tem um bom tempo de resposta mas o consumo de banda é grande e aumenta o número de mensagens enviadas para todos os *peers* (Vu et al. (2009)). Técnicas como a de *Routing Devices*(Crespo and Garcia-Molina (2002)) e *Semantic Overlay Networks*(Giunchiglia et al. (2011)) foram desenvolvidas para aliviar as falhas da busca por *flooding*(Vlachou et al. (2012)).

Uma rede *peer-to-peer* estruturada cria um método de busca dentro da rede que garanta localizar os recursos desejados se conectando com um números moderado de *peers*. Para fornecer rotas determinísticas, os *peers* são colocados num espaço de endereçamento virtual, a topologia é organizada com uma geometria específica e é criada uma função de convergência que une o identificador do *peer* e do dado para o algoritmo de roteamento (Shen et al. (2009)).

Como as redes *peer-to-peer* estruturadas têm uma topologia diretamente ligada com a localização dos recursos, ao invés de fazer uma busca aleatória, um *peer* pode rearranjar seus vizinhos e selecionar o melhor entre eles para solucionar a busca efetivamente(Dmitry Korzun (2013)). Isso tornará as próximas buscas similares entre esse grupo de *peers* mais rápida pois as mudanças na topologia serão preservadas. Dessa maneira, a rede pode manter a escalabilidade e a eficiência. O método mais comumente usado para mapear a rede é utilizar tabelas hash distribuídas.

Por fim, o sistema de *super-peers* (Yang and Garcia-Molina (2002a)) junta os méritos dos sistemas estruturados e não-estruturados, tentando resolver a limitação de escalabilidade e o problema de ter um ponto de vulnerabilidade, enquanto mantém todas as vantagens de um sistema completamente distribuído, onde cada *peer* faz sua própria tabela de indexação de seus arquivos (Shen et al. (2009)).

No sistema de *super-peers*, a distribuição de recursos não é igual entre os *peers*. A distribuição de recursos leva em consideração a capacidade de cada *peer*, como largura de banda e poder de processamento. Alguns *peers*, intitulados *super-peers*, recebem tarefas específicas como agregar *peers*, rotear buscas e mediar a rede inteira. Assim, só os *super-peers* formam a rede *peer-to-peer* e os outros *peers* se conectam diretamente a esse *backbone* de *super-peers* (Steinmetz and Wehrle (2005)). Utilizar essa estrutura que se aproveita da heterogeneidade dos *peers* para criar a rede pode torná-la eficiente (Yang and Garcia-

Molina (2002b)).

O restante desse capítulo discutirá técnicas de busca e diferentes implementações de redes P2P. A seção **2.1** apresenta a rede estruturada usando *DHT* e três tipos de *DHT*. A seção **2.2** mostrará a rede P2P chamada *bittorrent*, que é a rede usada nesse trabalho, e os algoritmos por trás dela.

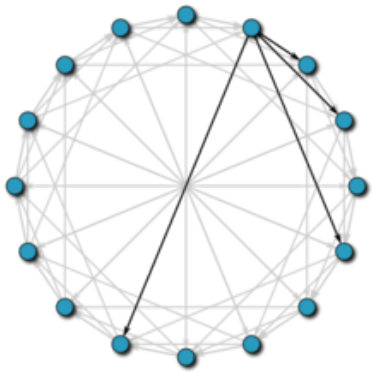
2.1 Tabelas Hash Distribuídas (DHT)

Uma DHT (*Distributed Hash Table*) foi uma solução desenvolvida para o problema de buscas em redes P2P estruturadas. Com uma DHT é possível ter uma visão global dos dados distribuídos dentro da rede, sem saber a localização atual dos dados. Diferente das não estruturadas, cada *peer* mantém uma lista de referências, ou vizinhos, para outros *peers*. Cada *peer* tem O(log N) referências, onde N é o número de *peers* da rede. Essa referência também guarda informação dos dados que determinado *peer* vizinho contém, deixando uma busca com até O(log N) saltos, que representa o número de conexões entre dois *peers* feitas durante a busca, até o *peer* que contém os dados desejados (Steinmetz and Wehrle (2005)). Como o número de referências de cada *peer* é igual, todo *peer* têm a mesmas funções dentro da rede, eliminando o problema de pontos de falhas.

Cada dado a ser inserido ou buscado dentro da DHT terá um identificador (ID) assinalado a ele, que será único dentro da rede. Esse ID normalmente é gerado a partir de uma hash table resistente a colisões criada usando as informações do dado (Steinmetz and Wehrle (2005)). Cada *peer* dentro da rede é responsável por um intervalo de IDs, que são endereços dentro da hash. As pesquisas e inserções dentro da rede podem ocorrer a partir de qualquer *peer*. Caso a operação recebida pelo *peer* conter um ID dentro do intervalo desse *peer*, a operação termina ali. Caso não seja um ID de seu intervalo, o *peer* irá pular para o *peer* vizinho que contém a chave mais próxima do ID da operação. Essa ação se repetirá até a operação chegar no *peer* responsável por esse ID.

Existem várias implementações que que utilizam uma DHT, como o Chord, CAN e Kademlia. A subseção *2.1.1* discute o Chord e a subseção *2.1.2* discute o CAN, duas implementações principais na área de redes P2P. Em seguida, na subseção *2.1.3* será discutido o Kademlia, que é a implementação usada no bittorrent, um dos focos desse trabalho.

2.1.1 Chord



Fonte: Wikipedia (Terashima (2017)).

Figura 2.1: Uma rede Chord com 16 *peers*, contendo em negrito a *finger table* de um dos *peers*.

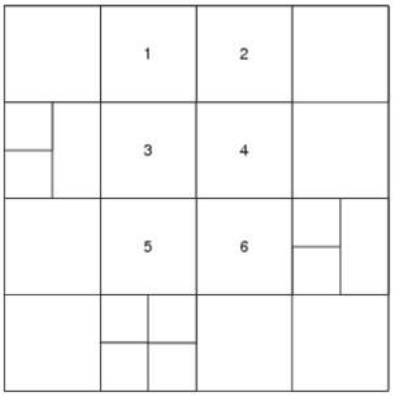
Chord (Stoica et al. (2001)) é um algoritmo e protocolo de redes P2P que organiza os *peers* usando uma DHT e deixando a topologia da rede de modo circular. Cada *peer* recebe um identificador único I baseado no seu IP, onde esse identificador é representado por b bits. Cada dado recebe uma chave única k e é guardado num *peer* que tenha um identificador $I \not\geq k$. Cada *peer* está organizado de maneira circular e de I crescente.

Para rotear as buscas e inserções cada *peer* n terá uma *finger table*, uma tabela que aponta para *peers* que estão, em sentido horário, na frente de n, estes chamados de *peers sucessores*. Uma *finger table* terá b entradas possíveis. Cada posição i da tabela aponta para o identificador do sucessor de até 2^{i-1} de distância, sendo a fórmula $n + 2^{i-1}$, onde $1 \leq i \leq b$. Com isso, temos uma média de $\frac{1}{2}log(N)$ passos, onde N é o número de *peers* na rede.

2.1.2 Content Addressable Network (CAN)

Content Addressable Network (Ratnasamy et al. (2001)), ou CAN, é uma estrutura de rede P2P que utiliza uma DHT, organizando os *peers* e dados dentro de um plano cartesiano de d dimensões. Os *peers* ficam espalhados dinamicamente dentro desse plano, onde cada *peer* é responsável por uma parte do plano. A busca dentro do CAN é bem simples, uma vez que cada *peer* só precisa conhecer seus vizinhos adjacentes.

No CAN, cada *peer* recebe um identificador x, onde esse identificador representa sua localização no plano. Quanto mais dimensões, mais elaborado o identificador será,



Fonte: Technische Universität Darmstadt (Darmstadt (2017)).

Figura 2.2: Uma rede CAN de duas dimensões.

como num plano onde *d=2* o *peer* terá um identificador *x,y*. O *peer* será responsável por todos os dados que tenham como uma chave da DHT uma posição dentro de sua área. Além disso, cada *peer* tem uma tabela com todos os seus vizinhos, onde um *peer n* é considerado vizinho de um *peer n2* se é adjacente em algumas das dimensões. Na figura **2.3**, podemos ver que o *peer* dentro do espaço 1 é vizinho do espaço 3.

Quando um novo *peer n* é colocado na rede, ele procurará sua posição. A busca pode começar de qualquer *peer*. Se o *peer n* não estiver em sua posição, *n* olhará a tabela de *peers* vizinhos e será enviado para o *peer* mais próximo de seu destino. Após achá-lo, ele verá se já existe um *peer* ocupando seu espaço. Caso exista, esse espaço será dividido em duas partes, cada uma ficando um *peer*. O tamanho do caminho percorrido dentro da rede será de $O(d(n^{\frac{1}{d}}))$, onde n é o número de *peers* e d o número de dimensões.

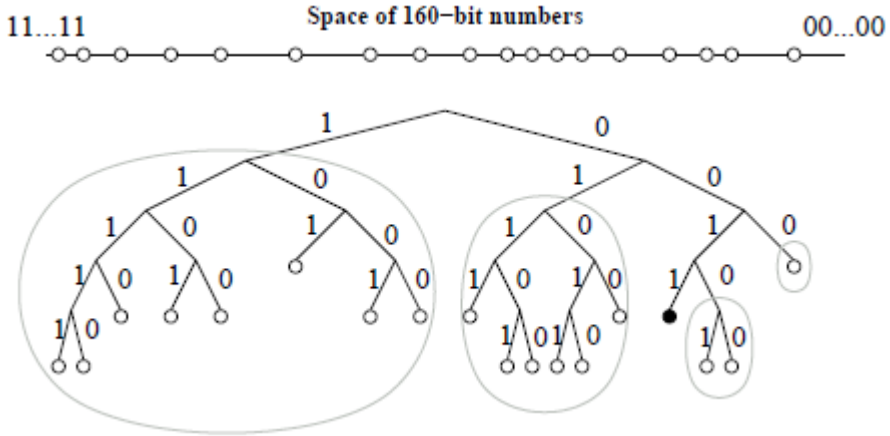
2.1.3 Kademlia

O Kademlia (Maymounkov and Mazières (2002)) usa uma estrutura similar às outras DHTs descritas. Cada *peer* recebe um identificador *ID*, esse tendo 160 bits, e ele é usado para as operações de busca e organização da rede. A topologia da rede é igual a uma árvore binária, onde cada *peer* é uma folha e sua posição é determinada pelo menor prefixo único do seu *ID*. Chaves de dados e *IDs* tem exatamente o mesmo tamanho e formato, tornando o cálculo da distância igual para ambos. Cada *peer* também contém uma tabela de roteamento.

Diferente das outras DHTs vistas e sendo seu ponto forte, a kademlia utiliza uma métrica XOR para a distância entre os *peers* dentro da rede. A principal vantagem de

usar a métrica XOR é ela ser simétrica, logo a distribuição de *peers* dentro das tabelas de roteamento é a mesma. Com isso, buscas iguais convergirão para o mesmo caminho (com o mesmo número de passos). A tabela também não precisará seguir regras fortes para manter a estrutura da rede, possibilitando aprender com as buscas realizadas. O *peer* poderá enviar várias buscas assíncronas e selecionar rotas baseadas por latência, agilizando as buscas.

A tabela de roteamento de cada *peer* são listas para cada um dos 160 bits de seu *ID*. Essas listas são chamadas de *k-buckets*, k sendo um parâmetro global de valor inteiro. Todo *k-bucket* é uma lista contém de 0 a k *peers*, onde k é escolhido dinamicamente. O valor de k é escolhido pensando que esse número de *peers* tem poucas chances de falhar dentro de uma hora.



Fonte: Maymounkov and Mazières (2002).

Figura 2.3: Árvore binária do kademlia. O ponto preto mostra o *peer* 0011. As subárvores circuladas indicam as possíveis origens dos *peers* que estão dentro de seus *k-buckets*.

O protocolo do kademlia contém 4 comandos:

- *PING* — Usado para verificar se o *peer* ainda está online.
- *STORE* — Guarda um valor dentro do *peer*.
- *FIND_NODE* — Comando que recebe um *ID* de 160 bits. O *peer* que receber esse comando retornará até k *peers* que estejam próximos do *ID* requisitado.
- *FIND_VALUE* — Funcionamento similar ao *FIND_NODE*, porém caso o *peer* contenha um *STORE* da chave procurada, ele retornará o dado guardado.

O processo por trás de achar um *peer* ou valor é a parte principal do desempenho do kademlia. O processo é um algoritmo recursivo que é chamado de *node lookup* e é executado assincronamente. O número de de execuções simultâneas é denotado por α e seu valor normalmente é 3. Seu processo começa quando um *peer* n envia um *FIND_NODE* para os α *peers* de seus *k-buckets* que estejam mais próximos do *ID* desejado. Caso um *peers* α não responda, ele será retirado temporariamente da lista. n então criará uma lista de resultados e a filtrará, deixando somente os k *peers* que estejam mais próximos do resultado. Com essa lista, n pegará os α *peers* mais próximos do *ID* e enviará um *FIND_NODE*. Esse processo se repetirá até ele não retornar nenhum *peer* que seja mais próximo que os *peers* já contidos na lista de resultados ou n ter enviado o comando de busca para k *peers* ativos.

Para um entrar na rede, um *peer* precisará ao menos saber o endereço IP e porta de um *peer* w que participe da rede. Caso ele nunca tenha participado da rede antes, ele receberá um *ID* aleatório ainda não assinalado na rede. O novo *peer* coloca w em seu *k-bucket* e o envia um *FIND_NODE*, populando os outros *peers* sobre sua chegada e seus próprios *k-buckets*.

2.2 Bittorrent

O bittorrent (Cohen (2008)) é um protococo semi-descentralizado para redes P2P que torna a distribuição de arquivos, principalmente grandes arquivos, mais simples e diminui o seu uso da largura de banda (Johnsen et al. (2005)).

No bittorrent, os arquivos são separados em pedaços que os *peers* irão trocar entre si. Cada *peer* é responsável por maximizar sua velocidade de *download* conversando com os *peers* corretos. Todo *peer* que faz parte do mesmo torrent formam um *exame*.

No protolo bittorrent os *peers* são divididos em *leeches* e *seeds*. Os *leeches* são *peers* que ainda não contém o arquivo completo enquanto os *seeds* têm os dados completos e estão na rede para enviar os dados para outros *peers*(Shah and Pâris (2007)). *Seeds* fazem apenas o envio, enquanto *leeches* recebem pedaços que ainda não têm e enviam os pedaços já recebidos.

Para gerenciar os *peers*, o protocolo utiliza um *tracker* que contém uma lista dos *peers* de um torrent. O controle da distribuição de dados da subrede é dividida entre o

tracker e os *peers*. A principal ação do *tracker* é fazer os *peers* se conectarem, não se envolvendo diretamente com a transferência pois ele mesmo não mantém nenhuma cópia dos dados (Johnsen et al. (2005)).

Quando um *peer* entra na rede, ele entrará em contato com o *tracker* e pedirá uma lista de IPs de *peers* já conectados à rede (Legout et al. (2006)). Essa lista se expandirá com o tempo quando outros *peers* se conectarem diretamente a ele. O *tracker* também enviará o IP do novo *peer* para outros *peers*, aumentando a possibilidade de conexões. Cada *peer* entra em contato com o *tracker* para atualizar seu status ou quando se desconecta da rede, dizendo o quanto ele já enviou e recebeu. Caso a lista de *peers* ativos fique pequena demais, o *peer* entrará em contato com o *tracker* pedindo uma nova lista de IPs.

A definição da escolha de quais pedaços dos arquivos são importantes afeta o desempenho da rede. Caso os *peers* baixem os pedaços de maneira aleatória, é possível que todos acabem com os mesmos pedaços disponíveis, deixando todos sem os pedaços que estão faltando. Existem vários algoritmos e políticas usados dentre da rede para estabelecer essa ordem, com *rarest first*, ou o mais raro primeiro, sendo o mais famoso dentro deles.

Outro ponto importante dentro da rede é garantir uma boa reciprocidade de *download* e *upload*. O principal ponto da rede é sua eficiência, logo todo *peer* tentará fazer o *download* dos dados da maneira mais rápida possível. Porém nem sempre todo *peer* terá uma conexão disponível para enviar os dados e é possível que um *peer* queira apenas receber os dados sem se importar com o balanço de dados da rede. *Peers* que nunca fazem *upload* são chamados de *free-riders*. O algoritmo usado para sanar o problema de reciprocidade é conhecido como *Choking algorithm*, que incentiva a cooperação dentro da rede. Com ele, *peers* com rápida taxa de *upload* têm maior probabilidade de ter uma alta velocidade de *download* (Pouwelse et al. (2005)).

Nas próximas duas subseções será explicado o processo de seleção de pedaços do arquivo (2.2.1) e o *Choking algorithm* (2.2.2).

2.2.1 Algoritmo de seleção de pedaços

A ordem de *download* dos pedaços dos dados implica diretamente no desempenho da rede. Para tentar manter uma taxa estável de *download*, os pedaços dos dados serão

divididos em sub-pedaços, normalmente de 16kb cada, para que cada *peer* esteja sempre fazendo *download* de dados. Para controlar a ordem com que os *peers* fazem o *download* dos pedaços, o bitorrent utiliza quatro políticas (Cohen (2003)).

A primeira política é a de uma vez que um *peer* faz a requisição de um sub-pedaço de um pedaço p, ele terá que fazer o *download* de todos os sub-pedaços de p para poder começar a fazer o *download* de outros sub-pedaços. Com isso, iremos conseguir um pedaço completo o mais rápido possível.

A segunda e mais importante política é a do *rarest first*. Ela dita que todo próximo pedaço que um *peer* deve fazer o *download* é o que está menos disponível na rede. Sua principal ação está no começo da rede, quando o *seed* ainda é o único com todos os pedaços. Os *leeches* que entrarem na rede recém estabelecida irão fazer o *download* de pedaços diferentes, espalhando ao máximo os dados do *seed*. O efeito disso é o aumento da taxa de *download*, pois quanto mais espalhado os pedaços estão, menos gargalos existirão dentro da rede pois o *upload* entre os *peers* estará distribuído. Outro efeito dessa política é prevenir a falta de pedaços. Caso o *seeder* saia da rede, é importante que todos os pedaços dos arquivos já estejam disponíveis na rede.

A terceira política dita que quando um novo *peer* entra na rede ele precisa fazer o *download* do primeiro pedaço disponível para ele. Isso fere a política de *rarest first*, mas é preciso aceitar essa política para que o *peer* comece a fazer o *upload* de dados o mais rápido possível. Após ele terminar o *download* do primeiro pedaço, o *peer* começará a usar a política de *rarest first*.

A última política é chamada de *endgame mode*. Ela tem como objetivo acelerar o *download* dos últimos pedaços de um dado. É possível que a taxa de transferência dos últimos pedaços do dado seja baixa, então o *peer* enviará várias requisições para o mesmo sub-pedaço, normalmente sendo 5 requisições por sub-pedaço. Essa requisição será transmitida para todos os *peers*. Quando o sub-pedaço for recebido, o *peer* que requisitou enviará uma mensagem cancelando o pedido para os outros *peers*.

2.2.2 Choking Algorithm

Os *peers* controlam a distribuição de dados dentro do *exame* usando o *choking algorithm*. O *choking algorithm* é baseado no algoritmo de *tit-for-tat*, ou olho por olho, que usa a ideia da cooperação mútua. Propositalmente os *peers* param de enviar dados

para outros *peers*, porém ainda podendo receber dados deles, colocando-os em estado de *choke* (Johnsen et al. (2005)). O princípio disso é fazer com que um *peer* só escolha enviar para *peers* que enviaram algo para ele recentemente, esses estando no estado de *unchoked*.

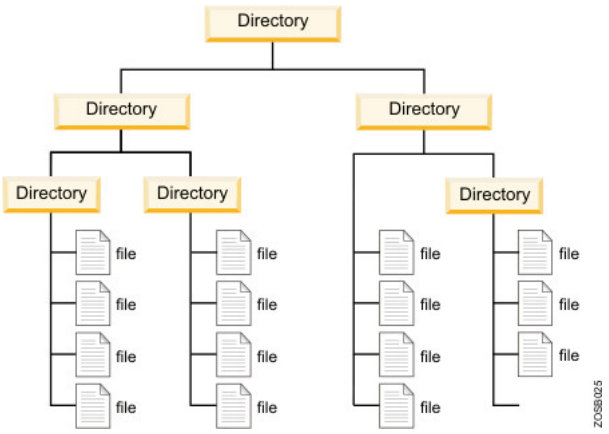
O *peer* irá passar para *unchoked* um número fixo de *peers*, sendo o padrão quatro de cada vez. Esses *peers* são os que recentemente mais enviaram dados a ele, criando uma conexão com os que parecem ser os melhores *peers* disponíveis.

Os *seeders* usam um algoritmo diferente pois, por não estarem mais recebendo dados, não conseguem calcular quais são os melhores *peers*. Os *seeders* então criam uma ordenação baseada na última vez que eles foram para o estado *unchoked*. A cada iteração o *seeder* desbloqueia 4 mais antigos *peers* dessa lista. Esse método é diferente do antigo, onde os *seeders* tinham uma versão similar ao *choking algorithm*. Esse novo método também provou ser mais eficiente em reduzir o número de *free riders* (Legout et al. (2006)).

3 — Sistemas de arquivos

Todo computador precisa armazenar e buscar informações.Enquanto um processo está rodando, ele terá um espaço limitado para ser utilizado. O problema disso é que esse espaço, além de limitado, não será mantido caso o processo seja finalizado. Para várias aplicações é preciso manter essas informações por um tempo indeterminado (Tanenbaum (2007)). Há também o problema de outros processos quererem acessar ou modificar a mesma informação, o que será inviável caso eles precisem acessar a área de memória de outro processo. Para isso, usamos unidades de armazenamento físicas, como discos rígidos, para armazenar os dados de maneira que eles se mantenham por um longo tempo e que possam ser acessados por diferentes processos ao mesmo tempo. Usamos um *sistema de arquivos* para organizar os dados dentro de unidades físicas.

Sistemas de aquivos são gerenciados pelo sistema operacional e possuem dois importantes componentes, *arquivos* e *pastas*. Arquivos são uma abstração de como os dados estão sendo guardados dentro dos discos rígidos. Arquivos podem ser acessados, alterados ou criados por processos, porém não serão apagados quando o processo for terminado. Um arquivo só pode ser apagado quando o dono do mesmo explicitar a ação (Tanenbaum (2007)).

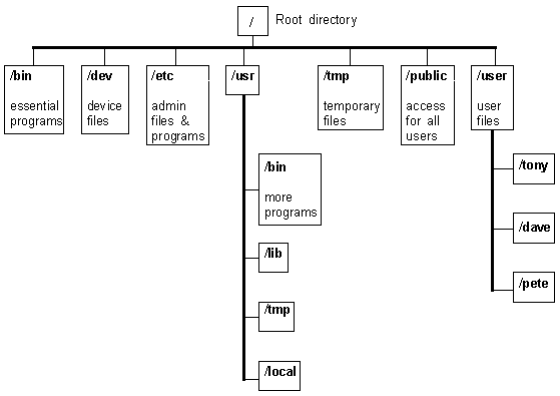


Fonte: IBM Knowledge center (IBM (2017)).

Figura 3.1: Estrutura hierárquica de pastas e arquivos

Para gerenciar os milhares de arquivos dentro de um disco rígido, o sistema de arquivo utiliza *diretórios* ou *pastas*, que em alguns sistemas também são considerados arquivos. Pastas são usadas para agrupar arquivos. A maneira mais utilizada para orga-

nizar as pastas é criando um sistema hierárquico. Com isso podemos organizar qualquer quantidade de arquivos de maneira simples e intuitiva. Como podemos ver na figura **3.1**, pastas também podem conter outras pastas, estas chamadas de *subpastas*. Além disso, em alguns sistemas de arquivos, as pastas guardam também informações, ou metadados, sobre suas subpastas e arquivos, como tipo dos arquivos, tamanho e nome. Isso facilita operações como uma busca dentro da árvore de pastas.



Fonte: Data Network Resource (Resource (2017)).

Figura 3.2: Árvore do sistema de arquivos Unix.

Diferentes sistemas de arquivos foram desenvolvidos, sendo o que usamos nesse trabalho um sistema similar ao UNIX (Ritchie and Thompson (1974)), chamado VFS. O VFS define um número básico de abstrações de sistema de arquivos e as operações que são permitidas nela (Tanenbaum (2007)). Sua principal diferença é que a árvore de arquivos inteira está embaixo na mesma raiz, representado na figura **3.2**. Sistemas de arquivos como o do Windows dividem suas partições de disco e mídias removíveis em diferentes árvores. Já no UNIX, eles serão montados como diretórios normais. Outra diferença é que diretórios são tratados como arquivos dentro do VFS.

No VFS existem 4 estrutura principais de sistema de arquivos (Tanenbaum (2007)):

- *Superblock*: O superbloco contém informação crítica sobre a organização de um sistema de arquivos.
- *i-node*: Um i-node, ou index-node, contém a descrição de um arquivo, como quem é o dono, as permissões e localização de seu conteúdo.
- *Dentry*: Para facilitar algumas operações de diretórios, o VFS cria *dentries*, que representa uma entrada de diretório. O dentry é um componente específico do ca-

minho do diretório. Por exemplo, no caminho */bin/vi*, *bin* e *vi* são estruturas dentry. Eles são criados na hora que for necessário realizar operações sobre o diretório.

- *File*: Uma estrutura *file* é uma representação em memória virtual de um arquivo aberto. Ela será criada quando um programa abrir algum arquivo.

Para acessar arquivos desses diretórios criados a partir de qualquer dispositivo, sejam eles mídias removíveis ou não, o sistema operacional precisará saber qual o local dentro da hierarquia virtual de diretórios em que os arquivos precisam ser colocados. Esse processo, chamado de *montagem* de um arquivo de sistemas, resultará na produção de um superbloco. O sistema operacional então montará o dispositivo em um diretório dentro do VFS. Esse diretório, denominado *ponto de montagem*, especifica qual o caminho desse diretório a partir da raiz do sistema de arquivos (Brouwer (2017)).

As interações entre uma aplicação e o sistema de arquivos são feita através de *chamadas de sistema*. O sistema operacional fará a ponte entre a aplicação e o sistema de arquivos. Existem cinco categorias principais de chamadas de sistema: controle de processos, manipulação de dispositivos, manutenção da informação, comunicações e manipulação de arquivos (Silberschatz et al. (2008)).

Na próxima seção explicaremos o *filesystem in userspace*, que é o sistema de arquivos usado nesse trabalho.

3.1 Filesystem in Userspace (FUSE)

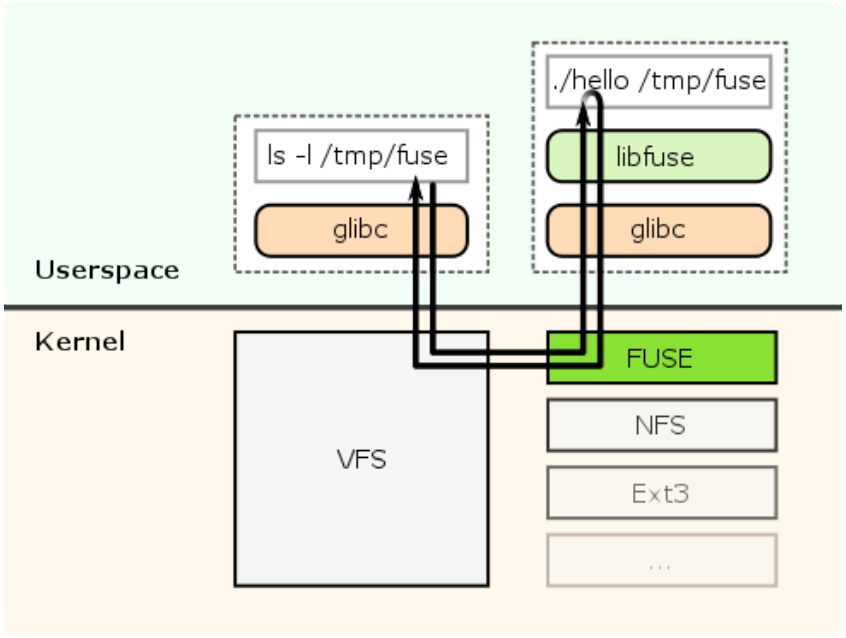
O *Filesystem in Userspace* (FUSE) é um projeto desenvolvido para sistema UNIX-like que serve como uma interface para programas dentro do espaço do usuário conseguirem exportar um sistema de arquivos para o kernel do Linux (Libfuse (2017)). Por estar dentro do espaço do usuário, é possível implementar um sistema de arquivos sem alterar o kernel do VFS e ser o administrador do sistema operacional.

O FUSE é composto de dois componentes, o módulo dentro do kernel e a biblioteca *libfuse*, uma API, que fica no espaço do usuário. Os dois componentes conversam a partir de um descritor de arquivo especial fornecido pelo nó de dispositivo, que serve como identificador do fuse dentro do sistema operacional, encontrado em */dev/fuse*. Quando o comando de montagem for enviado ao kernel o descritor de arquivos será passado junto a ele, criando o sistema de arquivos.

O FUSE fornece dois tipos de APIs para escrever um sistema de arquivos. Uma delas é uma API síncrona de alto nível de programação e a outra uma API assíncrona de baixo nível de programação (Libfuse (2017)).

A API síncrona é baseada em nome de caminhos, se aproximando muito das *chamadas de sistemas*. O inode do arquivo que está sendo afetado pela operação é identificado pelo seu caminho. Essa interface simples é suficiente para quase todos os sistemas de arquivos (Kantee and Crooks (2007)).

A API assíncrona tem operações completamente diferentes para o sistema de arquivos, lembrando muito as do VFS. Adicionalmente, ela exige que o sistema de arquivos cuide manualmente de todas as operações entre o sistema de arquivos e o kernel (Kantee and Crooks (2007)).



Fonte: Data Network Resource (Sven (2017)).

Figura 3.3: Fluxograma mostrando como o FUSE funciona.

A figura *3.3* demonstra como o comando *ls -l /tmp/fuse* é executado. O comando *ls* será transformado numa chamada de sistema e será enviada para o kernel, onde o VFS o receberá. O VFS percebe que esse comando está sendo usado em um sistema de arquivos em espaço de usuário e envia o comando para o módulo do FUSE. O módulo do kernel passará o comando através das duas bibliotecas de funções, uma do Linux e outra do FUSE, e entrará em contato com o arquivo alvo do comando. Assim que a função for executada no arquivo alvo o resultado do comando será retornado pelo mesmo caminho

até o programa que o requisitou.

Usando a API do FUSE, você pode escrever praticamente qualquer tipo de sistema de arquivos que você quiser, colocando quaisquer funcionalidades que quiser (Layton (2017)).

4 — Bittorrent File System(*BTFS*)

O Bittorrent File System (Gunnarsson (2017)), ou BTFS, foi desenvolvido por Johan Gunnarsson para a criação de um sistema de arquivos usando o FUSE a partir de um arquivo .torrent ou *magnet link*, que é uma referência para uma rede bittorent. O sistema de arquivos criado terá exatamente a mesma estrutura de arquivos contida no .torrent e só leituras poderão ser executadas.

As vantagens do BTFS são a capacidade de usar os comandos UNIX para explorar o conteúdo do torrent, seu acesso ao conteúdo ser transparente para outros programas. Os programas acessarão os arquivos sem perceberem que os arquivos ainda não estão fisicamente em disco e que ainda estão sendo recebidos do torrent.

O BTFS também conta com um *download* sob demanda, alterando a prioridade do *download* dos pedaços dos arquivos. Quando uma aplicação tentar ler algum arquivo do sistema de arquivos, o BTFS fará o *download* dos pedaços do arquivo sendo lido, aumentando a prioridade de *download* dos pedaços que estão sendo lidos.

Por padrão os arquivos que tiveram seu *download* concluído ficarão na máquina até o sistema de arquivos ser desmontado, mas é possível também mantê-los em disco. Há também opções como controle de velocidade de *download* e *upload*, quais portas de entrada e saída usar no torrent e apenas fazer o *download* dos metadados do torrent, que pode ser usado para explorar todo o sistema de arquivos sem fazer o *download* dos dados.

Quando for executado, o BTFS montará um sistema de arquivos FUSE no diretório indicado e irá colocar a estrutura de arquivos do *arquivo.torrent* dentro dessa pasta. Há outros parâmetros que podem ser utilizados para usar outras funcões do BTFS. O BTFS aceita qualquer parâmetro opcional do FUSE. Além deles, existem parâmetros opcionais para controlar alguns aspectos do sistema de arquivos e do funcionamento do BitTorrent, como alterar a taxa de *download* e *upload*.

O desafio de se criar um sistema de arquivos usando um torrent como base é a diferença de como os arquivos são tratados em um sistema de arquivos e no torrent. Um sistema de arquivos funciona sob demanda do usuário, acessando os dados da maneira que ele quiser. Já o torrent faz o *download* dos arquivos de maneira randomizada, usando suas próprias regras para a seleção de pedaços que serão feito o *download*.

Será explicado à fundo nas próximas seções a arquitetura, comandos e funcionamento do BTFS, assim como o BTFS lida com o desafio do *download* aleatório do torrent, tentando mantê-lo da maneira mais próxima de ser sob demanda.

4.1 Arquitetura e Funcionamento

A arquitetura do BTFS tem dois grandes componentes, o controle do sistema de arquivos usando FUSE e o controle do funcionamento do torrent. Para isso, é usado a biblioteca interna do FUSE para o Linux e a biblioteca libtorrent (Norberg (2017b)), uma biblioteca de código aberto feita em C++ que implementa o protocolo BitTorrent.

As operações do FUSE, estas descritas por Oseberg (Oseberg (2017)), implementadas no BTFS para seu funcionamento são: *getattr, readdir, open, read, statfs, listxattr, getxatter, init* e *destroy*. Com exceção das funções *init* e *destroy*, essas funções são operações básicas de um sistema de arquivo do Linux, usadas quando vemos informações, abrimos ou lemos um arquivo. As funções *init* e *destroy* são específicas do FUSE, sendo chamadas durante a montagem e desmontagem do sistema de arquivos, respectivamente.

O uso básico do libtorrent começa com o programa construindo uma *sessão*. A sessão guarda as configurações e administra os torrents inseridos nela, além de cuidar da iteração da rede (Norberg (2017c)). Após iniciar uma sessão, é adicionado um arquivo .torrent na sessão, que retornará um *torrent_handle*. O *torrent_handle* é usado para executar as operações referentes ao torrent, como pedir pedaços, checar o status deles e alterar suas prioridades de *download*. Para informações dos arquivos e mapeamento dos pedaços usamos a estrutura *torrent_info*, que pode ser acessada a partir do *torrent_handle*.

O BTFS começa inicializando as operações do FUSE e tratando de alguns parâmetros opcionais. Após isso, o BTFS montará o sistema de arquivos. Durante a inicialização da montagem do sistema de arquivos o FUSE chamará a função *init*. A função criará uma sessão do libtorrent e fará as configurações básicas do BitTorrent, gerando o *torrent_handle* para o torrent enviado na linha de comando.

Com o sistema montado e o libtorrent inicializado, o BTFS fará o *download* dos metadados do torrent e criará dentro do sistema de arquivos montado os arquivos usando os metadados obtidos. Caso a opção de *browse-only* tenha sido usado, o BTFS irá parar o *download* do torrent. A opção *browse-only* faz o *download* apenas dos metadados do

arquivo .torrent. Isso criará o sistema de arquivos e toda a estrutura de arquivos, porém os arquivos não terão dados dentro deles. Caso a opção de *browse-only* não tenha sido usada, o BTFS começará a requisitar pedaços dos arquivos do torrent.

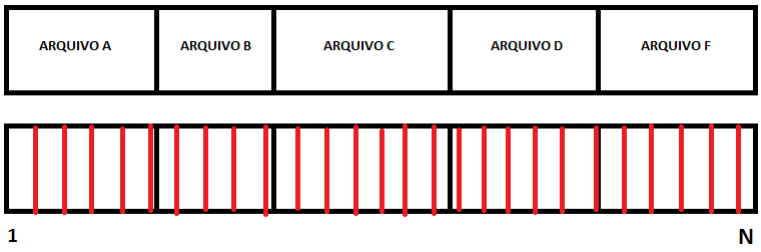


Figura 4.1: Organização dos arquivos dentro do libtorrent. Todos os arquivos de um arquivo .torrent estão numa fila. Cada arquivo está dividido em vários pedaços de torrent, demonstrados pela separação em vermelho.

Usando o *torrent_info* e *torrent_handle*, o BTFS saberá quais pedaços estão mapeados para cada arquivo, o número de pedaços P que o torrent inteiro contém e quais ainda não foram recebidos. Os arquivos estão organizados no libtorrent como um vetor de pedaços, indicados em vermelho na figura **4.1**, cada pedaço tendo um identificador. O identificador de cada pedaço é um número inteiro, indo de 1 a N, onde N é o número total de pedaços do torrent. Cada pedaço recebe o seu identificador utilizando a ordem de seus arquivos dentro do *torrent_info*.

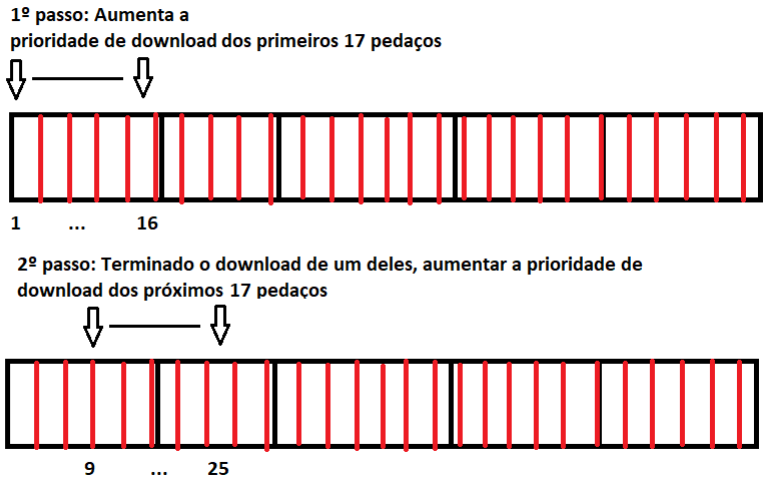


Figura 4.2: Exemplo do início do *download* no BTFS. O primeiro passo é sempre o mesmo para qualquer torrent.

Ao invés de fazer o *download* da maneira comum do torrent, o BTFS tenta controlar a ordem de download dos pedaços. Quando o torrent é iniciado, o BTFS aumentará

a prioridade do *download* dos 17 primeiros pedaços do torrent. Isso fará com que o *download* dos arquivos do torrent fique focado dentro desse intervalo de pedaços com prioridade maior, não deixando o *download* aleatório. Assim que o *download* de um pedaço P for completado, o BTFS aumentará a prioridade dos próximos 17 pedaços depois de P. No exemplo na figura **4.2**, o *download* do pedaço com identificador 8 foi terminado, logo o BTFS aumentará a prioridade dos pedaços 9 a 25. Esse processo se repetirá até o *download* de todos os pedaços for completado.

Para suprir as leituras sob demanda do usuário, o BTFS usa uma estratégia similar à do *download* de pedaços. Quando uma leitura de um arquivo é requisitada, o BTFS verá qual pedaço do arquivo está tentando ser lido. O BTFS então deixará a leitura em espera e checará se o pedaço sendo lido já teve seu *download* concluído. Caso o *download* do pedaço ainda não tenha sido concluído, o BTFS aumentará a prioridade do pedaço e dos próximos 17 pedaços adjacentes. Quando o *download* for completado, a leitura sairá da espera e lerá o pedaço. Esse processo se repete para toda leitura feita dentro do sistema de arquivos.

Quando o sistema de arquivos estiver sendo desmontado, o FUSE chamará a função *btfs_destroy*. A função parará o torrent, deletará a sessão do libtorrent e, caso o parâmetro *keep* não tenha sido utilizado, deletará os arquivos.

5 — Sistema de Arquivo Baseado em Torrent com Controle de Cota de Armazenamento

O BTFS é uma implementação de um sistema de arquivos remoto usando o protocolo bitorrent, mas não é uma ideia totalmente original. Arvid Norberg, criador da biblioteca libtorrent, já havia descrito e desenvolvido um sistema de arquivos remoto usando o bitorrent (Norberg (2017a)). Em seu desenvolvimento, Arvid focou na alocação de blocos e escrita dentro do sistema de arquivos, tentando adquirir um maior desempenho no *download* do torrent.

Considerando a possibilidade de ler um arquivo a partir de qualquer ponto e poder controlar as leituras dentro do sistema de arquivos, o usuário, em tese, não precisa ocupar todo o espaço que o sistema de arquivos e arquivos gerados a partir do arquivo .torrent ocupa. Com um controle do armazenamento do sistema de arquivos do BFTS, o usuário poderia ter fisicamente em seu sistema somente o que ele está usando no momento. Para realizar isso, foi desenvolvido uma alteração que permite o BTFS controlar o espaço ocupado dentro do sistema de arquivos. O objetivo dessa alteração é limitar o espaço físico ocupado pelo sistema de arquivos do BTFS, apagando pedaços de arquivos que estão com o *download* completo e não estão sendo utilizados pelo usuário.

O BTFS foi a implementação do FUSE escolhida por ser a mais simples. O libfuse (GitHub (2017)), outra implementação de um FUSE, tem uma complexidade maior, sendo o seu uso desnecessário para o trabalho desenvolvido. Para a biblioteca do torrent não foi encontrado uma que funcionasse bem com o BTFS como a libtorrent, uma vez que ela já foi feita para a mesma linguagem em que o BTFS foi desenvolvida e tem todos os elementos necessários para este trabalho.

Para desenvolver essa alteração foi preciso focar em dois pontos: qual método usar para escolher quais pedaços serem apagados e como apagar esses pedaços de maneira que o sistema de arquivos ainda possa utilizar os arquivos no futuro.

Para selecionar a ordem com que os pedaços serão apagados, foi usado a política do menos recentemente usado, ou LRU. O LRU, do inglês *least recently used* e com seu funcionamento exemplificado por Barnwal (Barnwal (2017)), parte do princípio que

os dados, no caso do BTFS os pedaços dos arquivos, utilizados com mais frequência nas últimas execuções provavelmente serão utilizados novamente num futuro próximo. O LRU cria uma lista dos arquivos, ordenando-os do pedaço menos usado recentemente para o pedaço mais usado recentemente. Usando essa lista, podemos apagar os pedaços que não foram utilizados recentemente, mantendo em disco o que provavelmente é relevante ao usuário. Outra opção considerada para apagar os pedaços foi o uso do *timestamp* dos arquivos, que mostra a data da última vez que o arquivo foi usado. Porém Johan Gunnarsson programou o BTFS para deixar o *timestamp* fixo em todos os arquivos, impossibilitando saber quando ou quais arquivos foram utilizados.

O BTFS cria durante sua inicialização um vetor de identificadores dos arquivos do torrent para servir de cache. O tamanho do vetor leva em consideração o tamanho da cache e o tamanho de cada pedaço do torrent. Como nem sempre é possível ocupar perfeitamente o espaço da cache pois pode faltar espaço, foi decidido arredondar o espaço da cache para baixo para não ultrapassar o limite imposto pelo usuário. Por exemplo, se a cache tem o tamanho máximo de 10MB e cada pedaço tem 3MB de tamanho, o BTFS ocupará no máximo 9MB da cache, ou seja, 3 pedaços.

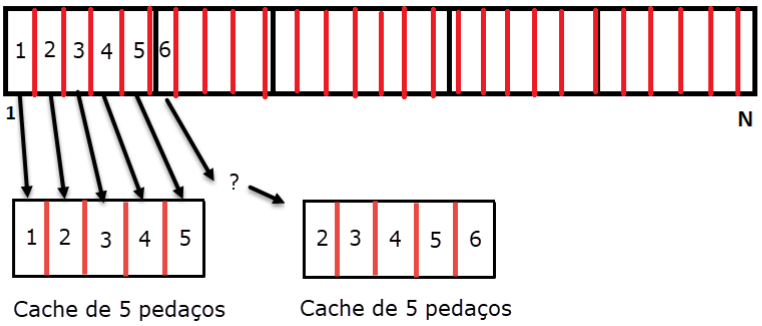


Figura 5.1: Exemplo de como a cota de armazenamento funciona. O pedaço 1, que é o pedaço menos usado recentemente, é removido para o pedaço 6 ser inserido.

Toda vez que um pedaço novo for requisitado, o programa irá checar se o pedaço já está dentro da cache. Caso não esteja, ele será colocado no vetor como o arquivo mais recentemente usado. Caso já esteja, sua posição será atualizada, se tornando o mais recentemente usado. Caso um novo pedaço seja requisitado e a cache esteja cheia, o programa irá percorrer o cache para encontrar o menos recentemente usado e irá apagá-lo.

O libtorrent, quando inicializado, não permite que hajam escritas nos arquivos para que não haja concorrência na hora da escrita. Para permitir a escrita, foi preciso

apagar a sessão do libtorrent. Assim que a sessão é deletada, é usado o comando *fallocate* (Kerrisk (2017)) do Linux como a opção de *FALLOC_FL_PUNCH_HOLE* para deletar o arquivo menos recentemente usado. O comando *fallocate*, juntamente com a localização do pedaço dentro do arquivo e a opção *FALLOC_FL_PUNCH_HOLE*, irá apagar os blocos ocupados pelo pedaço do torrent, desalocando o espaço ocupado em disco. Na figura **5.1**, a cache estava ocupada pelos pedaços 1 a 5. Antes de começar o download do pedaço 6, o pedaço 1, que é o atual pedaço menos usado recentemente, é desalocado para dar espaço ao pedaço 6.

Assim que a operação para desalocar os pedaços for completada, o sistema irá reiniciar a sessão do libtorrent. O sistema então usará a função *force_recheck()* do *torrent_handle*, que checará quais pedaços estão com seu *download* completo. Terminando sua execução, o sistema irá voltar a requisitar novos pedaços. Caso o processo de apagar um pedaço tenha acontecido quando uma leitura requisitou um pedaço, o pedaço requisitado receberá uma prioridade maior em seu *download* assim que a sessão foi reiniciada.

5.1 Resultados Experimentais

Para testar a funcionalidade do controle de armazenamento para o BTFS, foram usados torrents de tamanhos e número de arquivos variados. Os testes focaram em ver se a implementação cumpre os objetivos do trabalho e se ela altera o desempenho do BTFS. Os testes foram feitos usando quatro torrents diferentes, com tamanhos de 8,71MB, 148MB, 545MB e 102GB, e em um HD SATA-II com velocidade de 7200RPM e tamanho de 1TB. Os quatro torrents também tem uma alta taxa de velocidade de *download*.

O primeiro teste foi feito deixando a cache com tamanho igual ao do torrent para checar se as modificações não alteraram o funcionamento normal do código. O sistema de arquivos funcionou de maneira igual à do BTFS original, pois a parte de remover pedaços da cache não é executada. O vetor ainda é criado e preenchido completamente, mas nenhuma leitura acontece nele.

O segundo teste foi feito com o tamanho da cache sendo igual ao tamanho do maior arquivo dentro do torrent, com o tamanho da cache sendo 2,03MB, 13,75MB, 320MB e 1,05GB. A alteração funcionou como desejado, mantendo o espaço dentro do sistema de arquivos limitado. O desempenho sofreu um grande impacto nos testes com

cache grande pois a função *force_recheck()* precisava checar uma grande quantia de pedaços antes de voltar a fazer o *download*.

No terceiro teste o tamanho da cache era igual ao menor arquivo no torrent, com o tamanho da cache para os quatro torrents sendo 1,87MB, 5,34MB, 95MB e 1,02GB. Novamente as alterações deram certo, porém no último caso, onde a cache teve o tamanho de 1,02GB, o desempenho pelo tempo de execução do *force_recheck()* foi baixo. Aqui já podemos perceber que em casos onde a cache é muito grande o desempenho sempre é bem pior.

No quarto teste foi averiguado o que acontece quando há múltiplas leituras acontecendo ao mesmo tempo. Nesse teste ambas leituras tinha a mesma velocidade constante lendo arquivos de música diferentes. Esse teste foi feito com uma cache grande e uma pequena para o maior arquivo torrent, de 102GB. A cache pequena foi de 30MB e a grande de 1GB. O sistema de arquivos teve um bom desempenho usando a cache pequena, mantendo uma boa velocidade de leitura para as duas leituras simultâneas pois o tempo de execução do *force_recheck()*, que pode ser visto na tabela 5.1, é pequeno.

O quarto teste com a cache grande teve um péssimo desempenho devido ao dobro de requisições de novos pedaços. Por ter um número de requisição de novos pedaços elevados, o número de vezes que o *force_recheck()* é executado também dobra. Para esse teste, foi feito uma alteração que apaga metade da cache ao invés de só um único pedaço. Com essa alteração, o desempenho melhorou consideravelmente, diminuindo o número de vezes que o *force_recheck()* é executado e diminuindo o seu tempo de execução pois há menos pedaços a serem checados. O problema dessa alteração é diminuir pela metade o espaço ocupado pela cache toda vez que acontece uma deleção.

O último teste feito focou em leituras aleatórias ao invés de sequenciais dentro do torrent. Esse teste foi feito com uma cache grande e uma pequena para o maior arquivo torrent, de 102GB. Para a cache pequena de 30MB o desempenho foi mediano, pois o algoritmo para adiantar os pedaços adjacentes ao da leitura não ajuda em leituras aleatórias. O mesmo caso acontece quando caches grandes, mas o desempenho é pior por causa do *force_recheck()*.

As alterações no BTFS conseguiram alcançar o objetivo de manter o espaço ocupado do sistema de arquivos dentro do limite que foi estipulado durante sua criação. As alterações também não afetaram nenhum das funcionalidades originais do BTFS.

Em quase todos os testes feitos, a modificação diminuiu a velocidade de leitura toda vez que o programa inicia a desalocação de pedaços. O processo que causou a maior queda de desempenho é a função *force_recheck()*. O tempo de execução da função *force_recheck()* é proporcional ao espaço ocupado pelo sistema de arquivos. Caso o tamanho do torrent e o limitador de espaço do sistema de arquivos sejam grandes, o programa precisará percorrer vários pedaços para verificar o sistema de arquivos inteiro. Na tabela 5.1 podemos ver que o tempo de execução do *force_recheck()* já chega aos 16,42 segundos caso o sistema de arquivos esteja ocupando 1,05GB.

Espaço ocupado no sistema de arquivos	**Tempo de execução do *force_recheck***
33,54MB	3,07 segundos
148MB	3,50 segundos
545MB	8,40 segundos
1,05GB	16,42 segundos
4,54GB	67,48 segundos
36,1GB	364,03 segundos

Tabela 5.1: Tempo em segundos da verificação do *force_recheck* quando o espaço ocupado pelo sistema de arquivos. Testes executados em um HD SATA-II, com velocidade de 7200RPM e tamanho de 1TB.

Os dois casos onde o sistema teve o melhor desempenho foram quando o limitador de espaço é pequeno ou quando desalocamos metade da cache quando o seu tamanho é grande. No caso do limitador ser pequeno, o sistema de arquivos não estará com muitos pedaços para serem verificados, agilizando o processo. É possível também que quando o programa apaga os pedaços ele acabe removendo quase todo o espaço ocupado pelo sistema de arquivos, pois o programa desaloca todos os pedaços de um arquivo.

6 — Conclusão

Este documento apresentou uma modificação para o BTFS, incluindo um controle de cota de armazenamento para o sistema de arquivos com o objetivo de apagar pedaços do torrent que não estejam sendo mais utilizados. O BTFS utiliza o FUSE para montar um sistema de arquivos em espaço de usuário a partir de um torrent e a biblioteca libtorrent para controlar as funcionalidades do torrent. O sistema também tem a capacidade de dar prioridade a pedaços dos arquivos que estão sendo lidos no momento, fazendo seu *download* sob demanda.

A alteração feita no BTFS tem como objetivo controlar o espaço ocupado pelo sistema de arquivos, economizando espaço em disco. Para controlar o espaço, o usuário especifica um limite de espaço que o sistema de arquivos pode ocupar. O programa então irá criar uma lista dos pedaços que tiveram o *download* completo e os ordenará usando a política do LRU. Quando o sistema de arquivos alcançar o limite estipulado pelo usuário, o programa apagará o pedaço menos usado recentemente, desalocando-o do sistema de arquivos. Assim, o usuário poderá acessar todos os arquivos do torrent sem ocupar o espaço total de todos os arquivos.

Os testes feitos com a implementação mostraram que ela conseguiu manter o espaço limitado dentro do sistema de arquivos sem afetar as outras funcionalidades do sistema, porém afetando seu desempenho em demasia. Precisar deletar a sessão do torrent para conseguir apagar os pedaços dos arquivos e depois reavaliar os pedaços que ainda estão no sistema de arquivos pode se torna um processo muito custoso quando a cacheé grande.

Um dos trabalhos a ser realizado para melhorar o programa é alterar a biblioteca do libtorrent para possibilitar a escrita sem precisar apagar a sessão. Com essa alteração, o desempenho do sistema seria substancialmente melhor por não precisar reavaliar o sistema de arquivos inteiro. Quando os pedaços de um arquivo fossem apagados, poderíamos apenas sinalizar ao controlador do torrent que aqueles pedaços foram desalocados, mantendo o sistema atualizado sem precisar forçar a reavaliação.

Referências Bibliográficas

Kalyanee Patil, S. D. Khatawkar, and Amol Dange. Secure data sharing in cloud through limiting trust in third party/server. *International Research Journal of Engineering and Technology*, 2017.

Troy Hunt. The dropbox hack is real. 2017. URL `https://www.troyhunt.com/the-dropbox-hack-is-real/`.

Matei Ripeanu, Adriana Iamnitchi, and Ian Foster. Mapping the gnutella network. *IEEE Internet Computing*, 6(1):50–57, January 2002. ISSN 1089-7801. doi: 10.1109/4236.978369. URL `http://dx.doi.org/10.1109/4236.978369`.

Baoqun Yin, Xiaonong Lu, Jing Huang, and Yu Kang. Analysis of topology dynamics for unstructured p2p networks. *Computer Communications*, 80(Supplement C):72 – 81, 2016. ISSN 0140-3664. doi: https://doi.org/10.1016/j.comcom.2016.01.009. URL `http://www.sciencedirect.com/science/article/pii/S0140366416300020`.

Bradley Mitchell. Introduction to client server networks. 2017. URL `https://www.lifewire.com/introduction-to-client-server-networks-817420`.

Infosec. Peer to peer network. 2017. URL `http://infosecawareness.in/peer-to-peer-network`.

Johan Gunnarsson. Bittorrent file system. 2017. URL `https://github.com/johang/btfs`.

Arvid Norberg. A bittorrent filesystem. 2017a. URL `http://blog.libtorrent.org/2014/12/a-bittorrent-filesystem/`.

Bram Cohen. The bittorrent protocol specification. January 2008. URL `http://www.bittorrent.org/beps/bep_0003.html`.

Terje Oseberg. Estrutura de referência do fuse_operations. 2017. URL `https://lastlog.de/misc/fuse-doc/doc/html/structfuse__operations.html`.

Andy Oram, editor. *Peer-to-Peer: Harnessing the Power of Disruptive Technologies*. O'Reilly & Associates, Inc., Sebastopol, CA, USA, 2001. ISBN 059600110X.

Alok Sinha. Client-server computing. *Commun. ACM*, 35(7):77–98, July 1992. ISSN 0001-0782. doi: 10.1145/129902.129908. URL `http://doi.acm.org/10.1145/129902.129908`.

Akrivi Vlachou, Christos Doulkeridis, Kjetil Nrvg, and Yannis Kotidis. *Peer-to-Peer Query Processing over Multidimensional Data*. Springer Publishing Company, Incorporated, 2012. ISBN 1461421098, 9781461421092.

Ralf Steinmetz and Klaus (Eds.) Wehrle. Peer-to-peer systems and applications. In Ralf Steinmetz and Klaus (Eds.) Wehrle, editors, *Peer-to-Peer Systems and Applications*, chapter 1, page 12. Springer-Verlag Berlin Heidelberg, 2005.

Quang Hieu Vu, Mihai Lupu, and Beng Chin Ooi. *Peer-to-Peer Computing: Principles and Applications*. Springer Publishing Company, Incorporated, 1st edition, 2009. ISBN 3642035132, 9783642035135.

Arturo Crespo and Hector Garcia-Molina. Routing indices for peer-to-peer systems. pages 23–, 2002. URL http://dl.acm.org/citation.cfm?id=850928.851858.

Fausto Giunchiglia, Uladzimir Kharkevich, and Alethia Hume. Semantic flooding. *World Wide Web*, 14(5):651–669, Oct 2011. ISSN 1573-1413. doi: 10.1007/s11280-010-0108-y. URL https://doi.org/10.1007/s11280-010-0108-y.

Xuemin Shen, Heather Yu, John Buford, and Mursalin Akon. *Handbook of Peer-to-Peer Networking*. Springer Publishing Company, Incorporated, 1st edition, 2009. ISBN 0387097503, 9780387097503.

Andrei Gurtov Dmitry Korzun. Structured peer-to-peer systems. In *Structured Peer-to-Peer Systems*, chapter 13, page 366. Springer, New York, NY, 2013.

Beverly Yang and Hector Garcia-Molina. Designing a super-peer network. 2002a. URL http://dbpubs.stanford.edu:8090/pub/2002-13.

Beverly Yang and Hector Garcia-Molina. Improving search in peer-to-peer networks. In *Proceedings of the 22 Nd International Conference on Distributed Computing Systems (ICDCS'02)*, ICDCS '02, pages 5–, Washington, DC, USA, 2002b. IEEE Computer Society. ISBN 0-7695-1585-1. URL http://dl.acm.org/citation.cfm?id=850928.851859.

Seth Terashima. Ilustração de uma rede chord. 2017. URL https://en.wikipedia.org/wiki/Chord_(peer-to-peer).

Ion Stoica, Robert Morris, David Karger, M. Frans Kaashoek, and Hari Balakrishnan. Chord: A scalable peer-to-peer lookup service for internet applications. *SIGCOMM Comput. Commun. Rev.*, 31(4):149–160, August 2001. ISSN 0146-4833. doi: 10.1145/964723.383071. URL http://doi.acm.org/10.1145/964723.383071.

Technische Universität Darmstadt. Ilustração de uma rede can. 2017. URL http://peerfact.kom.e-technik.tu-darmstadt.de/de/overview/layered-architecture/overlay-layer/index.html.

Sylvia Ratnasamy, Paul Francis, Mark Handley, Richard Karp, and Scott Shenker. A scalable content-addressable network. *SIGCOMM Comput. Commun. Rev.*, 31(4):161–172, August 2001. ISSN 0146-4833. doi: 10.1145/964723.383072. URL http://doi.acm.org/10.1145/964723.383072.

Petar Maymounkov and David Mazières. *Kademlia: A Peer-to-Peer Information System Based on the XOR Metric*, pages 53–65. Springer Berlin Heidelberg, Berlin, Heidelberg, 2002. ISBN 978-3-540-45748-0. doi: 10.1007/3-540-45748-8_5. URL https://doi.org/10.1007/3-540-45748-8_5.

Jahn Arn Johnsen, Lars Erik Karlsen, and Sebjørn Sæther Birkeland. Peer-to-peer networking with bittorrent. December 2005. URL http://web.cs.ucla.edu/classes/cs217/05BitTorrent.pdf.

Purvi Shah and Jehan-François Pâris. Incorporating trust in the bittorrent protocol. 2007.

Arnaud Legout, G. Urvoy-Keller, and P. Michiardi. Rarest first and choke algorithms are enough. In *Proceedings of the 6th ACM SIGCOMM Conference on Internet Measurement*, IMC '06, pages 203–216, New York, NY, USA, 2006. ACM. ISBN 1-59593-561-4. doi: 10.1145/1177080.1177106. URL http://doi.acm.org/10.1145/1177080.1177106.

Johan Pouwelse, Paweł Garbacki, Dick Epema, and Henk Sips. *The Bittorrent P2P File-Sharing System: Measurements and Analysis*, pages 205–216. Springer Berlin Heidelberg, Berlin, Heidelberg, 2005. ISBN 978-3-540-31906-1. doi: 10.1007/11558989_19. URL https://doi.org/10.1007/11558989_19.

Bram Cohen. Incentives build robustness in bittorrent. May 2003. URL http://www.bittorrent.com/bittorrentecon.pdf.

Andrew S. Tanenbaum. *Modern Operating Systems*. Prentice Hall Press, Upper Saddle River, NJ, USA, 3rd edition, 2007. ISBN 9780136006633.

IBM. Estrutura do file system. 2017. URL https://www.ibm.com/support/knowledgecenter/zosbasics/com.ibm.zos.zconcepts/zconcepts_177.htm.

Data Network Resource. Estrutura do file system unix. 2017. URL http://www.rhyshaden.com/unix.htm.

Dennis M. Ritchie and Ken Thompson. The unix time-sharing system. *Commun. ACM*, 17(7):365–375, July 1974. ISSN 0001-0782. doi: 10.1145/361011.361061. URL http://doi.acm.org/10.1145/361011.361061.

Andries E. Brouwer. The linux virtual file system. 2017. URL https://www.win.tue.nl/~aeb/linux/lk/lk-8.html.

Abraham Silberschatz, Peter Baer Galvin, and Greg Gagne. *Operating System Concepts*. Wiley Publishing, 8th edition, 2008. ISBN 0470128720.

Libfuse. Filesystem in userspace. 2017. URL https://github.com/libfuse/libfuse.

Antti Kantee and Alistair Crooks. Refuse: Userspace fuse reimplementation using puffs. 2007.

Sven. Fluxograma mostrando como o fuse funciona. 2017. URL https://en.wikipedia.org/wiki/Filesystem_in_Userspace.

Jeffrey B. Layton. User space file systems. 2017. URL http://www.linux-mag.com/id/7814/.

Arvid Norberg. Libtorrent website. 2017b. URL http://libtorrent.org/.

Arvid Norberg. Documentação do libtorrent. 2017c. URL http://www.libtorrent.org/reference-Core.html#session.

GitHub. Github do libfuse. 2017. URL https://github.com/libfuse/libfuse.

Aashish Barnwal. Funcionamento da política de lru. 2017. URL http://www.geeksforgeeks.org/lru-cache-implementation/.

Michael Kerrisk. fallocate man page. 2017. URL http://man7.org/linux/man-pages/man2/fallocate.2.html.