

Implementando o BitThief e o BitTyrant

Bruno M. Finelli
Rafael C. Almeida
<{finelli,rafaelc}@dcc.ufmg.br>

17 de Dezembro de 2007

1 Introdução

O BitTorrent é uma rede peer-to-peer (P2P) criada originalmente por Bram Cohen. O objetivo é criar uma rede para distribuição de um arquivo entre todos os nós. Uma das grandes dificuldades de redes P2P é fazer busca nos arquivos; o BitTorrent “resolve” isso passando esse problema para algum site web. A rede em si está preocupada em distribuir apenas um arquivo. Para transferir o arquivo entre os nós ele usa uma política tit-for-tat (TFT), em que cada nó faz upload para o outro apenas após receber algum pedaço do arquivo do nó remoto.

Nessa rede existem dois tipos de nós: *leechers* e *seeders*. Os *leechers* são aqueles que ainda estão baixando o arquivo e os *seeders* são aqueles que já tem o arquivo completo e estão cedendo o arquivo aos outros. Para que a rede funcione bem tanto os *seeders* quanto os *leechers* devem distribuir o arquivo. A política TFT tenta garantir que isso aconteça. Entretanto, o artigo “Free Riding in BitTorrent is Cheap” mostra que é possível ser um free-rider numa rede BitTorrent. E o artigo “Do incentives build robustness in BitTorrent?” mostra que, mesmo usando TFT, é possível fazer uma escolha mais sensível dos nós na rede para melhorar a taxa de download.

2 Objetivo

Neste trabalho temos como objetivo estudar o funcionamento dos clientes *BitThief* e *BitTyrant*. Desejamos entender as dificuldades de implementá-los e verificar se é possível obter os resultados previstos nos artigos que descrevem seu funcionamento. Além disso, desejamos comparar o funcionamento do *BitTyrant* e *BitThief* entre si, além da comparação com o cliente original.

3 BitThief

O *BitThief* é um cliente *BitTorrent* que tem como objetivo obter as maiores taxas de *download* possíveis sem fazer *upload* algum. Para conseguir isso nem sempre a especificação do *BitTorrent* é respeitada. O programa foi descrito no artigo [1]

Os clientes *BitTorrent* fazem um *optimistic unchoke* a cada 30 segundos. Nesse tempo um cliente recebe dados sem ter feito qualquer *upload*. O objetivo dessa técnica é encontrar novos *peers* que sejam bons *uploaders* que não estavam fazendo *upload* por estarem no estado *choked*. Devido a política TFT, só se faz *upload* àqueles que estão dando algo em troca, exceto no caso do *optimistic unchoke*.

O *BitThief* confia nessa técnica para conseguir fazer *download* de outros *leechers*. Ao invés de buscar os arquivos mais raros primeiro, qualquer *unchoke* que ele recebe ele faz o *download*. Isso garante que ele vai conseguir pegar o máximo de arquivos que oferecem a ele.

Além disso, o *BitThief* faz mais buscas no *tracker* do que o *tracker* recomenda. A maioria dos *trackers* não tem controle sobre isso e deixam o cliente fazer quantas buscas queira. O objetivo desse grande número de buscas é conseguir o máximo de vizinhos possível, dessa forma melhorando as chances de receber um *optimistic unchoke*. Para conseguir o maior número de *unchokes* possível, o cliente *BitThief* sempre se anuncia como um cliente novo que acabou de entrar na rede.

Por fim, o *BitThief* utiliza bastante os *seeders*, uma vez que eles não tem qualquer controle sobre quem está fazendo mais ou menos *upload*. Assim, o *seeder* vê o *BitThief* como um cliente normal que acabou de entrar e nunca irá desconfiar que ele é um cliente trapaceiro.

3.1 Implementação

A implementação original do *BitThief* foi feita em Java e o código não está disponível para ser verificado. Por tanto, nossa implementação, feita em cima do cliente *BitTorrent* original versão 3.4.2 (versão essa que está disponível no repositório Debian e Ubuntu), pode ser de grande utilidade a todos aqueles que queiram estudar o cliente *BitThief*

As modificações foram feitas aos arquivos *BitTorrent/Connecter.py*, *BitTorrent/PiecePicker.py*, *BitTorrent/Uploader.py*, *BitTorrent/Rerequester.py* e *BitTorrent/download.py*. Eles são responsáveis pela comunicação do cliente com outros *peers* e com o *tracker*.

O cliente *BitThief* sempre envia a mensagem de bitfield com todos os bits zerados, ou seja, os outros clientes nunca pedem nada para o *BitThief* pois ele se parece um cliente que acabou de conectar na rede. Além disso, nós nunca enviamos qualquer pedaço, mas também não enviamos nem mensagens de *choke* ou *unchoke*. Os clientes devem identificar por si só que não

estamos enviando nada, se eles tiverem essa capacidade.

Em *BitTorrent/PiecePicker.py* abandonamos a estratégia de *rarest first* e nos colocamos interessados em qualquer pedaço que nós não tenhamos pego ainda.

Ao *tracker* sempre anunciamos que ainda não fizemos qualquer download ou upload, o que mantém nosso rating sempre em 0. Os contatos com o *tracker* crescem exponencialmente, de acordo com a função *new_interval*, implementada em *BitTorrent/Rerequester.py*.

3.2 Experimentos

Depois da implementação das idéias do artigo fizemos alguns experimentos com o *BitThief*. Para nossa surpresa os resultados foram tão bons ou melhores que os apresentados no artigo.

Fizemos testes do *BitThief* com várias redes reais. Para isso fizemos *download* de diversos torrents em *www.btjunkie.org*. Para todas as redes o *BitThief* conseguiu altas taxas de *download* e terminou antes do cliente original. Além disso, ele não fez nenhum *upload*, como esperado. Ao contrário do cliente original.

Para descobrir se realmente não fizemos *upload* utilizamos o aplicativo *tcpdump* para verificar o conteúdo dos pacotes. Um script era capaz de identificar se haviam pacotes de dados do *BitTorrent*.

Fizemos testes com dez *torrents* diferentes. Incluindo *torrents* do Debian e Ubuntu. O mesmo *torrent* foi utilizado em dias diferentes, mas no mesmo horário e no mesmo computador. Com isso objetivamos manter as condições o mais constante possível.

Os *torrents* referentes ao Debian e Ubuntu mostraram comportamento muito semelhante no cliente original e no *BitThief*, uma vez que eles tinham uma grande quantidade de *seeders*, o cliente original praticamente não fez qualquer *upload*. O tempo de *download* de ambos foi aproximadamente igual. O número de conexões do *BitThief* e o cliente original também permaneceram constantes.

Os gráficos para os demais *torrents*, onde o número de *leechers* e *seeders* é parecido, ficaram bastante semelhantes. Portanto, basta analisar um dos casos. Os gráficos das figuras 1 a 5 referem-se ao *torrent* de um arquivo de 1.4GB.

Para deixar a comparação justa, permitimos que o cliente original fizesse o mesmo tanto de conexões quanto o *BitThief*. Veja que eles mantem aproximadamente o mesmo tanto de conexões (figuras 1 e 3), entretanto, a taxa de download do *BitThief* é, em média, bem maior que a do cliente original. Na figura 2 podemos ver que o *BitThief* termina o download em pouco mais de 250 minutos, enquanto isso, na figura 4 vemos que o cliente original demora mais do que o dobro de tempo. Essa variação se deve a taxa de *upload* que o cliente original precisou fazer. No total, o cliente original fez 570MB de

upload. Essa taxa de upload fez com que o a taxa de download sofresse, afinal, com uma conexão ADSL de 1Mbps – ambiente que os clientes foram testados – mesmo uma pequena taxa de upload pode atrapalhar o *download*. Além disso, a política de sempre buscar os pedaços mais raros, apesar de garantir que existam boas quantidades de cada pedaço na rede, não é a forma mais vantajosa para um cliente específico.

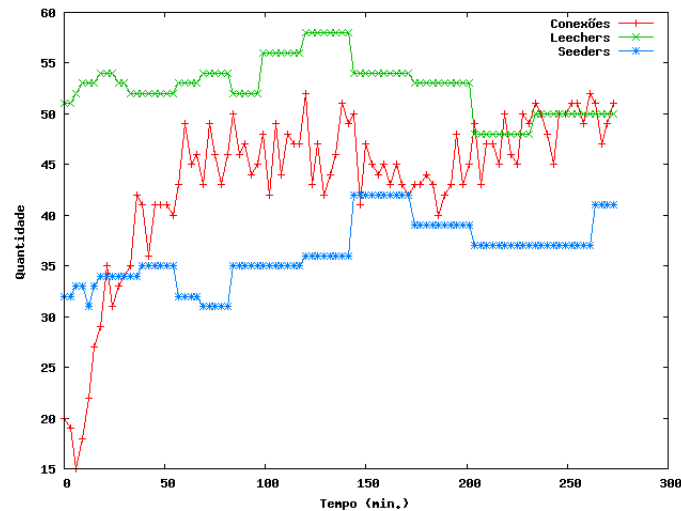


Figura 1: Conexões feitas pelo cliente BitThief ao usar o torrent de 1.4GB

O cliente *BitThief* apenas se saiu pior do que o *BitTorrent* em um teste em rede local, onde existia apenas um seeder e todos os outros clientes eram *BitTorrent*, exceto um *BitThief*. Nessa configuração o cliente *BitThief* eventualmente conseguiu fazer o download completo, mas demorou mais do que a maioria dos clientes *BitTorrent*.

4 BitTyrant

O *BitTyrant* é um cliente *BitTorrent* que tem como objetivo obter as maiores taxas de *download* possíveis sendendo as menores taxas de *upload*. Para isso, o cliente utiliza uma estratégia de escolha de nós vizinhos que aumente ao máximo o custo benefício de suas conexões.

Para calcular o custo benefício de cada conexão, separamos os nós em dois grupos: *seeders* e *leechers*. Como não fazemos *upload* para os *seeders*, definimos que o custo benefício para esse tipo de conexão é de $1000 * Taxa\ de\ Download$. Para os *leechers*, o custo benefício é definido pela divisão da *Taxa de Download* pela *Taxa de Upload*.

O *BitTyrant* realiza o maior número de conexões possíveis. Como o cliente busca estabelecer as conexões que lhe geram o maior custo benefício,

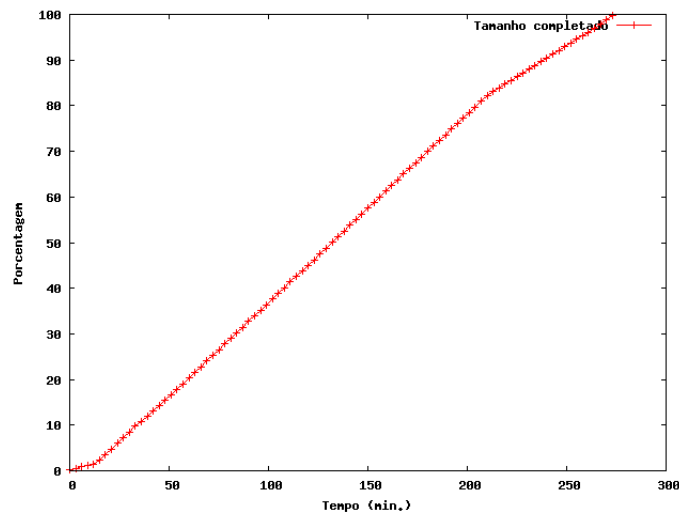


Figura 2: Download feito pelo cliente BitThief ao usar o torrent de 1.4GB

o cliente simplesmente realiza conexões até que a soma das taxas de *upload* supere o tamanho da banda de *upload*.

Por fim, visando melhorar ainda mais o custo benefício de cada conexão, o cliente utiliza a seguinte estratégia: se o nó com qual está conectado responde de forma recíproca ao envio de dados, então a taxa de *upload* para esse nó será diminuída em 10%; se o nó com qual está conectado não responde de forma recíproca ao envio de dados, então a taxa de *upload* para esse nó será aumentada em 20%. Dessa forma garantimos que a quantidade de banda de *upload* gasta, é exatamente aquela necessária para obtermos *download*, excluindo qualquer ação altruísta por parte do cliente *BitTyrant*.

4.1 Implementação

A implementação original do *BitTyrant* foi feita em *Java* baseada no cliente *Azureus* e seu código está disponível em:

<http://BitTyrant.cs.washington.edu/>

A nossa versão foi feita baseada no cliente *BitTorrent* original versão 3.4.2 (versão essa que está disponível no repositório Debian e Ubuntu).

As modificações foram feitas aos arquivos *BitTorrent/Choker.py*, *BitTorrent/Connecter.py* e *BitTorrent/download.py*. Esses módulos são os responsáveis pela definição de quais nós o cliente irá se conectar, assim como a taxa de *upload* para cada nó.

O arquivo *BitTorrent/Choker.py* foi modificado para calcular o custo benefício de cada conexão e o número de conexões que o cliente irá realizar.

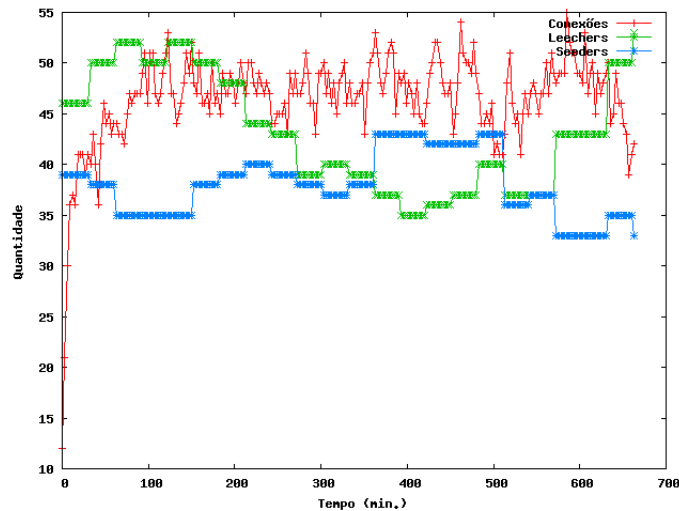


Figura 3: Conexões feitas pelo cliente original ao usar o torrent de 1.4GB

O arquivo *BitTorrent/Connector.py* foi modificado para limitar a taxa de *upload* para um determinado nó (o cliente original só consegue limitar a taxa de *upload* global) e alterar o limite da taxa de *upload* para um determinado nó quando este envia uma mensagem de *Choke* (aumenta o valor em 20%) ou *Unchoke* (diminui o valor em 10%).

5 Módulo estatísticas

Grande parte do trabalho realizado foi análise de estatísticas, portanto foi necessário adaptar o cliente original para que ele nos desse informações sobre como ocorreu o download.

Criamos o módulo *BitTorrent/Statistics.py* e adaptamos o módulo *BitTorrent/download.py* para chamar esse módulo. Na classe *Statistics* nós registramos uma função no escalonador da classe *RawServer*. Essa função é chamada de acordo com o tempo, em segundos, passado como parâmetro para a opção `—take_stats`, adicionada ao *BitTorrent* neste trabalho.

A classe *RawServer* possui um método, chamado *listen_forever* onde o programa passa a maior parte de sua execução. O método implementa um loop usando a função *poll* para negociar com os *sockets*. Quando se registra uma função nessa classe ela começa a chamar a função de acordo com um tempo determinado na hora do registro. Para fazer isso a *listen_forever* usa o timeout da função *poll*.

O módulo criado possui um template de um script que é capaz de usar os dados obtidos para gerar gráficos no *gnuplot*. Esse template é preenchido com as estatísticas obtidas a cada novo tempo. São elas: número de *seeders*,

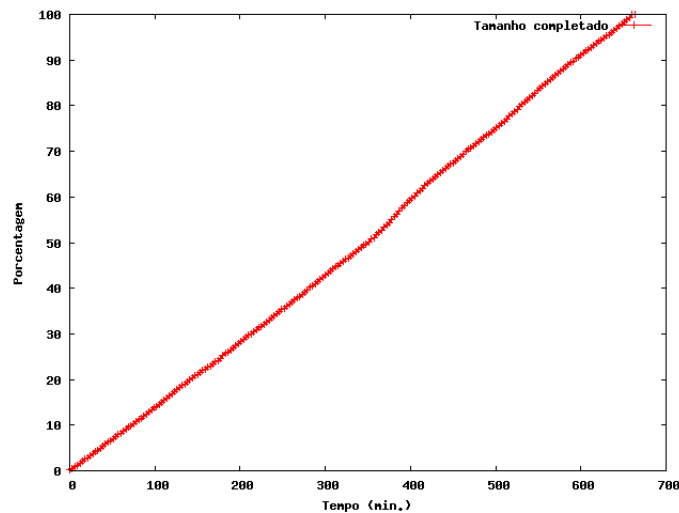


Figura 4: Download feito pelo cliente original ao usar o torrent de 1.4GB

número de *leechers*, total de download já feito até o momento e total de upload já feito até o momento.

Quando o download do torrent termina um script é gerado, esse script, quando executado, gera figuras como as figuras 1 a 5 deste documento. É possível sabermos o tempo total necessário para o download multiplicando o número de entradas pelo delay entre elas.

6 Comparação

Para comparar as diversas implementações utilizamos duas abordagens: teste na Internet e teste em rede local. Enquanto a primeira mostra o comportamento individual de cada nó em uma rede *BitTorrent* que está em funcionamento no mundo real, a segunda mostra o impacto de redes que, por ventura, venham a conter apenas nossos clientes modificados.

6.1 Teste na Internet

Uma das vantagens de modificar o cliente *BitTorrent* para que ele funcione de acordo com o que foi descrito nos artigos lidos é que podemos testar o comportamento do programa em uma rede *BitTorrent* já ativa. Dessa forma evitamos as imprecisões de uma simulação e podemos analisar se as ideias são realmente viáveis e como elas são comparadas umas as outras.

Para fazer esse teste, escolhemos um torrent, de um arquivo de aproximadamente 70 MB, do site *www.btjunkie.org*, e realizamos o download do arquivo em cada uma das três versões (original, *BitThief* e *BitTyrant*) em

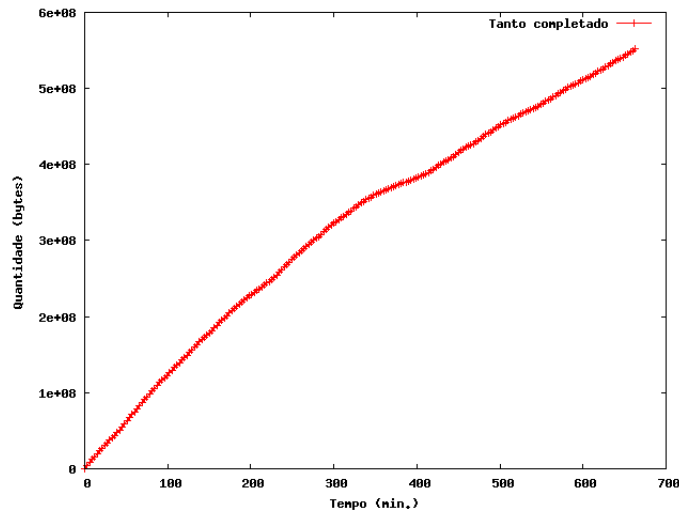


Figura 5: Download feito pelo cliente original ao usar o torrent de 1.4GB

um mesmo computador. Os resultados mostraram que o *BitTyrant* é o mais rápido dos três, como já esperado. Como o arquivo baixado era relativamente pequeno, o *BitThief* teve uma boa performance, devido a agressividade com que o cliente estabelece o número de conexões, e foi o segundo mais rápido.

As figuras de 6 a 15 mostram os resultados obtidos para cada uma das versões.

6.2 Teste em rede local

Ao criar uma rede local nós pudemos avaliar qual seria o impacto dos clientes modificados caso eles dominassem a rede. Para que a rede se pareça com o que encontramos na realidade precisamos que diferentes máquinas tivessem diferentes taxas de download/upload. Para fazer isso foi utilizado o programa *trickle*.

O *trickle* é um programa capaz de modificar as taxas máximas de upload e download que um programa consegue através de conexões TCP. O programa é implementado totalmente na *userland* e não precisa de permissão de superusuário para ser executado. O que foi bem interessante, tendo em vista que não tínhamos tal permissão nas máquinas utilizadas.

Para iniciar a execução de cada uma das implementações de *BitTorrent* criamos alguns *bash scripts*. Esses scripts e as implementações de *BitTorrent* podem ser encontrados em [3].

Criamos um arquivo de 40MB e um *.torrent* referente. Iniciamos o *tracker* na máquina 150.164.6.19 e transferimos o arquivo via SSH para as

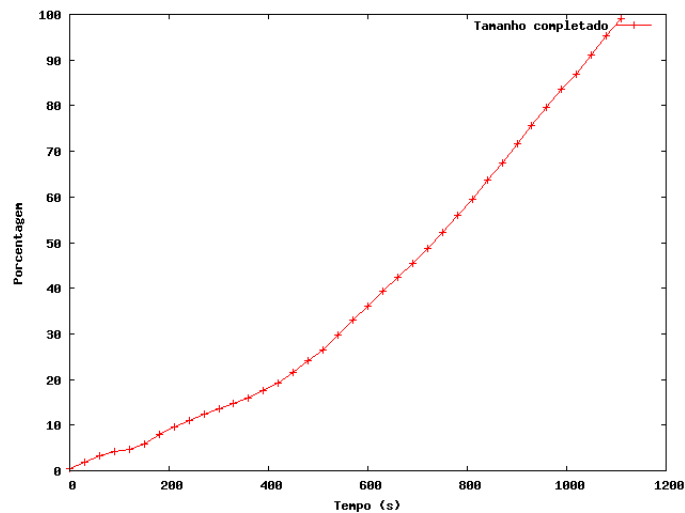


Figura 6: Download feito pelo cliente original ao usar o torrent de 70MB

máquinas que queríamos deixar com seed. As outras máquinas baixaram o arquivo através da rede torrent. Tínhamos ao total 20 máquinas rodando os algoritmos.

Utilizamos quatro configurações diferentes de redes locais:

- Um seeder e o resto dos nós BitThief
- Todos os nós BitTorrent
- Todos os nós BitTyrant
- Todos os nós BitTyrant e um nó BitThief

As demais configurações são supridas pelo teste na Internet.

6.2.1 Um seeder para vários BitThiefs

Ao se colocar um *seeder* e o resto dos nós usando o *BitThief* todos nós fizeram download apenas do *seeder*, de forma muito parecida a que acontece quando vários nós fazem download de uma fonte centralizada, como um servidor FTP. Esse comportamento era esperado, o teste foi feito apenas para manter este documento completo.

6.2.2 Redes homogenas de BitTorrent e BitTyrant

Criamos dois nós *seeders*, ambos rodando o algoritmo BitTyrant. O restante das máquinas rodamos nós BitTyrant como *leechers*. Sorteamos aleatoriamente taxas de upload e download entre os nós, variando de 100KB/s a

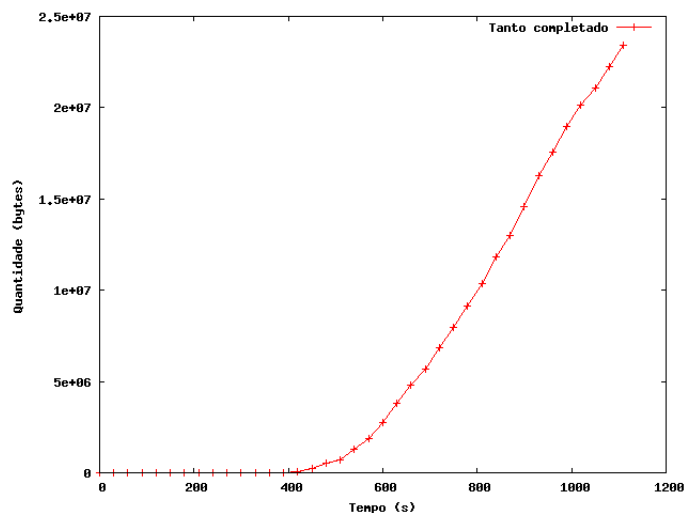


Figura 7: Upload feito pelo cliente original ao usar o torrent de 70MB

400KB/s. Deixamos os BitTyrants rodando até que todos nós tivessem o arquivo.

Depois criamos uma rede com as mesmas taxas de download e upload para cada nó, mas trocando o BitTyrant por BitTorrent. Com isso objetivamos comparar o comportamento dos dois tipos de rede.

A rede BitTyrant demorou menos tempo para se estabilizar, levando um total de 4200s, contra os 4500s que demorou a rede BitTorrent. A média entre os dois algoritmos ficou parecida, enquanto cada nó na rede BitTorrent demorou, em média, 3536s, a média no BitTyrant foi de 3420s.

A maior diferença entre os dois foi que, no BitTyrant, os nós com menor banda demoraram mais tempo para conseguir seus arquivos, que no BitTorrent. Enquanto os nós com maior banda demoraram mais tempo para conseguir o arquivo na rede BitTorrent do que na BitTyrant.

Os testes mostram que é possível se manter uma rede com apenas BitTyrant, ainda que ela prejudique os nós de banda mais baixa. Para se ter certeza de que a rede BitTyrant é boa o suficiente para ser a nova implementação oficial do BitTorrent ainda são necessários mais testes. Com redes maiores e com churn. No entanto, os testes aqui apresentados são bastante animadores e a implementação parece ser capaz de funcionar sem a necessidade de nós com outras implementações.

6.2.3 Um BitThief em uma rede de BitTyrant

Criamos dois nós *seeders*, ambos rodando o algoritmo BitTyrant. O restante das máquinas rodamos nós BitTyrant, exceto uma, que rodava o BitThief.

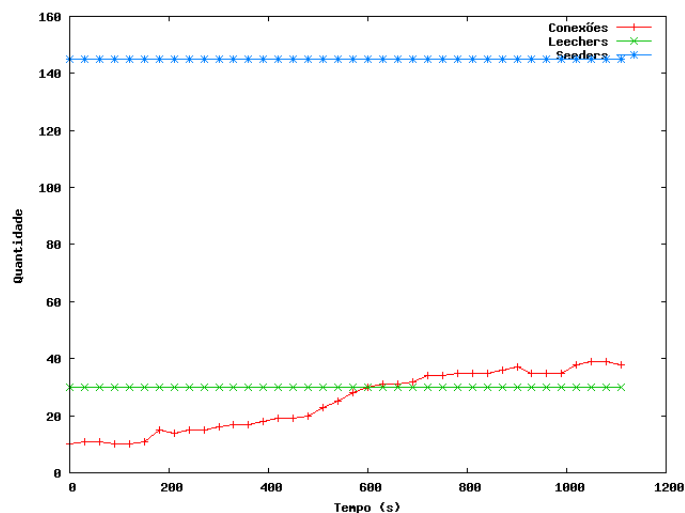


Figura 8: Conexões feitas pelo cliente original ao usar o torrent de 70MB

Para efeitos de comparação, limitamos todos nós a uma taxa de 300KB/s de download e 300KB/s de upload.

Como resultado a rede demorou 1980s para se estabilizar, isto é, todos os nós terem o arquivo (incluindo o BitThief). O cliente BitThief foi o último a terminar, demorando 1980s para obter o arquivo. Entre os nós BitTyrant, o maior tempo gasto foi de 1800s, sendo que o menor foi de 900s. É interessante notar que essa diferença costuma ser maior nas redes BitTyrant do que nas redes BitTorrent regulares.

Em uma configuração semelhante, mas usando BitTorrents não modificados, o BitThief demorou o mesmo tempo para obter o arquivo. Isso vem a mostrar que o BitThief consegue terminar o arquivo independente do algoritmo utilizado, mesmo que demore mais que todos outros nós.

Nos dois casos o BitThief teve desempenho tão parecido porque, uma vez que um nó se torna *seeder*, ele se comporta de maneira parecida no BitTyrant e no BitTorrent. Para comprovar isso, fizemos o mesmo teste, em uma rede muito parecida, mas deixávamos apenas dois *seeders* ativos, em qualquer momento da rede. Ao fazer isso, o BitThief demorou mais para terminar o arquivo na rede BitTyrant do que na rede BitTorrent. No entantanto, ele terminou o arquivo nas duas.

É importante notar que, na pior das hipóteses, o BitThief confia apenas nos *seeders* para fazer download. Portanto, as políticas do *BitTyrant*, que nada modificam os *seeders*, não impedem que o BitThief funcione. Nas figuras 14 e 15 podemos ver o comportamento de como o download foi feito pelo *BitThief* nas redes BitTorrent e BitTyrant. Note que no final o download fica consideravelmente mais lento, isso é porque o BitThief começa a

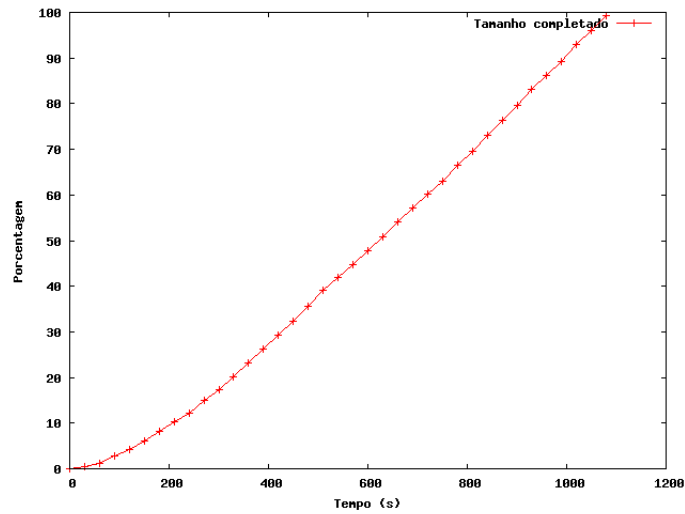


Figura 9: Download feito pelo cliente BitThief ao usar o torrent de 70MB

ter que competir com outros clientes por pedaços mais raros do arquivo.

7 Conclusão

Notamos que é um tanto difícil conseguir-se sistemas que impedem injustiças. O *BitThief* foi capaz de terminar os seus downloads em todo tipo de rede, mesmo que se beneficiasse em algumas e em outras seus downloads fossem piores. Além disso, o *BitTyrant*, apesar de ter certo suporte a reputação, ela não é avançada o suficiente para evitar qualquer tipo de abuso.

O *BitThief*, na maior parte das redes BitTorrent que encontramos na Internet, foi capaz de dar boas vantagens para o nosso nó. O que deixa isso interessante é que, apesar desse benefício individual, ainda assim o *BitThief* não é muito utilizado. As redes *BitTorrent* continuam funcionando bem, apesar da possibilidade das pessoas de usar o *BitThief*. A análise de porque a maioria das pessoas escolhem não roubar em redes P2P pode ser um interessante estudo sociológico.

Apesar do *BitTyrant* modelar sistema de reputação no *BitTorrent*, parece que a maior parte das pessoas não vêem muita vantagem nele e acabam por escolher novos clientes. Isso se deve, em parte, ao fato da maioria das redes *BitTorrent* já funcionarem muito bem da forma que estão construídas e o *BitTyrant* não dá resultados extremamente melhores que o algoritmo convencional na maior parte dos casos.

Nós acreditamos que talvez seja possível modelar reputação em uma rede *BitTorrent* de forma mais agressiva, contando com a ajuda do *tracker*. Se os nós forem capazes de notificar ao *tracker* de quem eles recebem upload,

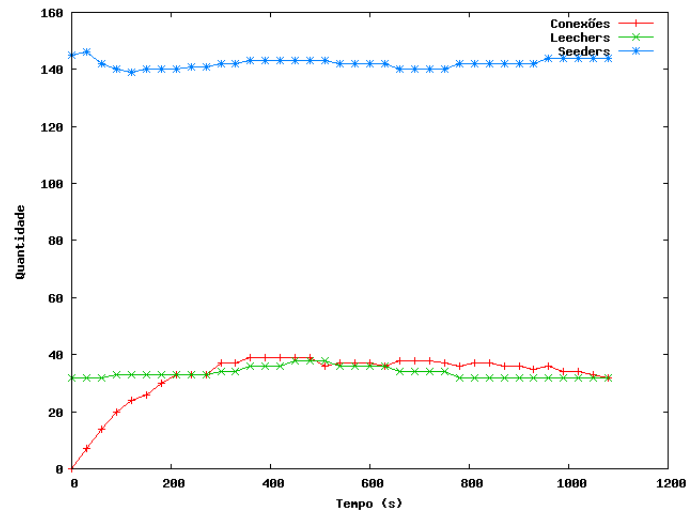


Figura 10: Conexões feitas pelo cliente BitThief ao usar o torrent de 70MB

o *tracker* pode ser mais inteligente no momento de dar aos clientes a informação sobre os nós na rede, beneficiando os nós que mais contribuem. Acreditamos que esse seja um interessante campo para estudo, a natureza centralizada do tracker pode permitir um controle de reputação muito melhor, sem que fique excessivamente carregado.

Referências

- [1] Tomas Locher e Patrick Moor e Stefan Schmid e Roger Wattenhoffer. Free riding in bittorrent is cheap. *Hot Nets*, 2006.
- [2] Michael Piatek e Tomas Isdal e Thomas Anderson e Arvind Krishnamurthy e Arun Venkataramani. Do incentives build robustness in bittorrent? *NSDI 2007*, 2007.
- [3] Implementações. <http://www.dcc.ufmg.br/~rafaelc/ear>. Página contendo as implementações feitas para este documento, 2007.

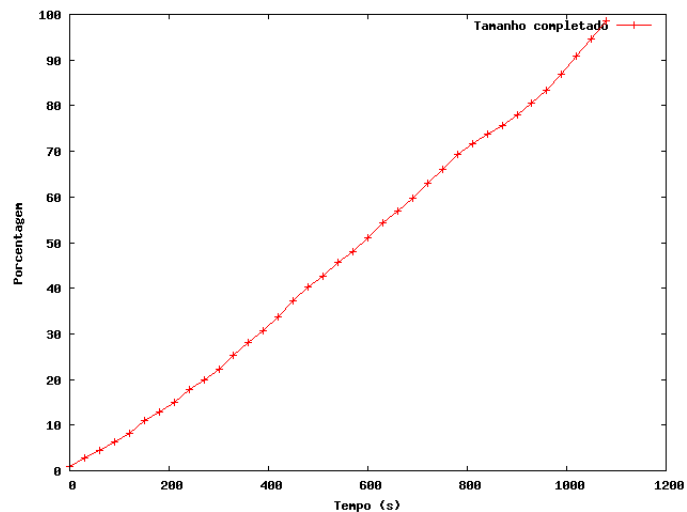


Figura 11: Download feito pelo cliente BitTyrant ao usar o torrent de 70MB

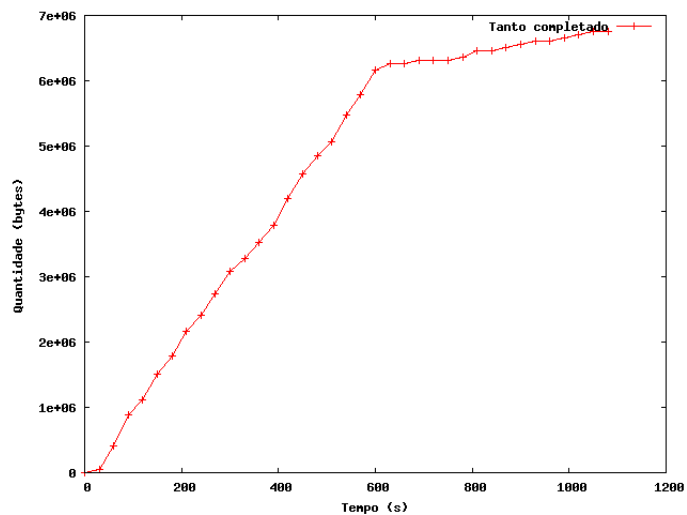


Figura 12: Upload feito pelo cliente BitTyrant ao usar o torrent de 70MB

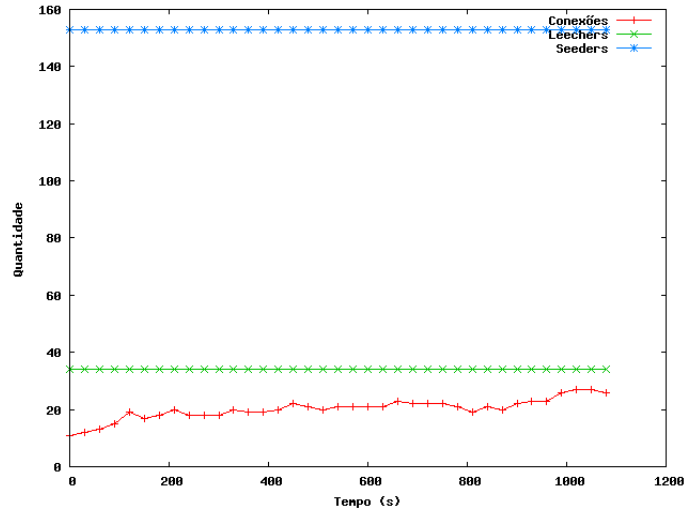


Figura 13: Conexões feitas pelo cliente BitTyrant ao usar o torrent de 70MB

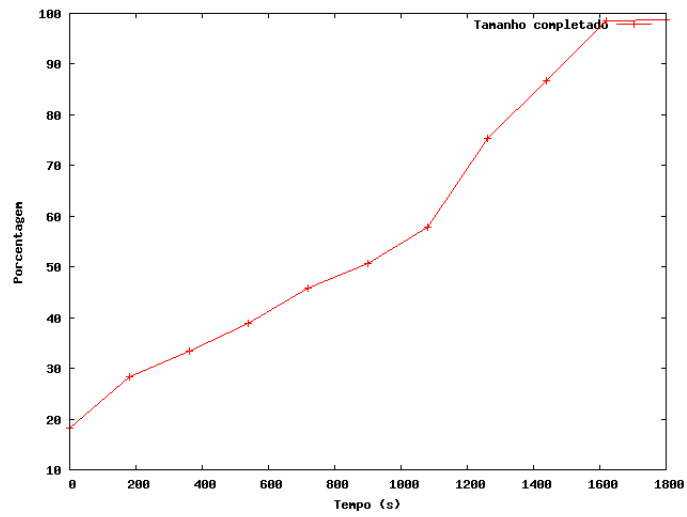


Figura 14: Comportamento do download do BitThief em uma rede BitTorrent

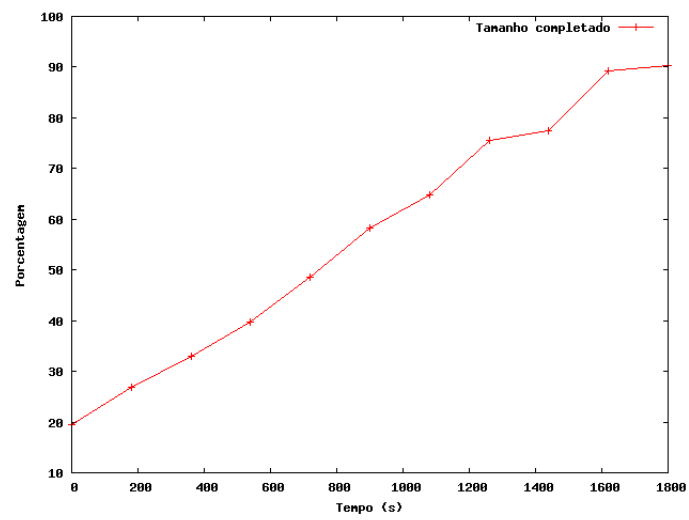


Figura 15: Comportamento do download do BitThief em uma rede Bit-Tyrant