UNIVERSIDADE FEDERAL DO ESTADO DO RIO DE JANEIRO
CENTRO DE CIÊNCIAS EXATAS E TECNOLÓGICAS
PROGRAMA DE PÓS-GRADUAÇÃO EM INFORMÁTICA

LIDANDO COM TRÁFEGO EGOÍSTA EM UMA REDE BITTORRENT

Maximiliano Martins de Faria

Orientadores
Morganna Carmen Diniz

Rio de Janeiro, RJ - Brasil
Julho 2010

LIDANDO COM TRÁFEGO EGOÍSTA EM UMA REDE BITTORRENT

Maximiliano Martins de Faria

DISSERTAÇÃO APRESENTADA COMO REQUISITO PARCIAL PARA OBTENÇÃO DO TÍTULO DE MESTRE PELO PROGRAMA DE PÓS-GRADUAÇÃO EM INFORMÁTICA DA UNIVERSIDADE FEDERAL DO ESTADO DO RIO DE JANEIRO (UNIRIO). APROVADA PELA COMISSÃO EXAMINADORA ABAIXO ASSINADA.

Aprovado por:

Morganna Carmen Diniz, D.Sc. - UNIRIO

Sidney Cunha de Lucena, D.Sc. - UNIRIO

Ana Paula Couto da Silva, D.Sc. - UFJF

Ronaldo Moreira Salles, Ph.D. - IME

Rio de Janeiro, RJ - Brasil
Julho 2010

F224

Faria,Maximiliano Martins de.
Lidando com tráfego egoísta em redes BitTorrent/Maximilaino Martins de Faria,2010.
xiii,91f

Orientador: Morganna Carmen Diniz.
Dissertação (Mestrado em Informática) - Universidade Federal do Estado do Rio de Janeiro, Rio de Janeiro, 2010.

1. Sistemas operacionais distribuídos (computadores). 2. Sistemas colaborativos. 3. Redes de computadores. 4. Freeriders – Medidas de segurança. 5. Redes BitTorrent. I. Diniz, Morganna Carmen. II. Universidade Federal do Estado do Rio de Janeiro (2003-). Centro de Ciências Exatas e Tecnologia. Curso de Mestrado em Informática. III. Título.

CDD-004.36

Aos meus filhos Alexander, Ludmila e minha companheira Gisele.

Agradecimentos

A Gisele, minha amada, companheira e amiga, aos meus filhos Alex e Ludmila, pela paciência e compreensão do tempo retirado deles.

À Profa. Morganna Carmem Diniz, pela amizade, paciência e determinação com que me orientou neste trabalho, sabendo compreender minhas limitações e me ajudando a superá-las.

Ao Prof. Asterio Kiyoshi Tanaka, por me reapresentar à academia e por nestes últimos anos ser um sábio conselheiro.

Ao amigo Henrique Andrade, por ouvir meus lamentos e desabafos nos momentos críticos deste trabalho.

Ao amigo Renato Rosas, pelo apoio logístico em equipamentos imprescindíveis nas pesquisas realizadas deste trabalho.

Ao amigo de longa data, Felipe Schueler, pelo meu sumiço nestes últimos seis meses finais.

À dona Terezinha Martins, minha mãe, por ter me ensinado acreditar, persistir e lutar.

Ao Nelson Faria, meu pai, por todo seu esforço ao longo da vida, que me ajudaram alcançar meus objetivos.

FARIA, Maximiliano Martins de. **Lidando com Tráfego Egoísta em uma rede BitTorrent**. UNIRIO 2010. 108 páginas. Dissertação de Mestrado. Departamento de Informática Aplicada, UNIRIO.

RESUMO

Com o crescimento da Internet e a popularização dos computadores, surgiu um forte apelo na colaboração entre os usuários. Esta colaboração acontece principalmente pelas vias digitais nas tranferências de arquivos em redes P2P. Este trabalho foca em uma destas redes mais conhecidas, o BitTorrent. Aqui é mostrado como este tipo de rede pode ser ameaçado por usuários maliciosos que se beneficiam de suas vulnerabilidades, rompendo a idéia principal do algoritmo que é a cooperação mútua entre seus usuários. Estes usuários maliciosos são conhecidos como *freeriders*, pois eles se beneficiam de excelentes taxas de *download* e contribuem com pequenas (ou quase nada) taxas de *upload*. O objetivo deste trabalho é propor duas políticas para combate de *freeriders* em rede BitTorrent. Para mostrar a eficácia de tais propostas foi implementado o protocolo BitTorrent no simulador NS-2.

Palavra-chave: Redes, BitTorrent, Sistemas distribuídos

Sumário

Lista de Tabelas

Lista de Figuras

1 Introdução

Com o crescimento da Internet e a popularização dos computadores, surgiu um forte apelo pela colaboração entre os usuários. Esta colaboração acontece principalmente pelas vias digitais nas transferências de arquivos entre eles. As redes *peer-to-peers* se encaixam perfeitamente neste cenário. Cada vez mais usuários buscam por informações na grande rede. Estas informações muitas vezes se concentram em *datacenters* de forma centralizada e com relativa segurança. Mas a busca por conteúdo, arquivos, vídeos, músicas, cresce mais que as estruturas destas organizações e as redes P2P preenchem esta lacuna pois as informações são distribuídas pelos próprios participantes da rede.

Segundo [16], aplicações P2P exploram recursos disponíveis da borda da internet - armazenamento, ciclos, conteúdo e presença humana. Estas redes, ao contrário de sistema centralizados, tem como força a distribuição de recursos por meio de usuários com um ou vários objetivos comuns, o conteúdo. Como citado em [19], os processos em uma rede P2P são simétricos: cada processo atua como um "cliente" e um "servidor" ao mesmo tempo. Esta característica abriu um campo vasto de pesquisa e começaram a aparecer diversas aplicações e protocolos *peer-to-peers*, todas tendo um ponto comum a distribuição descentralizada e a escalabilidade. Neste cenário, um protocolo se destacou, o BitTorrent, criado por Bram Cohen [2] em 2003. O BitTorrent rapidamente passou ser um dos protocolos mais usados para distribuição de conteúdo.

Como no modelo cliente servidor citado em [18], as redes *peer-to-peer* apresentam também limitações. Existem muitos problemas nestas redes a serem estudados, principalmente no que tange a colaboração de seus participantes, já que esta é a principal força da existência destas redes. Um dos grandes desafios em sistemas de difusão de conteúdo é garantir que todos os pares contribuam para o processo de difusão da informação. Cada par contribui com a sua taxa de *upload* com o intuito de acelerar a entrega de todo conteúdo. Em outras palavras, a implementação do sistema deve impossibilitar a presença de pares denominados *freeriders*, usuários que obtem o conteúdo sem contribuir para a difusão do mesmo, sendo que este comportamento pode ocorrer de forma intencional ou não.

A consequência da presença de *freeriders* pode ser devastadora para o desempenho de muitos sistemas P2P, gerando, principalmente um nível de injustiça muito grande com o restante dos pares que contribuem. Um amplo conjunto de trabalhos sobre este tópico

pode ser encontrado na literatura, mas ainda existem vários pontos a serem explorados. Entre eles, como garantir que todos os pares funcionem efetivamente e ativamente no processo da distribuição, deixando de lado características egoístas com o intuito do bem comum de todos pares.

Existem muitos trabalhos que abordam o problema dos *freeriders* em redes BitTorrent. É possível classificar este trabalho em dois tipos: os que se preocupam em qualificar e/ou quantificar a perda de desempenho da rede com a presença de *freeriders* e os que propõem alguma medida para lidar com os *freeriders*. A seguir é apresentada uma lista de alguns desses artigos.

Em [12], Qiu e Srikant usam modelo de fluido para estudar a escalabilidade, o desempenho e a eficiência do BitTorrent. Um dos resultados apresentados mostra que o uso do algoritmo *optimistic unchoking*, usado na rede BitTorrent para possibilitar que os pares encontrem novos vizinhos para troca de dados, permite a perda de até 20% da largura de banda de um cliente para os *freeriders*.

Em [1], Barbera *et al* apresentam um modelo markoviano para analisar o desempenho dos pares em uma rede BitTorrent. Os resultados numéricos do modelo mostram que o algoritmo *optimistic unchoking* não é eficiente no tratamento dos *freeriders*. Com base na análise, os autores propõem que seja modificada a forma de como um par é selecionado no algoritmo para ser perdoado.

Em [17], Sirivianos *et al* modificam um cliente BitTorrent de forma que ele adquire uma visão mais ampla de rede que o normal. Com isto, o comportamento do cliente é alterado em dois pontos: ele se conecta com todos os pares da rede e não responde a qualquer pedido por dados feito pelos outros clientes. O estudo mostra que a melhoria no desempenho dos clientes modificados é significativo, fazendo com que eles consigam uma alta taxa de *download* com zero taxa de *upload*. Mas isto provoca um efeito no desempenho da rede: à medida que o número de *freeriders* aumenta, tanto os clientes normais quanto os clientes modificados sofrem as consequências da degradação do ambiente.

Em [8], Legout *et al* mostram que os pares de um *swarm* tendem a se agrupar com pares que possuem larguras de banda semelhante a sua. Em [10], Murai e Figueiredo provam que a formação de clusters por largura de banda é consequência do uso do algoritmo *optimistic unchoking*. Entretanto, em [14], Sherman *et al* demonstram que a presença de *freeriders* impede o agrupamento de nós, pois o protocolo não é capaz de lidar com comportamentos egoístas.

Em [20], Zghaibeh e Harmantzis apresentam um amplo estudo do comportamento dos usuários em redes BitTorrent. São algumas das conclusões deste trabalho: o volume de *freeriders* tem crescido chegando atualmente a cerca de 16,8% dos pares em um *swarm*; os *freeriders* não estão associados a determinadas faixas de lagura de banda, é possível encontrar *freeriders* com alta largura de banda; o algoritmo *optimistic unchoking* é o ponto fraco do mecanismo *Tit-for-Tat* e deveria ser modificado.

Em [14], Sherman *et al* propõem a criação de um grupo especial de pares chamados de *Trusted Auditors* (TAs). Esses pares participam do *swarm* como qualquer outro par, mas possuem uma tarefa adicional: observam e classificam os clientes de acordo com o nível de contribuição de cada um. O que os TAs descobrem sobre o comportamento dos pares é propagado para toda a rede.Os resultados dos testes mostraram que os TAs garantem o bom desempenho do sistema apesar do aumento do tempo de download dos clientes não *freeriders*.

Em [15], Shin *et al* propõem um esquema chamado *Treat-Before-Trick* (TBeT) que penaliza os *freeriders* e recompensa o comportamento participativo dos pares. Neste esquema, os arquivos são divididos em pedaços e criptografados pelo *seeder*. A chave usada para criptografar um arquivo é dividida em n subchaves, onde quaisquer t dessas subchaves são suficientes para recuperar a chave original e descriptografar os pedaços do arquivo. O *seeder* então distribui os pedaços do arquivo e as subchaves entre os pares. A ideia é que os pares troquem pedaço de arquivo por subchave, ou seja, para receber a subchave em poder de um vizinho, o par deve primeiro enviar o pedaço de arquivo que possui e que foi solicitado por esse par. As simulações feitas mostram que os *freeriders* são eliminados ou, pelo menos, severamente penalizados com o uso do TBeT, enquanto o tempo de *download* dos pares participativos são reduzido em até 50%.

O protocolo BitTorrent, sugere que o comportamento de cooperação entre os pares é essencial para o bom funcionamento do sistema, especialmente porque o protocolo é baseado no mecanismo de troca recíproca, conhecido como *Tit-for-Tat* [13]. Além disso, a implementação do BitTorrent enfatiza que o seu mecanismo de incentivo é robusto contra comportamento malicioso ou egoísta [2]. No entanto, estudos recentes questionam e sugerem que até mesmo o mecanismo de incentivo implementado pelo BitTorrent não seja suficiente para deter usuários que frustam a reputação de justiça e robustez do protocolo [20, 7, 9].

O objetivo deste trabalho é apresentar duas políticas de combate aos *freeriders*. A primeira pontua pares que pouco contribuem dentro de uma rede BitTorrent. Quando esses pares alcançarem uma determinada pontuação, eles são marcados como *freeriders*. A segunda é semelhante à política 1, com a diferença que os pontos agora são atribuídos aos pares que melhor contribuem dentro de uma rede BitTorrent.

Este trabalho possui a seguinte organização: o capítulo 2 discute as principais características do protocolo BitTorrent; o capítulo 3 comenta o ambiente de testes utilizado para coleta e análise do protocolo; o capítulo 4 apresenta duas políticas de combate aos *freeriders* e os resultados obtidos com a aplicação dessas políticas; por último, o capítulo 5 resume as conclusões.

2 Protocolo BitTorrent

2.1 Introdução

Em uma arquitetura *peer-to-peer*[1] (P2P), os elementos da rede (nós) tanto fazem papel de clientes quanto de servidor. A figura 2.1 mostra um tipo de rede P2P conhecida como rede P2P híbrida. Este tipo de rede *peer-to-peer* mantém uma arquitetura descentralizada, mas existe o elemento servidor com funções específicas como: organizar a rede e armazenar informações estatísticas. Esta arquitetura ganhou força com o crescimento de usuários na Internet e com o aumento da distribuição de arquivos entre os mesmos já que se mostra altamente escalável, pois pode crescer com entradas de novos pares no sistema, e robusta, já que a informação se encontra descentralizada e não depende de uma única fonte de dados.

O BitTorrent é um protocolo para redes *peer-to-peer* do tipo híbrida (figura 2.1) para compartilhamento de arquivos criado por Bram Cohen[2]. Redes do tipo híbrida é dos tipos de arquiteturas encontradas nas redes *peer-to-peers*, nestas todos os participantes da rede podem se comunicar entre si, mas a organização da rede fica sob responsabilidade de um servidor. Uma das grandes vantagens do protocolo BitTorrent em relação a outros protocolos de distribuição de dados (ex. HTTP) é que quando vários usuários baixam um mesmo dado concorrentemente estes enviam partes deste dado uns aos outros tornando possível para o arquivo fonte ajudar um grande número de usuários com um pequeno aumento na sua taxa de envio.

O objetivo deste capítulo é discutir as principais características e o funcionamento do protocolo BitTorrent.

2.2 Elementos de uma rede BitTorrent

Para uma melhor compreensão de uma rede Bittorrent, faz-se necessário conhecer os termos usados no ambiente.

1. ***Peer* (par)**

[1] Optamos por usar os termos em ingles referentes ao funcionamento do BitTorrent por serem estes amplamente difundidos.

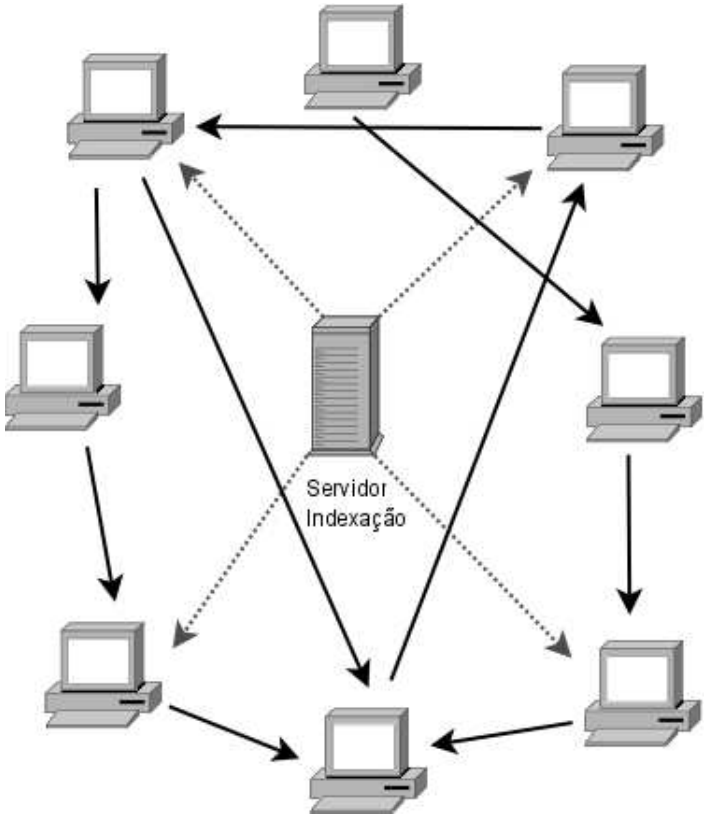


Figura 2.1: Funcionamento de uma rede *peer-to-peer* híbrida

Par é um cliente BitTorrent executado na estação local. Cada par é capaz de preparar, requisitar e transmitir qualquer tipo de arquivo através da rede usando o protocolo. Em uma rede BitTorrent, existem vários clientes (pares). Costuma-se usar o termo vizinho para o par que fica em uma estação remota quando comparado a um par em uma estação local. Portanto, um par pode ter vários vizinhos em uma rede BitTorrent.

2. ***Seeder* (semeador)**

   É o par que possui o arquivo completo a ser compartilhado.

3. ***Leecher* (sugador)**

   É o par que ainda não possui o arquivo completo, tem por objetivo baixar o que falta do arquivo até completá-lo. Além disso, ele deve ajudar os outros pares da rede enviando os pedaços do arquivo que já possui.

4. ***Swarm* (enxame)**

   Rede formada por pares que estão buscando o mesmo arquivo.

5. ***Tracker* (rastreador)**

Servidor responsável por manter uma lista dos pares nos enxames por ele administrado. Também armazena estatísticas sobre a rede. Cada par que entra em um enxame BitTorrent manda suas informações periodicamente para o Servidor Tracker.

6. **Arquivo metadado ".torrent" (metafile)**

   Arquivo texto com uma codificação especialmente criada para BitTorrent, onde são armazenadas informações como: arquivo(s) a serem baixado(s); URL dos Servidores Tracker; tamanho do arquivo a ser baixado.

7. ***Chunk*(pedaço)**

   O arquivo a ser compartilhado no enxame é partido em pedaços com tamanho pré-determinados. Esta técnica ajuda tanto na distribuição do mesmo como na otimização das taxas transmissão. No caso de erro na transmissão ou recepção do pedaço os pares precisam apenas requisitar o mesmo pedaço e não o arquivo inteiro. Um par pode baixar, ao mesmo tempo, vários pedaços de pares diferentes.

8. ***Block* (Bloco)**

   Cada pedaço é dividido em blocos com tamanhos pré-determinados. Isto ajuda tanto no envio quanto na obtenção do pedaço, pois podem ser baixados vários blocos de pedaços diferentes ao mesmo tempo. Quando um par começa a baixar blocos de um determinado pedaço este se fixa ao pedaço até baixar todos seu blocos.

2.3 Visão resumida do protocolo

Para se conectar em uma rede BitTorrent o usuário baixa o metadado ".torrent" do arquivo desejado de algum sítio especializado, abrindo-o em seu aplicativo BitTorrent. Neste momento, o aplicativo lê as informações do arquivo de metadados, descobre o endereço do Servidor *Tracker* e solicita a sua entrada na rede. Imediatamente o servidor envia uma lista de todos os usuários daquele *swarm* (referenciado no arquivo ".torrent").

Agora o par está pronto para trocar informações com seus possíveis vizinhos. À medida que o cliente baixa pedaços de vários pares, ele se torna apto a ajudar outros pares enviando os pedaços que já tem. Cada pedaço baixado é verificado. Caso a verificação falhe, o pedaço é descartado e é feita uma nova solicitação para o vizinho. Enquanto estiver no *swarm* o par mantém o Servidor *Tracker* informado sobre total de *download*, total de *upload*, total ainda a ser baixado, entre outros. Estas informações ajudam o *Tracker* a manter atualizadas a lista de pares participantes e estatísticas de uso daquele *swarm*.

Ao completar todos os pedaços de um arquivo o par se transforma em um *seeder* e passa a poder enviar todos os pedaços a quem solicitar.

2.4 Problemas do protocolo BitTorrent

Várias redes P2P são suscetíveis a ameaças e ataques. Alguns destes ataques são relacionados a vulnerabilidades encontradas nestes sistemas. Em alguns casos, ameaças e ataques podem levar limitações na performance e até mesmo devastar um sistema P2P. As redes BitTorrent não são diferentes. Muitos problemas já foram documentados, entre esses problemas, é possível citar:

1. **Falta de Anonimato**

   O protocolo BitTorrent não oferece anonimato, ou seja, fácil descobrir o IP dos pares de um *swarm*;

2. **O problema de Leech**

   O protocolo BitTorrent oferece poucos incentivos a usuários *seeders* depois que terminam seus *downloads*. Isto pode provocar a diminuição da rede BitTorrent para um determinado torrent, ou seja, arquivos torrent antigos tendem a desaparecer.

3. ***Freeriders***

   Freeriders são usuários que se beneficiam baixando arquivos sem oferecer contribuição em troca.

2.5 Visão detalhada do protocolo

No BitTorrent, cada um de seus elementos tem funções claras e específicas. O processo se inicia com a distribuição do arquivo metadado (arquivo ".torrent"). Os interessados baixam então este arquivo em seus clientes BitTorrent para participar do *swarm*. Em seguida, o usuário troca pedaços com outros pares até completar o arquivo. A seguir, são apresentados com mais detalhes as várias atividades executadas pelo protocolo.

2.5.1 Arquivo metadado (".torrent")

Quando o usuário deseja compartilhar alguma informação, ele cria um arquivo de metadado chamado torrent. Este arquivo usa uma codificação conhecida como *bencoding* [6] que suporta quatro tipos de dados: *strings* (cadeias de caracteres), números inteiros, listas (strings e inteiros) e dicionários (listas *strings*). As listas e dicionário podem conter qualquer um dos quatro tipos. Abaixo segue a especificação dos dados.

1. **Bytes Strings**

   Bytes *strings* são representados da seguinte forma: <tamanho da string>:`<string>`.

   Exemplo: 4:spam ->representa uma *string* com conteúdo SPAM.

2. **Integers**

   Os valores inteiros são representados da seguinte forma: **i**<inteiro>**e**. São iniciados com letra “i” e finalizados com a letra “e”.

   Exemplo: **i**3**e** ->que representa o número inteiro três.

3. **Listas**

   As lista são representadas da seguinte forma: **l**<lista de strings>**e**. São iniciadas com a letra “l” e finalizadas com a letra “e”.

   Exemplo: **l**4:casa3:mae**e** ->que representa uma lista de duas *strings*: “casa” e “mae”.

4. **Dicionário**

   Dicionários possuem definições e são representados por : **d**<string><elemento>**e**. Eles são iniciados com a letra “d” e finalizados com a letra “e” .

   Exemplo: O termo **d**4:casa4:moro**e**, que representa um dicionário onde casa = moro.

   O conteúdo de um arquivo metainfo (o arquivo com a terminação “. Torrent”) é uma “bencoding” dicionário, que contém as chaves listadas abaixo.

   (a) **Info** - uma string tipo dicionário com diversas informações sobre aquele torrent, por exemplo: os pedaços do arquivo, tamanho de cada pedaço, nome do arquivo, etc. Existem duas formas possíveis para chave “info”: “single-file” torrent sem qualquer estrutura de diretório, apenas um arquivo e “multi-file” torrent com uma estrutura de diretórios e mais de um arquivo a ser compartilhado.

   (b) **Announce** - especifica a URL do *tracker* (*string*).

   (c) **Announce-list** (opcional) - é uma extensão para a especificação oficial. Esta chave é utilizada para implementar listas de backup de *trackers*.

   (d) **creation date** (opcional) - hora da criação do torrent, no formato hora UNIX . (Inteiro -Segundos desde 1/1/1970).

   (e) **comment** (opcional) - comentários do autor (*string*).

   (f) **create by** (opcional) - nome e versão do programa utilizado para criar o .Torrent (*string*).

Na figura 2.2 é mostrado o conteúdo de uma arquivo Torrent. Este arquivo contém todas as informações para um par ingressar em um *swarm* BitTorrent como discutido acima.

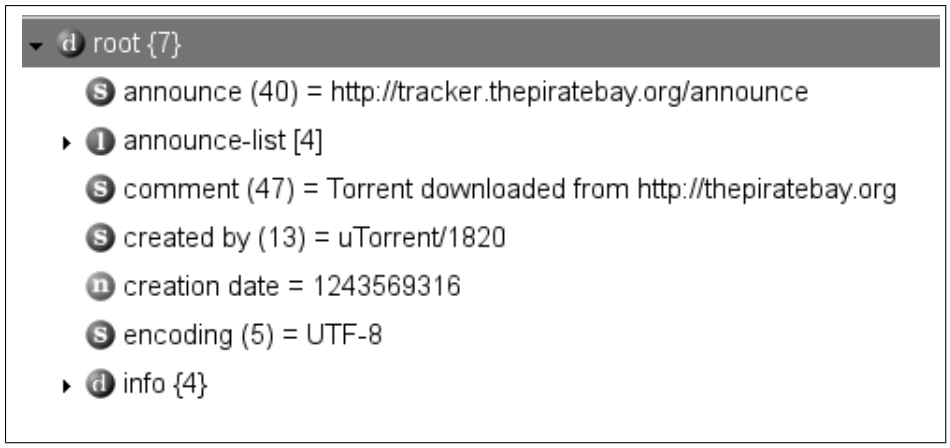


Figura 2.2: Conteúdo do arquivo torrent com seus respectivos campos

2.5.2 Detalhes da chave "info"

A chave "info" é uma *beencoding* dicionário que trabalha em dois modos: "single-file" e "multi-file". Na criação de um arquivo Torrent o dado que será compartilhado poderá ser único, no caso "single file", ou vários arquivos, no caso "multi-file".

Abaixo os campos comuns para ambos os modos (*Single* e *Multi*):

1. **piece length** - número de bytes de cada pedaço (inteiro).
2. **pieces** - string com a concatenação de todos os *hash*[2] de 20 bytes SHA-1[3], um por pedaço (byte string).

Chave Info no modo "Single File" (Figura 2.3)

Neste caso existe a seguinte estrutura:

1. **Name** - nome do arquivo, meramente para consulta (*string*).
2. **Length** - comprimento do arquivo em bytes (inteiro).
3. **Md5sum** (opcional) - uma seqüência de 32 caracteres hexadecimais correspondentes à soma MD5[4] do arquivo. Isto não é utilizado pelo BitTorrent em tudo, mas é incluído por alguns programas para maior compatibilidade.

Na figura 2.3 é mostrado o conteúdo de um arquivo Torrent do tipo "Single-File" com as seguintes informações: tamanho do arquivo igual a 33MB, tamanho do pedaço igual 64KB e a representação *hash* de todos os pedaços concatenados.

Chave Info no modo "Multiple File" (figura 2.4)

Neste caso existe a seguinte estrutura:

[2]É a transformação de uma grande quantidade de informações em uma pequena quantidade de informações utilizando alguns algoritmos como: MD5, SHA-1.

[3]Sucessor do MD5, também usado para gerar assinaturas criptografadas de arquivos quaisquer.

[4]algoritmo que gera uma "assinatura" de um arquivo qualquer.

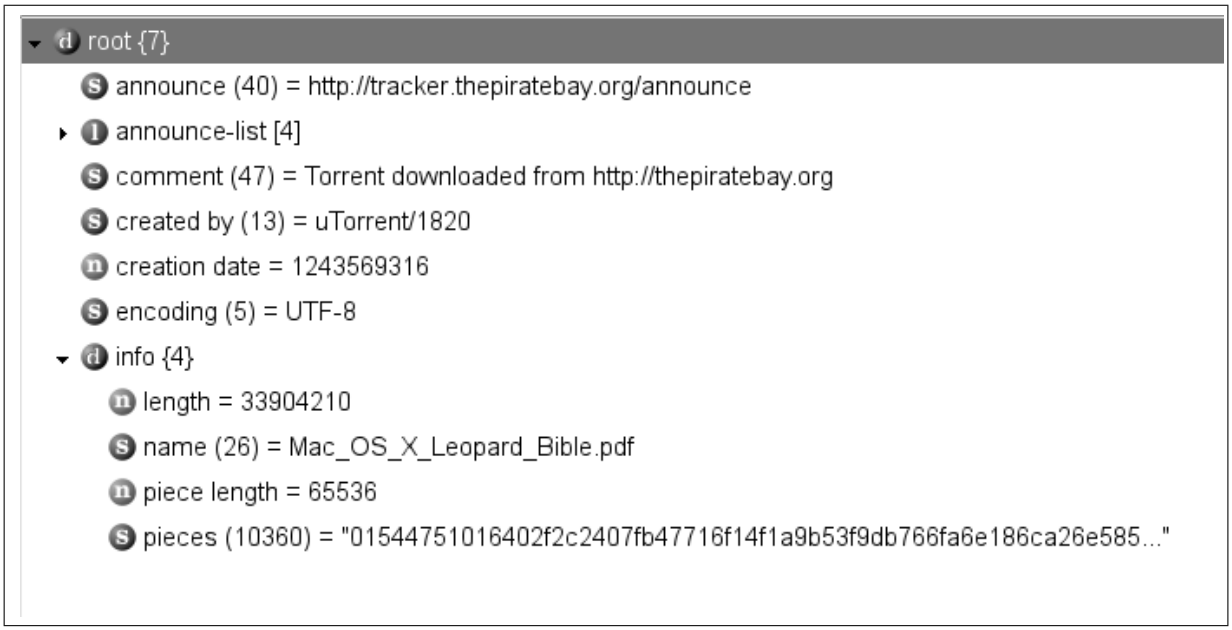


Figura 2.3: Conteúdo de um Torrent com destaque ao campo “info” - SingleFile

- **Name** - nome do diretório para armazenar todos os arquivos (*string*).
- **Files** - uma lista dicionário, um para cada arquivo. Cada dicionário nesta lista contém as seguintes chaves:
  - **Length** - comprimento do arquivo em bytes;
  - **md5sum** (opcional) - uma seqüência de 32 caracteres hexadecimais correspondentes à soma MD5 de cada arquivo;
  - **path** - uma lista contendo um ou mais elementos que juntos representam string, o caminho e o nome do arquivo. Exemplo, um arquivo “dir1/dir2/file.ext” codificada em lista - **l**4:dir14:dir28:file.ext**e**.

Na figura 2.4 é mostrado o conteúdo de um arquivo Torrent do tipo “Multi-File” com as seguintes informações: a quantidade de arquivos a serem compartilhados é igual a 15, o tamanho de um dos arquivos, no caso é de 38MB e o nome de um dos arquivos.

2.6 Criando e publicando torrents

Os clientes BitTorrent que desejam compartilhar arquivos, devem primeiramente gerar o “.torrent” deste arquivo. Este processo quebra o arquivo em pedaços que variam de 64KB a 8MB. Na criação, os softwares geram um *checksum* de cada pedaço, utilizando o algoritmo SHA-1, depois gravam este no arquivo “.torrent” com informações como : nome do torrent, endereço URL do servidor *Tracker*, tamanho do pedaço, etc.

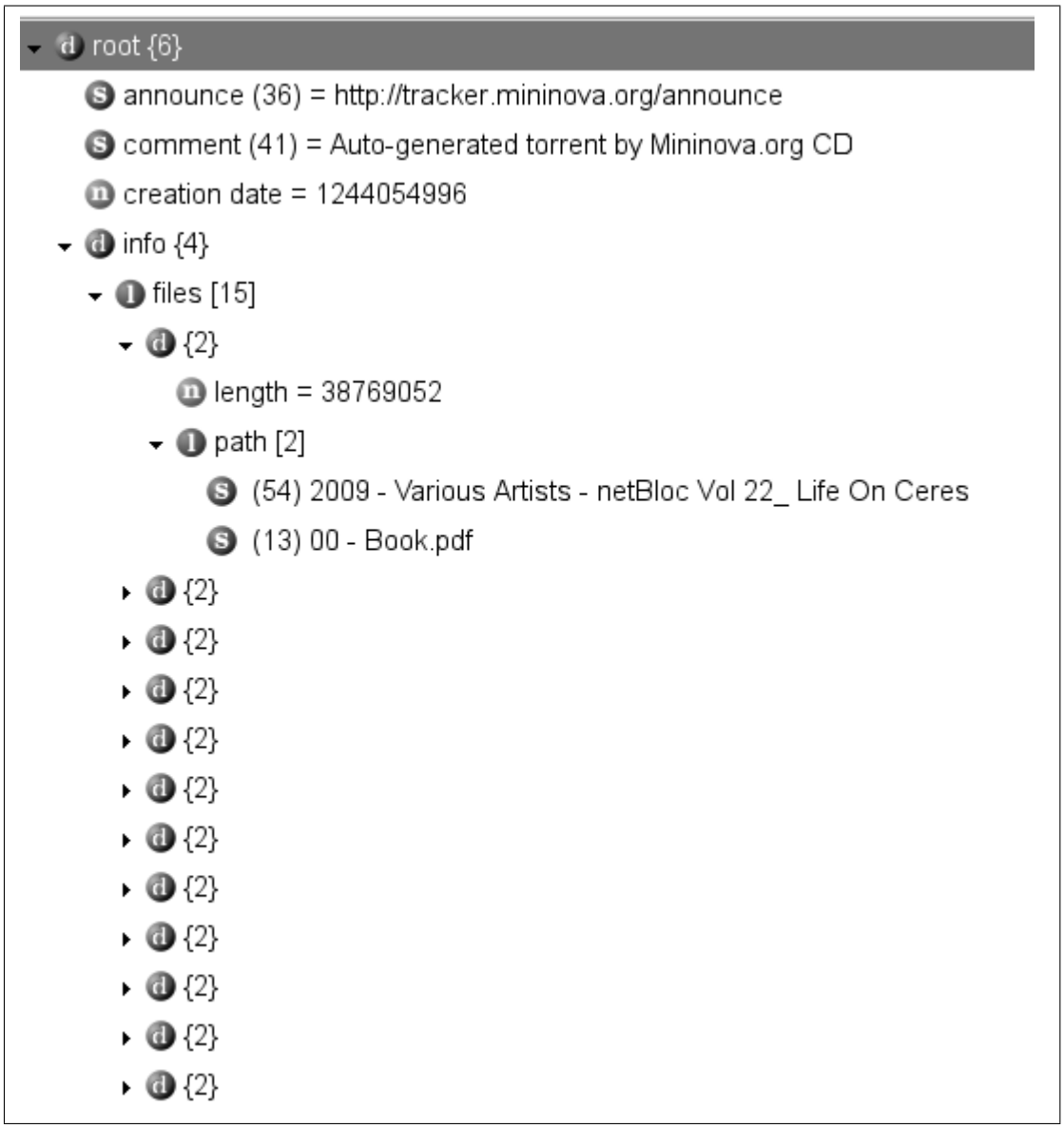


Figura 2.4: Conteúdo de um Torrent com destaque ao campo “info” - MultiFile

Para publicação, o cliente coloca o arquivo ".torrent" em um ou mais servidores de torrents na Internet. Geralmente, esses servidores são sítios com aparência de catálogos, onde se pode criar contas e fazer pesquisas por ".torrent". Estes servidores também podem ser servidores *Tracker*.

2.7 Servidor Rastreador (*Tracker*)

O *Tracker* é responsável por organizar os *swarms* por ele administrados. Cada arquivo metadado (arquivo ".torrent") formará um *swarm* e por isso possui como endereço a URL do servidor *Tracker*. O par ao abrir o arquivo metadado entra em contato com o servidor *Tracker*, se apresentando (anúncio). O *Tracker* é um serviço HTTP ou HTTPs que responde a requisições do tipo HTTP GET. As requisições incluem métricas que o ajudam a manter estatísticas do torrent, enquanto as respostas do *Tracker* incluem a lista de pares no *swarm*.

As requisições dos pares aos servidores *Trackers* são enviadas via URL usando métodos do tipo CGI, isto é, a URL seguida de "?" e depois os campos e valores (`parametro=valor`) separados por "&".

```
GET /announce?peer_id=-KT2210-534566901978&port=6881&uploaded=0
&downloaded=0&left=714408232&compact=1&numwant=100&key=1317096065
&event=started
&info_hash=%85%14%85n%b1%d7%b0d%14%25%a8cPY%d6%09%84%3c%a2%b9
```

Figura 2.5: Requisição Cliente ao Servidor *Tracker*

A figura 2.5 mostra um exemplo de requisição de um par ao servidor *Tracker*, onde se pode notar alguns parâmetros como: *peer_id*; identificação do cliente BitTorrent; *uploaded*, tudo que o cliente já enviou (bytes); *downloaded*, tudo que o cliente já recebeu; *left*, o que falta para completar (bytes) o arquivo e *info_hash*, a identificação do *swarm*. Note que na URL dados binários, como *info_hash*, são devidamente encapsulados. Isto quer dizer que qualquer byte não ASCII deve ser codificado usando o formato "%nn", onde "nn" é o valor hexadecimal do byte representado. Maiores detalhes são encontrados na RFC 1738 – URL.

2.7.1 Requisições Cliente para Rastreador(*Tracker*)

As requisições HTTP GET (figura 2.5) que os clientes BitTorrent solicitam ao *Tracker* são passadas via URL onde seus parametros são separados por "&". Os parâmetros mais comuns das requisições são:

1. **info_hash** - Sequência de caracteres (*hash*), do tipo SHA-1, que representa o valor da chave info.

2. **peer_id** - identificação única para o par. É gerado quando o par entra no *swarm*.
3. **port** - o número da porta que o par está ouvindo;
4. **uploaded** - quantidade total de dados enviados (bytes).
5. **downloaded** - quantidade total de dados baixados (bytes).
6. **left** - O numero de bytes que o par ainda precisa baixar.
7. **no_peer_id** - indica que o tracker pode omitir o campo peer_id.
8. **event** - se especificado deve ser uma das opções abaixo:
   (a) **started** - a primeira requisição ao tracker deve incluir a chave *event* com este valor.
   (b) **stopped** - deve ser enviada se o par está saindo do *swarm* sem erros.
   (c) **completed** - deve ser enviada quando o *download* é terminado.
9. **Ip** (opcional) - o endereço IP do par. Suporta Ipv4 e Ipv6.
10. **Numwat** (opcional) - número de vizinhos que o par gostaria de receber do tracker. Pode ser 0 (Zero).
11. **Key** (optional) - uma identificação adicional que não é compartilhada por nenhum usuário. É utilizada para permitir ao par provar sua identidade se seu endereço se seu IP mudar.
12. **Trackerid** (optional) - Se no anúncio contiver a identificação do *Tracker*, ela deverá ser configurada aqui.

2.7.2 Respostas *Tracker* para cliente

Após o par ter se anunciado ao servidor *Tracker*, o mesmo retorna com os parâmetros especificados abaixo:

1. **Failure reason** - se presente, então nenhuma outra chave deverá estar presente.
2. **Warning message** (optional) - similar a *failure reason*, mas a resposta ainda é processada normalmente.
3. **interval** - intervalo mínimo entre duas requisições para o *tracker*;
4. **Min interval** (optional) - intervalo mínimo entre os anúncios do par ao servidor *tracker*.

5. **Tracker id** - Uma *STRING* que o par deve mandar de volta nos seus próximos anúncios.

6. **Peers** (tipo dicionário) - aqui é uma lista tipo dicionário com os pares do *swarm*. Contém os seguintes parâmetros:

   (a) **peer id** - identificação do par. Este é único.

   (b) **ip** - IP ou nome DNS do par.

   (c) **port** - Porta TCP/UDP do par

7. **Peers** (Tipo binário) - ao invés de usar uma lista dicionário descrita acima, a lista de pares pode ser representada por "string" multiplo de seis bytes. Os primeiros quatro bytes são o IP do par e os dois últimos são a porta.

Por padrão, a lista dos pares tem cinquenta vizinhos. Então, se existem poucos vizinhos no *swarm*, a lista será pequena. Se existem mais de cinquenta vizinhos no *swarm*, o *Tracker* se encarregará de escolher randomicamente os vizinhos para compor essa lista.

Outro tipo de mensagem de resposta que o *Tracker* pode usar é conhecida como "Scrape do *Tracker*". Ela é usada para que os pares façam consultas sobre determinados *swarms* sem precisar entrar no sistema. Estas consultas seguem via URL passando como parâmetro o "info_hash" do torrent. Existem diversos programas para "Scrape do *Tracker*" que ajudam a decifrar as informações desse tipo de mensagem. A figura 2.6 mostra um exemplo de mensagem "Scrape do *Tracker*" feita por um par a um servidor *Tracker*.

```
GET /scrape?info_hash=%ce%2f%22%c9%02%78%5c%3a%b9%5d%fa%40%d4%37%b8%d3%d2%90%60%3d
HTTP/1.1
```

Figura 2.6: Mensagem "Scrape" do Cliente para o Servidor *Tracker*

No momento que o servidor *Tracker* recebe uma consulta "Scrape", ele irá responder com informações sobre o *swarm* como: quantidade de *seeders* e quantidade de *leecher* no *swarm*, tamanho do arquivo compartilhado, etc. A figura 2.7 mostra um exemplo deste tipo de mensagem. São campos que existem na mensagem "Scrape":

1. **complete** - número de pares com o arquivo inteiro;

2. **downloaded** - número total de vezes que o *tracker* registrou um *download* bem sucedido;

3. **incomplete** - número de vizinhos que não são *seeders*;

4. **name** (opcional) - o nome interno do torrent, especificado como "name" na secão de informacão do arquivo .torrent

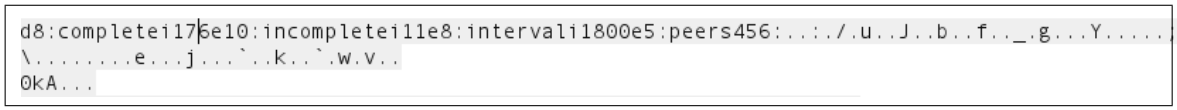


Figura 2.7: Mensagem resposta “Srape do *Tracker*” para o cliente

2.8 Algoritmos

Após a descoberta dos *seeders* e dos leechers, faz-se necessário escolher de quais pares baixar e para quais pares enviar. Como o BitTorrent não possui uma central de alocação de recursos, cada par é responsável por maximizar sua própria taxa de *download*. Cada par alcança este objetivo fazendo *download* do maior número de pares possível e decidindo para qual par fazer *upload*.

Existem duas classes de algoritmos no BitTorrent: Algoritmos de Seleção de Pares e Algoritmos de Seleção de Pedaços. A seguir são apresentadas as características desses algoritmos.

2.8.1 Algoritmos de seleção de pares

Neste tópico é descrito como o par BitTorrent escolhe seus pares para baixar e/ou enviar arquivos. Esta escolha é baseada em uma estratégica conhecida como “tit-for-tat”. O “tit-for-tat” é usado com muita eficácia na teoria de jogos e foi introduzido pela primeira vez por Robert Axelrod’s[13] em dois torneios em 1980. A estratégia se vale da cooperação ou não entre dois jogadores. Se um jogador cooperar, seu adversário irá responder à ação cooperando. Se um jogador não cooperar, então o outro jogador retalia. No BitTorrent, esta estratégia foi implementada da seguinte forma:

1. **Choke** - um par BitTorrent pode afogar seu vizinho, no momento que este não coopera com ele, interrompendo todos os *uploads* a este;

2. **Unchoke** - caso o vizinho comece a cooperar com o par BitTorrent, ele recebe um desafogamento, a partir deste momento este vizinho pode receber dados do par;

3. **Optimistic unchoke** - como o par afoga seus vizinhos que não cooperam com ele, este par pode ter depois de um tempo muitos vizinhos afogados (*choke*). Por isso, a cada 30 segundos um dos vizinhos é desafogado (*unchoke*) aleatoriamente.

Resumindo, para cooperar, pares enviam pedaços. Se o par não cooperar é afogado (*choke*) pelos outros.

2.8.2 Algoritmos de seleção de pedaços

Cada arquivo, referenciado no Torrent, ao ser baixado é dividido em pedaços, que podem ser obtidos simultaneamente de vários vizinhos pelo par. Os pares BitTorrent incorporam

várias políticas para seleção de pedaços. Todas visam otimizar a troca de pedaços entre os pares, pois uma boa ordem de seleção é importante para uma boa performance. Ao se conectar a um *swarm*, os pares BitTorrent iniciam as conexões entre si para troca de pedaços, que pode ser resumida da seguinte forma:

1. **Prioridade Estrita**

   Para melhorar ainda mais a performance, o par BitTorrent quebra o pedaço em subpedaços, conhecidos como blocos. Uma vez que o par solicitou um bloco, ele espera pelos outros blocos que compõem o pedaço solicitado, antes de requisitar outro pedaço daquele vizinho. Embora este par possa baixar vários pedaços simultaneamente de vizinhos diferentes.

2. **Mais raro primeiro**

   Ao selecionar o próximo pedaço a ser baixado, os pares escolhem os pedaços que seus vizinhos possuem em menor quantidade. Também se certificam que os pedaços mais comuns são deixados por último. No caso do *swarm* com um único *seeder* (apenas um par com o arquivo inteiro), os pares participantes procuram baixar pedaços diferentes entre si.

3. **Primeiro pedaço aleatório**

   Se nenhum par possui pedaços para fazer *upload*, então é importante pegar um pedaço o mais rápido possível. Além disso, é também importante que os pares possuam pedaços diferentes para iniciar as trocas entre si. Por essa razão, os pedaços são selecionados aleatoriamente.

4. **Modo "EndGame"**

   Faltando poucos pedaços para o par formar o arquivo completo desejado, este envia requisições para todo o *swarm*, solicitando que lhe seja enviado o pedaço que falta. Neste momento, o par cancela todos os *upload* que por ventura esteja fazendo. O objetivo é garantir sua taxa máxima para *download* dos últimos pedaços.

2.9 Visão do BitTorrent sob ótica do protocolo TCP

O protocolo BitTorrent é um protocolo de aplicação que roda em cima do modelo TCP/IP. Abaixo segue uma visão detalhada de como o algoritimo do BitTorrent interage com a camada de transporte.

Um par BitTorrent mantém informações de estado de cada conexão com seus pares. Estas informações antecedem as trocas de dados entre par e vizinhos e podem ser:

1. **choked**

Quando um par afoga um vizinho, é um aviso de que os pedidos não serão respondidos até que este seja desafogado no par. Os pedidos pendentes dos vizinhos afogados (*choked*) são descartados.

2. **interested**

   Quando um par está interessado em algo que esse vizinho tem para oferecer, ou vice-versa. Esta é uma notificação do par que está requisitando pedaços quando o mesmo foi *unchoked* por algum vizinho.

Note que isto implica que o par deve manter uma lista de quem ele está interesado, quem está interessado nele e quem está afogado e que não está afogado. Esta lista é representada da seguinte forma:

1. **am_choking** - o par foi afogado por um vizinho;
2. **am_interested** - o par está interessado por um vizinho;
3. **peer_choking** - o vizinho foi afogado por este par;
4. **peer_interested** - um vizinho está interessado por este par.

Portanto, cada conexão do par com seus vizinhos iniciam em *choked* e *not interested*. Em outras palavras:

1. **Am_choking = 1** - o par está afogado no vizinho;
2. **Am_interested = 0** - par não está interessado no vizinho;
3. **Peer_choking = 1** - o vizinho está afogado no par;
4. **Peer_interested = 0** - o vizinho não está interessado no par.

Em resumo, ao entrarem no *swarm*, pares e vizinhos estão afogados um em relação ao outro e ninguém está interessado em ninguém.

A figura 2.8 mostra os estados dos pares ao entrarem no *swarm*. Pedaços são baixados pelo par quando este está interessado por um vizinho que não deve está afogado no par.

É importante para o par manter os vizinhos informados se está interessado por eles ou não. Isto permitirá aos vizinhos saberem se o par vai começar baixar pedaços quando estiverem desafogados (*unchoked*) e vice-versa.

A figura 2.9 mostra o par trocando mensagens com seus vizinhos a fim de receber e enviar pedaços. Os pedaços são enviados por um par quando este não está afogado em um vizinho, e este par está interessado por este vizinho.

A comunicação inicial entre o par e seus pares é um pacote conhecido como “handshake”. Depois deste começam as trocas de mensagens descritas acimas, montando toda a estrutura para o funcionamento dos algoritmos “Seleção de pares” e “Seleção de pedaços” .

2.9.1 Handshakes

Handshake é uma mensagem obrigatória entre um par e um vizinho. Na sua estrutura, esta mensagem possui os seguintes campos:

1. **pstrlen** - comprimento da *string* `<pstr>`;
2. **pstr** - *string* que identifica o protocolo;
3. **reserved** - oito bytes são reservados para uso o futuro. A maioria dos pares BitTorrent usam tudo zero;
4. **info_hash** - *hash* da chave "info" do arquivo torrent. Este *info_hash* é o mesmo que o par transmite nas requisições para o *Tracker*;
5. **peer_id** - *string* que identifica o par BitTorrent ou par na conexão. Este é o mesmo transmitido para o servidor *Tracker* para identificar este em um *swarm*.

Em relação a troca de mensagens de *handshakes*, é importante enfatizar:

- O par transmissor é responsável por enviar *handshakes* imediatamente ao outro par. Já o receptor aguarda por estes *handshake*;
- Os *swarms* são identificados pela chave *info_hash* trocadas nos *handshakes*;
- Caso um par receba a chave *info_hash*, trocadas no *handshake*, que o não tenha, esta conexão é cancelada.
- Caso um par receba a chave *peer_id*, trocadas no *handshake*, que não tenha seu valor esperado, a conexão é cancelada.

Importante destacar, que as chaves *info_hash* (identificadora do *swarm*) e a *peer_id* (identificadora de par) já são conhecidas pelo servidor *Tracker*, pois todos os pares se anunciam ao ingressar no *swarm*. Com isso os pares BitTorrent esperam valores conhecidos para estas chaves.

2.9.2 Mensagens

Todas as mensagens do protocolo é composta de um campo para tamanho, um identificador (*ID*) e dados (*payload*).

Seguem abaixo as principais mensagens:

1. **keep-alive:** `<len=0000>`

   A mensagem "keep-alive" tem tamanho 0 (zero) e não tem identificador nem dados. Ela é usada para manter conexões entre pares ativa.

2. **choke:** `<len=0001><id=0>`

   Mensagem “choke” tem tamanho fixo e não tem dados. Usada para afogar pares.

3. **unchoke:** `<len=0001><id=1>`

   Mensagem “unchoke” tem tamanho fixo e não tem dados. Usada para desafogar pares.

4. **interested:** `<len=0001><id=2>`

   Mensagem “interested” tem tamanho fixo e não tem dados. Usada para mostrar interesse a um par.

5. **not interested:** `<len=0001><id=3>`

   Mensagem “not interested” tem tamanho fixo e não tem dados. Usada para mostrar que não tem interesse por um par.

6. **have:** `<len=0005><id=4><piece_index>`

   Mensagem “have” tem um tamanho fixo. No campo dados, vem o índice do pedaço que já foi baixado e verificado.

7. **bitfield:** `<len=0001+X><id=5><bitfield>`

   Mensagem de tamanho variável, que deve ser enviada imediatamente após os “handshakes” e antes de qualquer outra mensagem. Os dados representam os pedaços que foram baixados corretamente.

8. **request:** `<len=0013><id=6><index><begin><length>`

   Mensagem de tamanho fixo usada para requisitar pedaços dos pares. O campo dados tem três parâmetros: “index”, “begin” e “length”.

9. **piece:** `<len=0009+X><id=7><index><begin><block>`

   Mensagem de tamanho variável, onde o X é o tamanho do bloco. O campo dados tem três parâmetros: “index”, “begin” e “block”, onde “index” representa o índice do pedaço e o “block” representa o bloco, uma porção do pedaço.

10. **cancel:** `<len=0013><id<=8><index><begin><length>`

    Mensagem de tamanho fixo, usada para cancelar requisições de pedaços nos pares.

11. **port:** `<len=0003><id=9><listen-port>`

    Mensagem enviadas nas novar versões do protocolo que implementam *Trackers* com DHT.

Nos “handshakes” e nas mensagens é onde acontecem as troca de informações entre par e os vizinhos no *swarm*.

O estudo apresentado neste capítulo foi essencial para o desenvolvimento de ferramentas de análise dos *traces* coletados nas redes BitTorrent e discutidos no próximo capítulo.

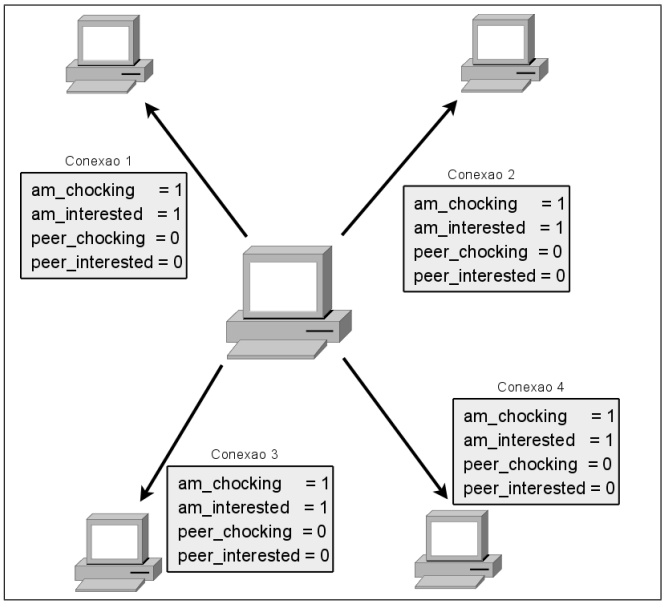


Figura 2.8: Pares ao entrarem no *swarm*

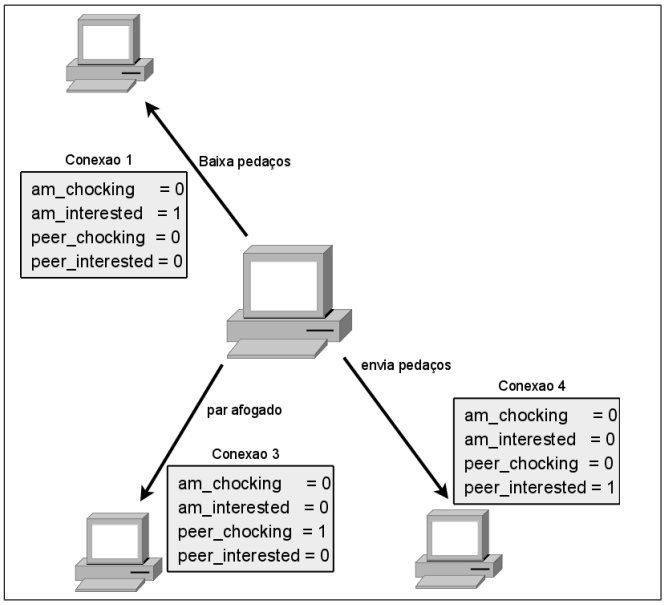


Figura 2.9: Cliente trocando informações e pedaços com seus pares

3 Ambiente de testes

3.1 Introdução

Este capítulo tem dois objetivos: caracterizar o tráfego de uma rede BitTorrent e apresentar a simulação em NS-2 feita a partir desta caracterização. Para caracterizar o tráfego BitTorrent, foi montado um ambiente real na UNIRIO onde dados foram coletados durante três meses. Para simular uma rede BitTorrent, foi implementado um *patch* na ferramenta NS-2.

3.2 Ambiente real

Visando obter dados reais de ambiente BitTorrent para caracterização do tráfego foi montado um ambiente real com os seguintes componentes:

- Servidor Rastreador (*tracker*);
- Clientes BitTorrent Semeadores (*seeder*);
- Clientes BitTorrent Sugadores (*leecher*);
- Arquivos a serem compartilhados.

A partir do ambiente montado, passou-se a monitorar a entrada de pares, a transmissão de pedaços dos arquivos e as saídas dos pares do sistema. De posse destes dados, verificou-se a existência de *freeriders* no experimento. Segundo [20], *freeriders* podem ser classificados como:

- **Trapaceiros** - Usuários que modificam o código do cliente BitTorrent para conseguir taxa 0 KBps de *upload*;
- **Estratégicos** - Usuários que alteram a taxa de *upload* para valores pequenos e constantes;
- **Sortudos** - Usuários que conseguem baixar pedaços em um *swarm* sem nenhuma ou pouca contribuição para o sistema. Estes pares apenas dão sorte em determinados momentos, pois não recebem nenhuma solicitação de pedaços dos seus vizinhos.

Neste estudo, optou-se por trabalhar com os *freeriders* conhecidos como trapaceiros e estratégicos, onde suas taxas de envio variam de zero a valores bem pequenos e cujas taxas de recebimento são ilimitadas. Muitos trabalhos usam taxa do *freerider* de *upload* entre 0 KBps a 4 KBps como em [7], [9] e [14], apesar de nenhum deles explicar exatamente o porquê desses valores. Neste trabalho optou-se a definição de *freerider* adotada em [20] onde taxas de *upload* é de até 2 KBps. Isto se deve ao fato do uso de internet no Brasil variar de um modem comum em uma conexão discada com velocidade de *upload* a partir de 7KBps até um modem DSL em uma conexão banda larga chegando a taxas superiores a 1MBps. Portanto, uma taxa inferior a 2KBps normalmente representa a intenção de economia proposital da banda de *upload*.

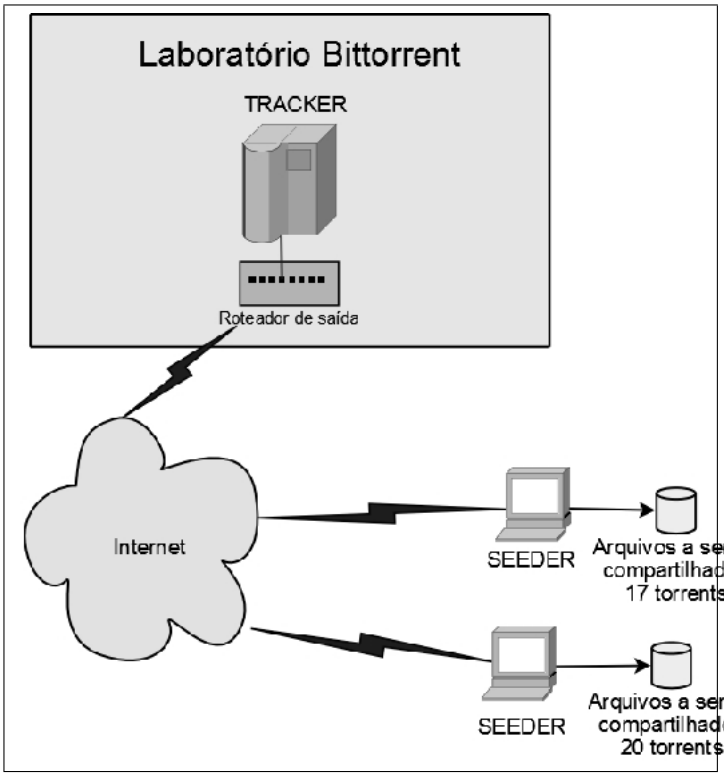


Figura 3.1: Experimento de rede BitTorrent

A figura 3.1 apresenta a infraestrutura do laboratório montado na UNIRIO para coleta de dados sobre o comportamento BitTorrent. A seguir são discutidos cada um dos componentes da rede BitTorrent usados nesse experimento.

3.2.1 Servidor *tracker*

O Servidor *tracker* é responsável por organizar os *swarms* a ele atribuídos. Para esta tarefa foi montada uma máquina com as seguintes características: processador DualCore da Intel de 2GHz, 1GB de memória, disco rígido de 250 GB de capacidade, IP fixo e público,

sistema operacional Linux OpenSuse 10.3[1] e instalação de programas para coleta, armazenamento e análise de dados. Foram criados e armazenados trinta e sete *swarms* nesse *tracker*. Cada *swarm* representa um arquivo a ser compartilhado e que se encontra armazenado em dois clientes BitTorrent fora da rede do laboratório (figura 3.1). O servidor é também responsável por armazenar os *logs* de cada *swarm*.

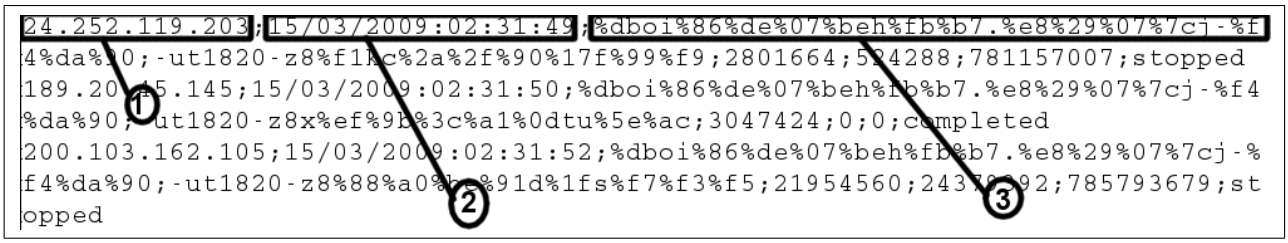


Figura 3.2: Arquivo de *LOG* do software do *tracker*

A figura 3.2 apresenta uma amostra dos *logs* armazenados pelo *tracker*. Neste exemplo, é interessante destacar três campos numerados na figura:

- Campo 1 - IP do par que no momento se comunica com o *tracker*;
- Campo 2 - Data e hora que o par se comunica com o *tracker*;
- Campo 3 - Identificação do *swarm* que o par está inserido.

Implementação do software do *tracker*

O *software* escolhido para implementação do *tracker* foi o BNBT Easytracker versão 7.7 Beta[2], pois é implementado em linguagem C e tem código fonte para dois sistemas operacionais: GNU Linux e Microsoft Windows. A principal razão da escolha foi a facilidade de instalação, sendo necessário apenas compilá-lo antes do uso. Além disso, ele também serve como mecanismo de busca para torrents.

3.2.2 Cliente BitTorrent *seeder*

O *seeder* armazena os arquivos que são compartilhados em cada *swarm*. Ele é responsável por enviar, quando requisitado, os pedaços iniciais dos arquivos para os pares vizinhos no *swarm*. No sistema BitTorrent é necessário a existência de pelo menos um *seeder*.

Os *seeders* foram instaladas em duas estações de trabalho com sistema operacional Linux Opensuse 10.3 e com sistema operacional Debian Lenny[3], ambas usando cliente BitTorrent Ktorrent[4]. As duas estações foram colocadas fisicamente em locais e redes

[1] http://pt.opensuse.org/
[2] http://bnbteasytracker.sourceforge.net/
[3] http://www.debian.org/
[4] http://ktorrent.org/

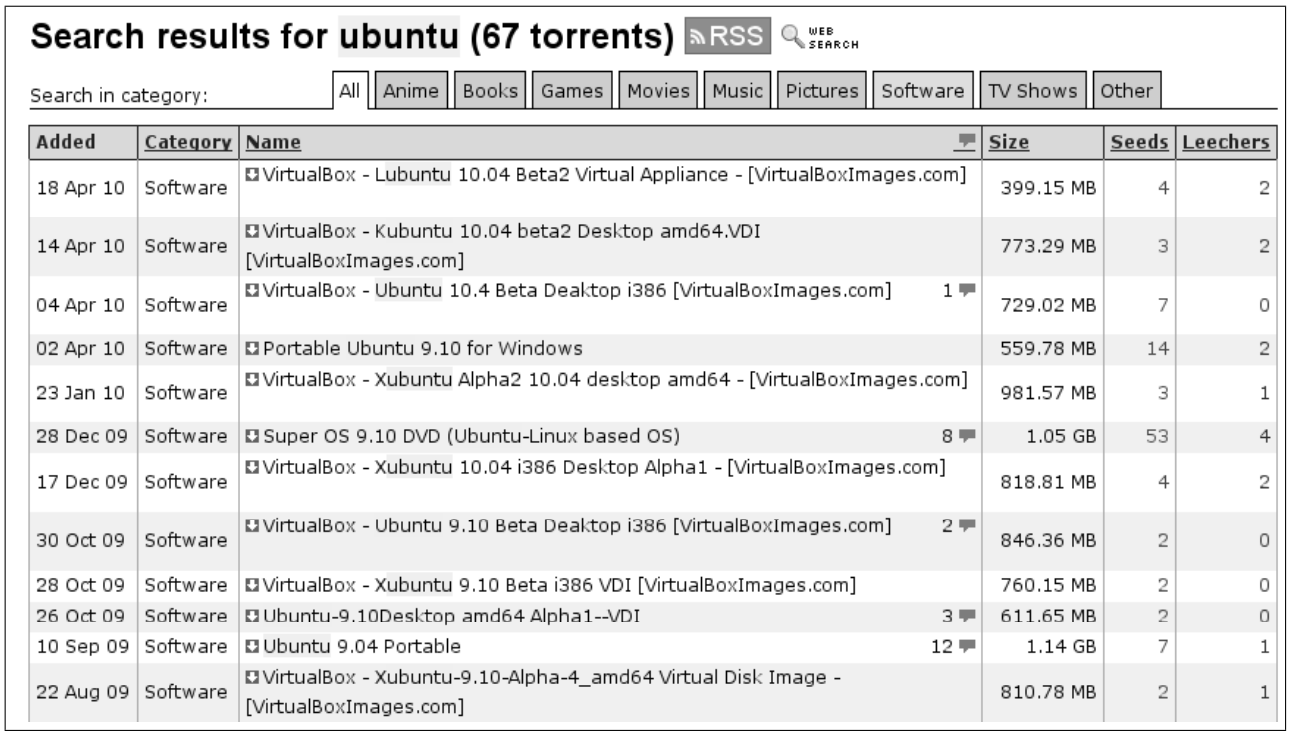
diferentes: a primeira foi instalada em uma rede ADSL da operadora OI[5] e a segunda foi instalada em uma rede de CABO da operadora Virtua[6].

3.2.3 Cliente BitTorrent *leecher*

O *leecher* é uma estação de trabalho simples com a função inicial apenas de entrar no *swarm* para baixar o torrent desejado. O *leecher* foi instalado com sistema operacional Linux Opensuse 10.3 com cliente BitTorrent Ktorrent em uma rede ADSL da operadora OI. Este é um componente implementado artificialmente na rede BitTorrent estudado com o objetivo apenas de entender os *logs* gerados por um *leecher*.

3.2.4 Arquivos compartilhados

Foram gravados nos dois *seeders* trinta e sete arquivos com tamanho médio de 700MB. O conteúdo dos arquivos são diversos como, por exemplo, conjuntos de ferramentas, atualizações de softwares, sistemas operacionais, etc. Todos esses arquivos possuem licenças de domínio público BSD-like e/ou GPL. Para cada arquivo a ser compartilhado foi criado um *torrent* disponibilizado nos sítios Mininova e ThePirateBay.



Search results for ubuntu (67 torrents) RSS WEB SEARCH

Search in category: All | Anime | Books | Games | Movies | Music | Pictures | Software | TV Shows | Other

Added	Category	Name	Size	Seeds	Leechers
18 Apr 10	Software	VirtualBox - Lubuntu 10.04 Beta2 Virtual Appliance - [VirtualBoxImages.com]	399.15 MB	4	2
14 Apr 10	Software	VirtualBox - Kubuntu 10.04 beta2 Desktop amd64.VDI [VirtualBoxImages.com]	773.29 MB	3	2
04 Apr 10	Software	VirtualBox - Ubuntu 10.4 Beta Deaktop i386 [VirtualBoxImages.com] 1	729.02 MB	7	0
02 Apr 10	Software	Portable Ubuntu 9.10 for Windows	559.78 MB	14	2
23 Jan 10	Software	VirtualBox - Xubuntu Alpha2 10.04 desktop amd64 - [VirtualBoxImages.com]	981.57 MB	3	1
28 Dec 09	Software	Super OS 9.10 DVD (Ubuntu-Linux based OS) 8	1.05 GB	53	4
17 Dec 09	Software	VirtualBox - Xubuntu 10.04 i386 Desktop Alpha1 - [VirtualBoxImages.com]	818.81 MB	4	2
30 Oct 09	Software	VirtualBox - Ubuntu 9.10 Beta Deaktop i386 [VirtualBoxImages.com] 2	846.36 MB	2	0
28 Oct 09	Software	VirtualBox - Xubuntu 9.10 Beta i386 VDI [VirtualBoxImages.com]	760.15 MB	2	0
26 Oct 09	Software	Ubuntu-9.10Desktop amd64 Alpha1--VDI 3	611.65 MB	2	0
10 Sep 09	Software	Ubuntu 9.04 Portable 12	1.14 GB	7	1
22 Aug 09	Software	VirtualBox - Xubuntu-9.10-Alpha-4_amd64 Virtual Disk Image - [VirtualBoxImages.com]	810.78 MB	2	1

Figura 3.3: Busca por Sistema Operacional no sítio Mininova

A Figura 3.3 mostra uma busca feita no Mininova pelo sistema operacional Ubuntu[7].

[5] http://www.oi.com.br
[6] http://netcombo.globo.com
[7] http://www.ubuntu.com/

O aplicativo de busca apresenta para cada *torrent* disponibilizado as seguintes informações: data da publicação, categoria, nome, tamanho do arquivo, número de *seeders* e número de *leechers*. Basta ao usuário clicar no *torrent* desejado para passar a fazer parte do *swarm* correspondente.

3.3 Análise dos dados do ambiente real

Com o experimento montado, foi feita a coleta dos dados de janeiro de 2009 a março 2009. À medida que usuários começavam a baixar os *torrents* para seus computadores e entravam nos *swarms*, o servidor *tracker* coletava informações dos usuários. No fim do período, foram armazenados no servidor cerca de 2.5 GB de dados em arquivos texto. Para se alcançar objetivo de se encontrar *freeriders* e entender o comportamento do protocolo BitTorrent, foram desenvolvidas ferramentas para análises dos *logs* coletados.

A ferramenta seleciona informações como IP, hash do arquivo (identifica o *swarm*), bytes baixados, bytes enviados e hora da informação dos *logs*, em seguida insere esses dados em uma tabela no banco de dados MySQL. Este trabalho é feito por meio de uso de expressões regulares, já que os parâmetros passados no anúncio não seguem uma ordem pré-definida e alguns parâmetros podem não constar na especificação do BitTorrent 1.0 por serem customizações de um determinado cliente. Isso acontece porque cada cliente tem liberdade para fazer adendos ao protocolo, desde que estes adendos também funcionem com outros clientes BitTorrent que não tenham estas alterações. Também foi necessário confirmar o hash de cada torrent, já que no envio do cliente ao *tracker* alguns *hashes* apresentavam partes codificados com *bencoding* e outras partes não (ver seção 2.5.1). A tabela 3.1 mostra os trinta e sete *swarms* com a quantidade de pares participantes durante o período de observação. Os *swarms* estão ordenados pelo número de pares, o maior *swarm* tem 15315 pares, o mediano tem 9004 e o menor tem 1987 pares. Na tabela 3.1 estão enfatizados esses três *swarms*. Eles são usados a seguir para caracterizar o tráfego da rede BitTorrent. Por questão de simplicidade, eles serão denominados, respectivamente, A, B e C de agora em diante.

3.3.1 Resultados do experimento

A partir dos dados armazenados durante o experimento foi estudado o comportamento dos pares em uma rede BitTorrent para os seguints parâmetros: intervalos entre entrada de pares no sistema, número de pares por intervalo de tempo e número de *freeriders*. Para este estudo foram considerados apenas os três *swarms* selecionados anteriormente.

Swarms	Total	Swarms	Total	Swarms	Total
1	**15315**	14	9976	27	4765
2	13431	15	9951	28	4722
3	13378	16	9948	29	4211
4	13245	17	9879	30	4123
5	13247	**18**	**9004**	31	4087
6	13244	19	5319	32	3987
7	11500	20	5298	33	3934
8	10560	21	5257	34	3876
9	10590	22	5210	35	3851
10	10230	23	4950	36	3445
11	10078	24	4940	**37**	**1987**
12	10006	25	4877	–	–
13	9987	26	4865	–	–

Tabela 3.1: Todos os *swarms* com seus pares

Intervalo entre entradas

Como dito no ínicio desta seção, cada *swarm* representa três meses de coleta de dados. Para verificar o intervalo entre as entrada de pares, foram escolhidos os três dias com maior número de entradas de pares em cada *swarm*, e então calculados os intervalo entre as entradas de pares no sistema.

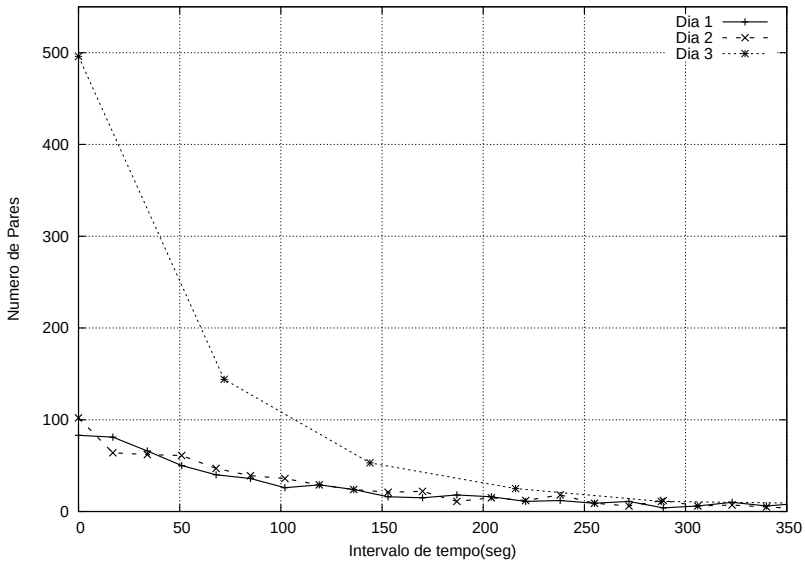


Figura 3.4: Intervalos de entrada de pares no sistema por dia do *swarm* A

As figuras 3.4, 3.5 e 3.6 mostram os intervalos entre chegadas de pares nos três dias selecionados para os *swarms* A, B e C. É possível notar pelas figuras que a chegada de pares se concentra em períodos curtos de tempo, ou seja, existem muitos pares entrando

no *swarm* próximos uns dos outros e existem longos períodos sem entrada de novos pares nos *swarms*.

Essas figuras também mostram um comportamento relativamente próximos nos três dias estudados. A exceção fica por conta do dia 3 do *swarm* A. O *trace* desse *swarm* mostra a entrada de 500 pares praticamente ao mesmo tempo o que não foi uma característica encontrada nos outros dias. Os intervalos entre as entradas tende a aumentar à medida que os usuários vão completando seus *downloads* e saindo do sistema e também quando a informação começa a ficar antiga e não desperta mais tanto interesse a novos usuários.

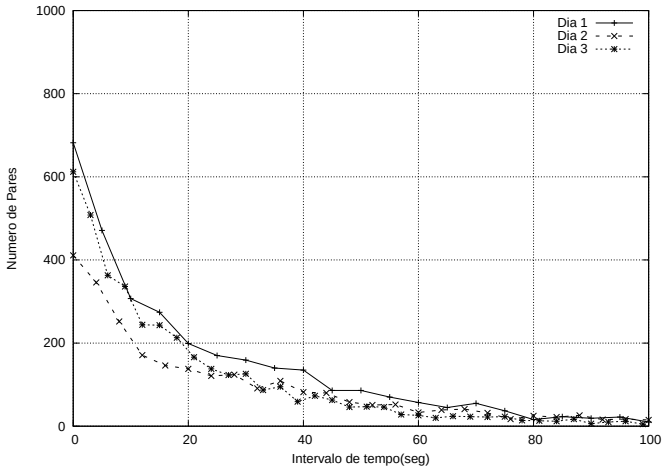


Figura 3.5: Intervalos de entrada de pares no sistema por dia do *swarm* B

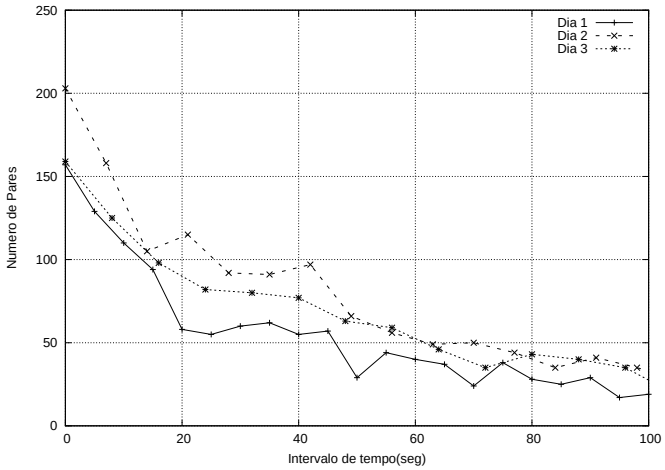


Figura 3.6: Intervalos de entrada de pares no sistema por dia do *swarm* C

O estudo dos *traces* mostrou que a chegada dos pares no experimento corresponde a uma distribuição de Poisson em todos os dias selecionados. Isto pode ser visto na figura 3.7 a figura 3.12, onde a função densidade de probabilidade dos intervalos das amostras e a função densidade de probabilidade da exponencial são plotadas para os nove dias.

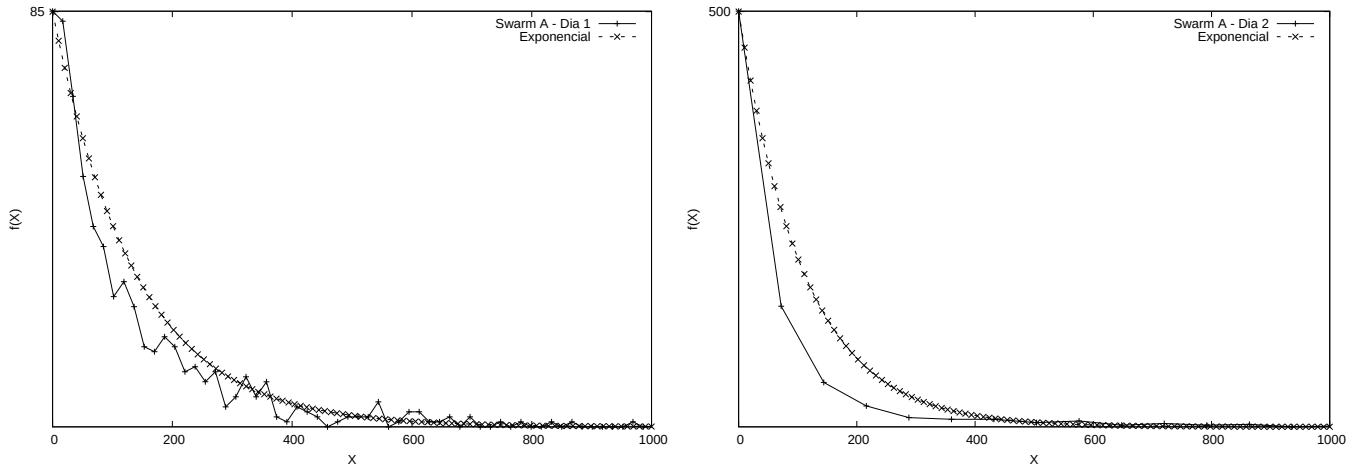


Figura 3.7: Dias 1 e 2 do *Swarm* A

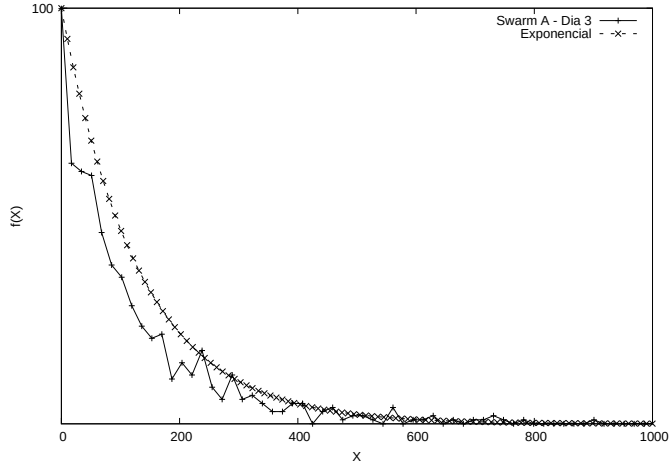


Figura 3.8: Dias 3 do *Swarm* A

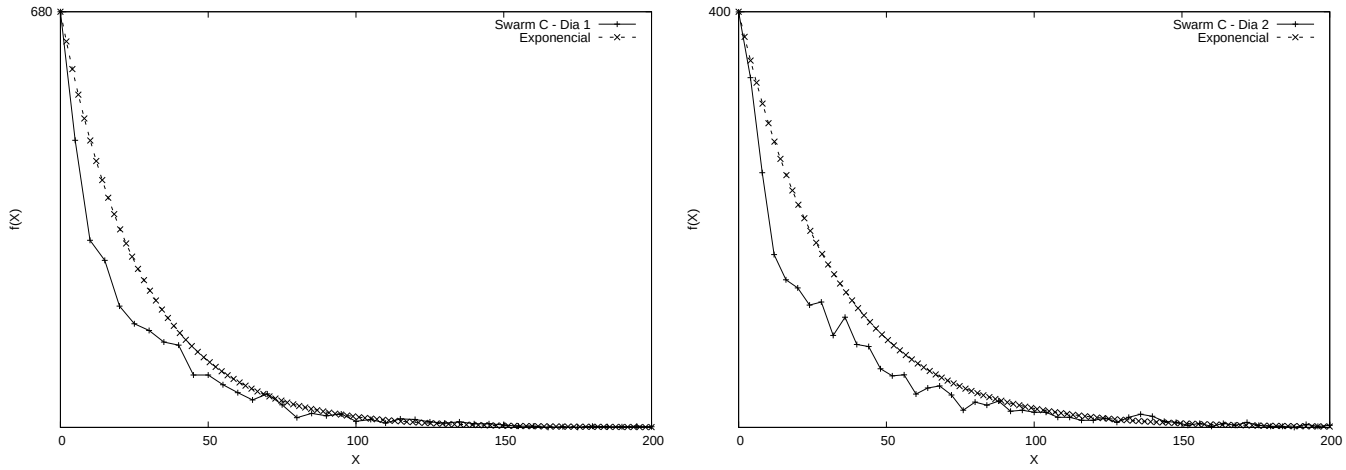


Figura 3.9: Dias 1 e 2 do *Swarm* B

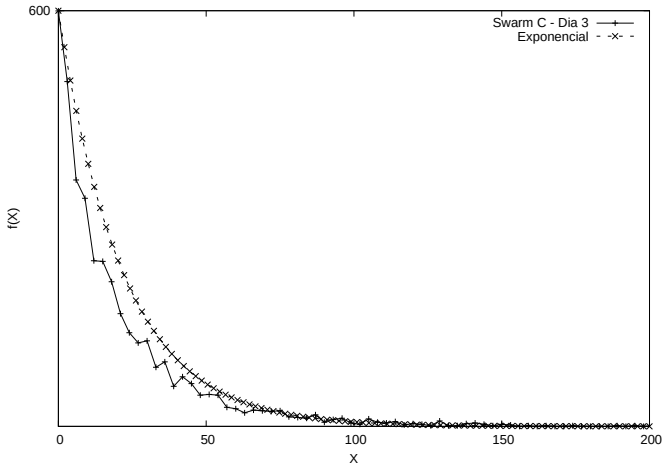


Figura 3.10: Dias 3 do *Swarm* B

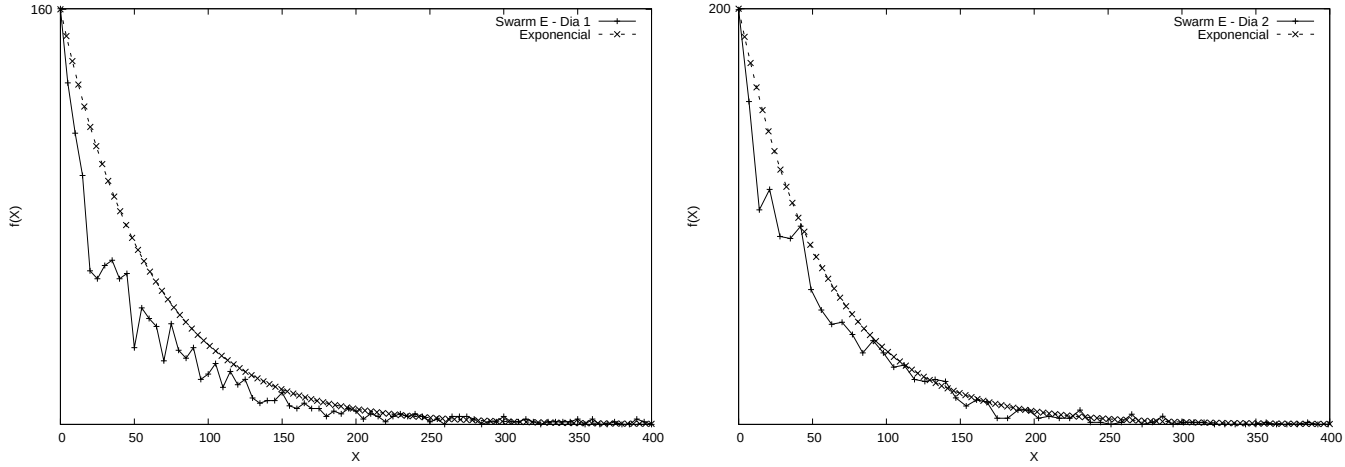


Figura 3.11: Dias 1 e 2 do *Swarm* C

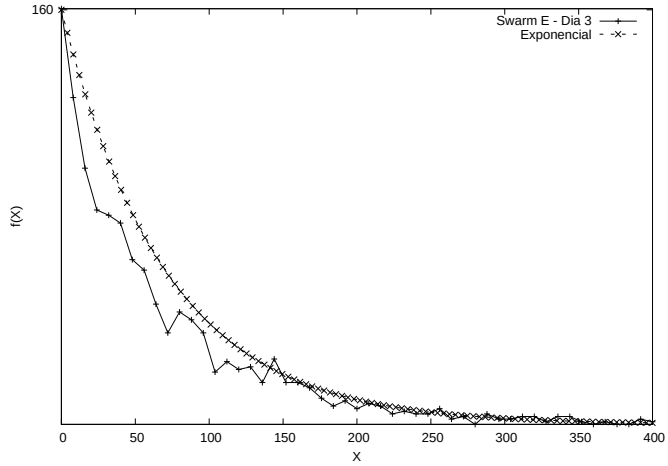


Figura 3.12: Dias 3 do *Swarm* C

Para uma maior precisão nesta análise, a tabela 3.2 mostra o erro nas funções probabilidades por dia em todos os *Swarms*. Esta tabela valida a afirmação que as funções probabilidades dos intervalos entre chegadas corresponde a uma distribuição de Poisson, pois o erro entre as funções probabilidades do intervalo entre chegada e as funções probabilidades da exponencial correspondentes são relativamente pequenas.

Swarm	Dias	Erro
A	1	0,001198
	2	0,002094
	3	0,001909
B	1	0,008906
	2	0,008470
	3	0,008219
C	1	0,005769
	2	0,004628
	3	0,002779

Tabela 3.2: Erro das funções probabilidades

Pares X Tempo

Uma visão com o número de pares dentro de um intervalo de tempo esclarece quais horários o sistema mais trabalhou, também pode-se entender melhor o comportamento diário dos usuários. Nos três *swarms* escolhidos (A, B e C) foram usados os mesmos dias do tópico anterior entre o intervalo das 00:00hs até as 23:59hs.

As figuras 3.13, 3.14 e 3.15 mostram que em todos os três sistemas existe um crescimento no número de pares no ínicio da tarde, chegando ao pico máximo no ínicio da madrugada. Na madrugada os pares começam a sair, movimento que dura até de manhã. Notam-se aqui dois aspectos interessantes sobre rede BitTorrent a partir do estudo dos *logs*: em primeiro lugar, o comportamento das entradas e das saídas de usuários por dia são parecidos; em segundo lugar a maioria dos usuários se retiram do sistema após completarem o *download* do arquivo desejado.

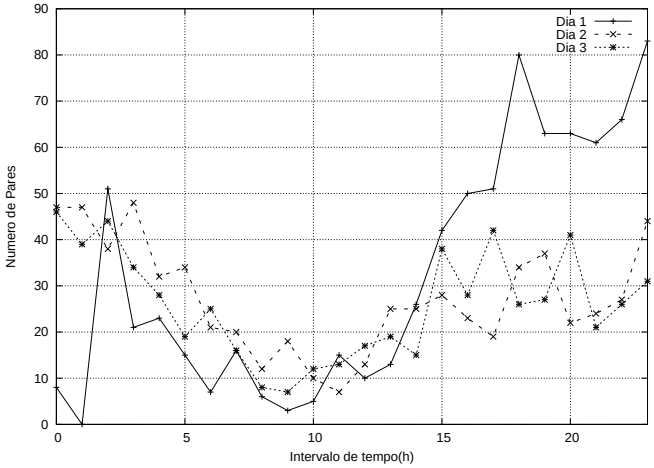


Figura 3.13: Número de pares por hora nos dias 1, 2 e 3 do *swarm* A

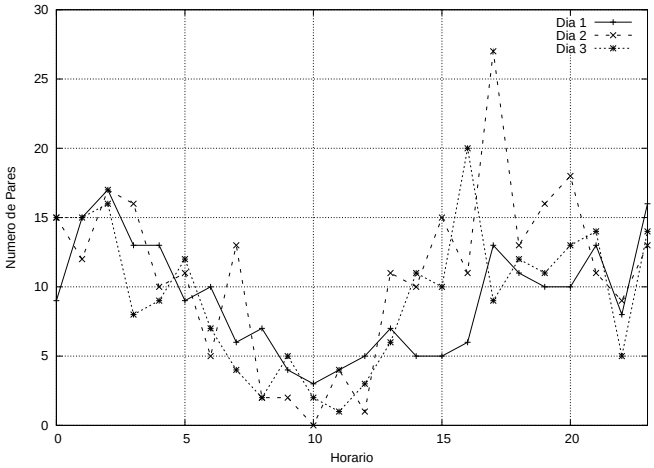


Figura 3.14: Número de pares por tempo nos dias 1,2 e 3 do *swarm* B

Busca por pares *freerider*

Nesta análise um par é considerado *freerider* quando apresenta velocidade de *upload* de zero a 2 KBps e velocidade de *download* ilimitada (esta definição é usada em [20]). Esta escolha é bem coerente, pois no Brasil ainda existem muitas conexões discadas, onde as taxas de *upload* variam de 7 a 8 KBps. Taxas abaixo desses valores podem ser entendidas como uma estratégia para economizar banda de *upload*. Segundo a pesquisa sobre o uso das tecnologias da informação e da comunicação de 2007 [3], 42% das residências com Internet no Brasil usam acesso discado, cerca de 50% dispõem de uma conexão dedicada e outros 8% não souberam informar qual a tecnologia usada na sua conexão.

A figura 3.16 ilustra a porcentagem de *freerider* nos *swarms* durante o experimento, onde os *swarms* estão agora organizados em ordem cescente do número de *freeriders*. Isto significa que o *swarm* 1 corresponde ao *swarm* com menos *freeriders* (3%), enquanto o *swarm* 37 corresponde ao *swarm* com maior quantidade de *freeriders* (16%).

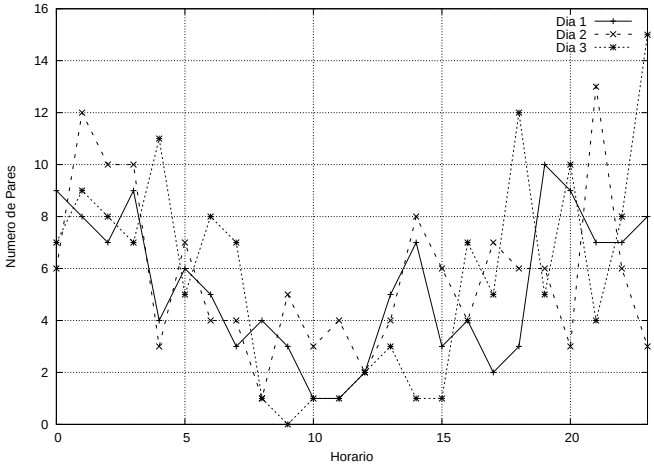


Figura 3.15: Número de pares por tempo nos dia 1,2 e 3 no *swarm* C

A figura 3.17 mostra a Função de Distribuição Acumulada da porcentagem de *freeriders* encontrados no experimento. A partir dessa FDA é possível ver que a probabilidade de um *swarm* apresentar mais de 13% de *freeriders* é igual a 20% e que a probabilidade de um *swarm* apresentar entre 3,5% e 11% de *swarms* é de 80%. Portanto, não é possível ignorar a existências de *freeriders* em uma rede BitTorrent.

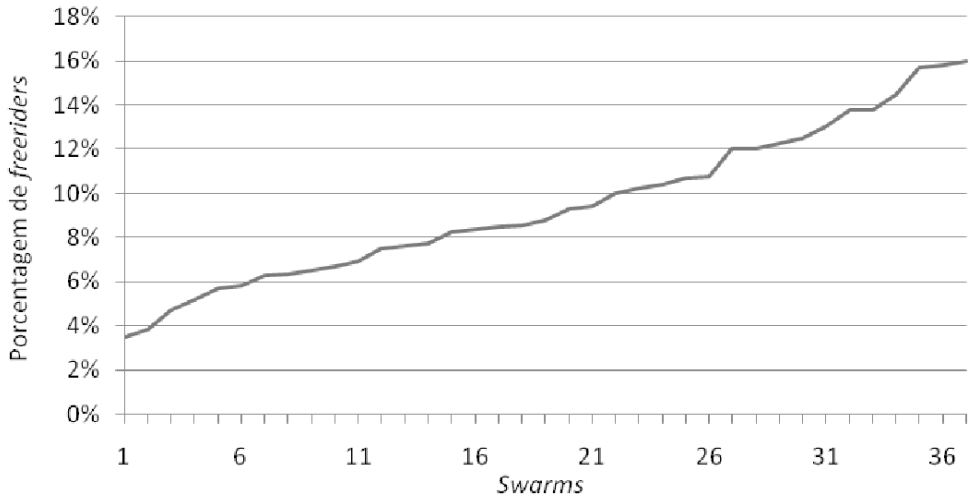


Figura 3.16: A porcentagem de *freerider* nos 37 *swarms*

3.4 Simulação

Para simulação foi escolhida a ferramenta NS-2 (Network Simulator), desenvolvida desde 1989 e atualmente mantida pela Universidade de Berkley. O NS-2 é um simulador de eventos discreto totalmente gratuito e de código aberto, muito usado em pequenas e grandes simulações. Essa ferramenta oferece uma grande variedade de tecnologias de rede,

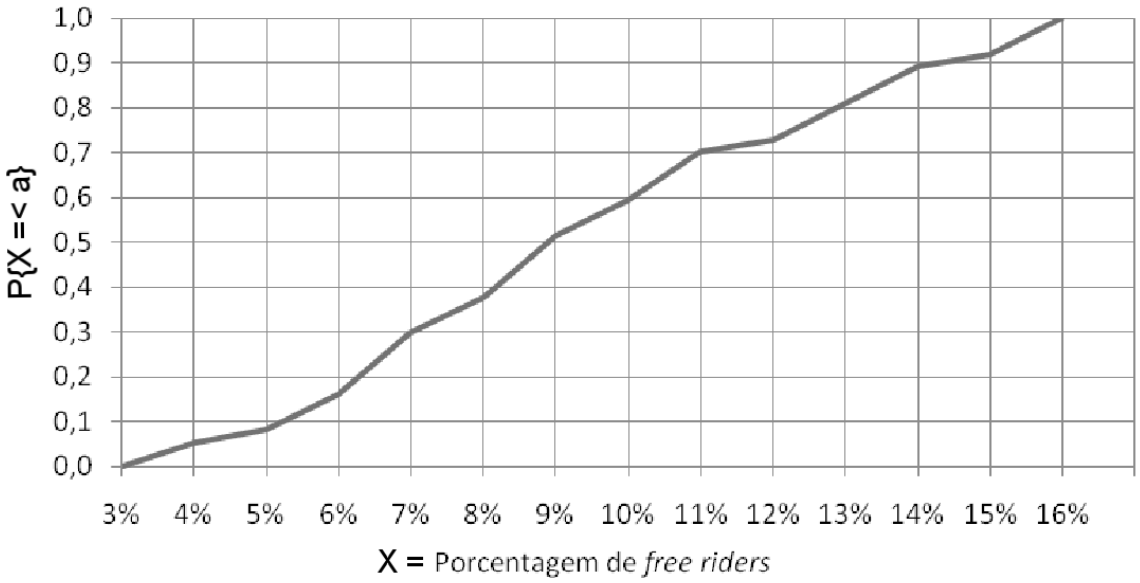


Figura 3.17: Função Distribuição Acumulada da percentagem de *freeriders*

como por exemplo, tecnologias com ou sem fio, diferente topologias baseadas em TCP e UDP, diversas políticas de fila e caracterização de tráfego com saídas para análises em texto puro ou para animações.

A programação do NS-2 é feita com duas linguagens: C++ e oTCL. A linguagem C++ é usada para descrever o comportamento dos objetos da simulação como : protocolos, agentes, nós, enlaces e geradores de tráfego. A linguagem oTCL é usada como interface entre o usuário e a simulação, onde é fornecida a descrição da rede e seus componentes como: enlaces, protocolos e agentes utilizados. O motivo dessa dualidade é a necessidade de por um lado ter uma linguagem mais robusta e rápida para manipulação de bytes, pacotes e grande conjunto de dados e por outro lado ter uma linguagem mais simples e intutiva para os usuários. Resumindo, pode-se dizer que o NS-2 é um ambiente que lê oTCL e interpreta as definições de acordo com as bibliotecas de rede em C++. A Figura 3.18 mostra uma visão simplificada de como funciona a simulação no NS-2.

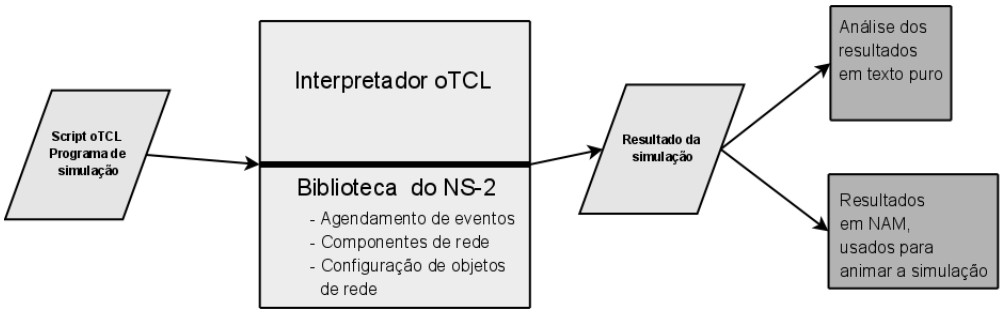


Figura 3.18: Visão do NS-2

Para se usar o NS-2, o usuário escreve um script em oTCL com configurações e as características de sua simulação. No fim, os resultados das simulações podem ser salvos em texto puro (TXT) e/ou no formato do NAM (Network Animator) usado para animar a simulação. Mais detalhes do NS-2 são apresentados no Apêndice A.

3.4.1 Implementação do protocolo BitTorrent no simulador

Neste trabalho foram implementados os seguintes componentes do BitTorrent no NS-2:

- Rastreador (*tracker*);
- Par (*peer*);
- Semeador (*seeder*);
- Sugador (*leecher*);
- Enxame (*swarm*);
- Algoritimos de seleção de pedaço;
- Algoritimos de seleção de pares (*Tit-for-Tat*);
- *freeriders*;
- Algoritimos para memória (Punição e Bonificação).

O algoritmo do BitTorrent usado neste trabalho tem como base a tese de doutorado de Kolja Eger[4] da Universidade de Tecnologia de Hamburgo, Alemanha. Primeiro, foi estudado o projeto de Eger que tinha implementado duas versões para o BitTorrent:

- **Algoritmo a nível de camada de rede** - a implementação se preocupa com todas as atividades da camada de rede como abertura da conexão entre os pares, negociação de rede, mensagens inerente ao protocolo BitTorrent.
- **Algoritmo a nível de camada de aplicação** - na implementação o foco fica com a camada de aplicação, ou seja, somente as negociações e as mensagens inerentes ao protocolo BitTorrent são detalhados.

Para este trabalho foi escolhida a implementação a nível de aplicação, pois o objetivo é estudar políticas de combate aos *freeriders* tendo como foco os algoritimos do BitTorrent como, seleção de pares e seleção de pedaços, explicados na seção 2.8.

Em seguida, foram implementadas melhorias e corrigidos erros encontrados no projeto original. O apêndice B mostra mais detalhes do algoritmo original.

3.4.2 Alterações do algoritmo do BitTorrent para simulação

O projeto original do algoritmo do BitTorrent para NS-2, foi disponibilizado em 2008 para a versão 2.0 do NS-2, no ínicio deste trabalho a versão atual já era a NS2-2.29. São alterações e implementações feitas no algoritmo original:

- Atualização do algoritmo do BitTorrent para versão atual do simulador;
- Alterações em sua estrutura para que diversos parâmetros pudessem ser modificados;
- Alterações no comportamento do objeto *Peer* nas respostas às requisições de seus vizinhos;
- Implementação de pares *freeriders*.

Também foram implementadas duas políticas para combate aos *freeriders* que serão discutidas no próximo capítulo. A seguir são fornecidos mais detalhes das alterações e implementações feitas no BitTorrent.

Atualização do algoritmo para versão atual do NS2

O algoritmo original do BitTorrent para NS-2 foi desenvolvido para versão 2.0 do NS-2, sendo necessária atualização de todas as classes que herdavam classes bases do NS-2 como: Handler, NSObject, Node e Agent. No NS-2-2.29, versão usada neste trabalho, muitas funções tinha sido descontinuadas e outras acrescentadas, por isso o código do algoritmo do BitTorrent que fazia referência a estas funções teve de ser atualizado. Antes do fim deste trabalho outras versões de NS-2 já foram lançadas, NS-2-2.34 e NS-2-3, mas não foram testadas com o algoritmo do BitTorrent implementado.

Alterações no comportamento do objeto Peer

Para execução das simulações é preciso alterar parâmetros do modelo como: capacidade dos pares executarem diversas requisições ao mesmo tempo, número mínimo de *unchokes*, intervalos de tempo dos algoritmos de *choking* e *optimistic unchoke*, etc. Para facilitar a alteração desses parâmetros, sem precisar recompilar o código, foram criados dois arquivos globais:

1. **bittorrent_params.h** - usado para ajustes das variáveis de C++ nas simulações, como:
   (a) Comportamento dos pares no atendimentos as requisições ao mesmo tempo (*pipeling*);
   (b) Possibilidade de habilitar e desabilitar mensagens para “Debug”;

(c) Controle dos intervalos de tempo para Choking, Optmistic Unchoke e Request;

(d) Controle dos contadores que indicam um par ser ou não *freerider*;

(e) Controles de taxas dos pares especiais como os*freeriders*.

2. **bittorrent_default.tcl** - usado nos ajustes das variáveis do oTcl como:

(a) Controle dos intervalos de tempo para *Choking*;

(b) Controle do tamanho das listas de pares entregues pelo *tracker* para o par requisitante;

(c) Controle do número de pares para Unchoke sem necessidade de sorteio;

(d) Aplicação ou não da política global de combate a *freeriders*;

(e) Controle do número de conexões abertas por pares.

Implementação de *freeriders*

Para implementação de *freerider* no algoritmo BitTorrent usado nas simulações foram desenvolvidos dois métodos dentro da Classe **BitTorrentTrackerFlowlevel**:

1. **select_freerider_flow(int,int)**

   Este método recebe dois parâmetros. O primeiro corresponde ao número de pares no sistema no momento e o segundo corresponde a quantidade de *freeriders* a ser escolhida na simulação. Então é retornada uma lista tipo `vector<int>` com os pares a serem configurados como *freerider*.

2. **set_freerider_flow(vector)**

   Com a lista de pares criada, os pares escolhidos são marcados como *freerider*. O *freerider* é um par especial onde duas variáveis do objeto *peer* são modificadas:

(a) taxa de *upload*, C = TX_UP_FREERIDER ;
(b) boleano is_freerider = TRUE

Tabela 3.3: Variáveis responsáveis por instanciar um par como *freerider*

As variáveis acima fazem parte do objeto *peer*, criado pela classe BitTorrentAppFlowlevel detalhado no Apendice B. A primeira variável (TX_UP_FREERIDER) define a taxa de *upload* de um par que é *freerider*, esta deverá ter um valor entre 0 e 4 KBps. A segunda variável(is_freerider) define que o par se comportará como *freerider*.

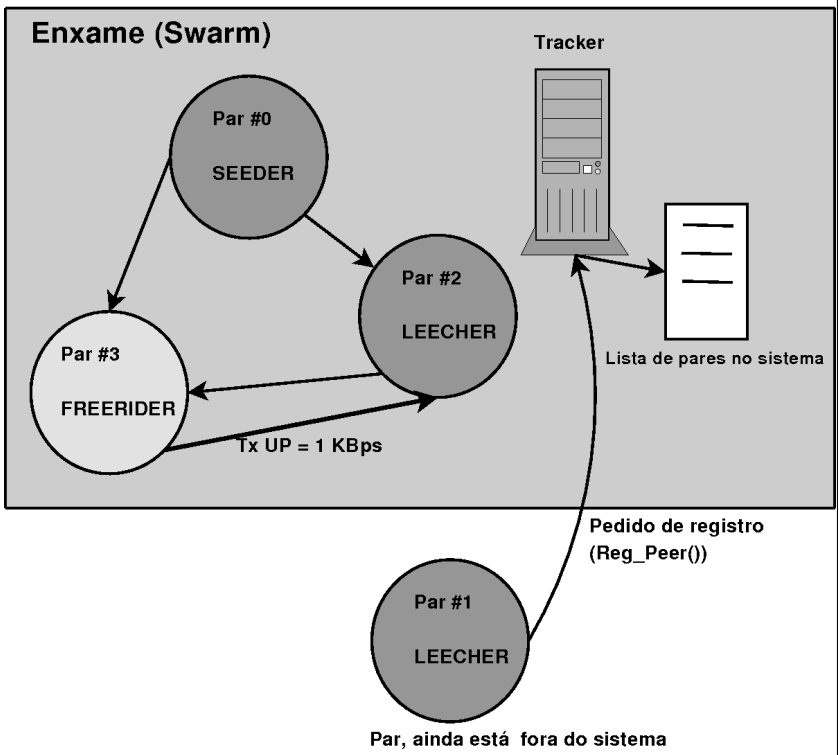


Figura 3.19: Ambiente BitTorrent com *freerider* estratégico

A figura 3.19 apresenta um ambiente da simulação em funcionamento que possui todos elementos de uma rede BitTorrent: *swarm*, *tracker*, *seeder*, *leecher* e o *freerider*. Para este trabalho o *freerider* tem uma taxa constante de 1 KBps. Neste caso já existem pares dentro do *swarm* iniciando troca de pedaços. Os pares não *freeriders* enviam pedaços usando taxas máximas de *uploads* enquanto os pares *freeriders* enviam pedaços usando taxa de *upload* de apenas 1 KBps. Também a figura mostra um par fazendo requisição ao *tracker* para poder entrar no *swarm*.

3.4.3 Funcionamento da simulação do protocolo BitTorrent

Ao iniciar a simulação, dois objetos são imediatamente criados: um *tracker*, que é responsável pelo gerenciamento do *swarm*, e um *seeder*, que é responsável por iniciar a troca de pedaços com os *peers* (é possível definir mais de um *seeder* na simulação, mas este cenário não é abordado neste trabalho). Os *leechers* vão sendo criados e vão entrando no *swarm* de acordo com uma distribuição qualquer (por exemplo, uniforme ou exponencial) ou podem ser todos criados em um mesmo instante de tempo. Quando um *leecher* entra no *swarm*, ele se identifica para o *tracker* requisitando a lista de todos os participantes. De posse da lista, os *peers* começam a fazer requisições entre si. O *seeder* é o primeiro *peer* a ser procurado pelos *leechers*, pois tem todos os pedaços. Por isso, ele responde a esses pedidos mandando pedaços distintos, possibilitando assim que os *peers* em um dado

momento na simulação comecem a trocar dados ente si. À medida que os pares baixam todos os pedaços e completam o arquivo, eles vão saindo do *swarm*. A simulação termina quando todos os pares saem do *swarm*.

Todo funcionamento da simulação descrito acima é controlado pelo *script* oTCL, nele são criados os objetos que irão compor a simulação. Neste aplicativo existem seções que são obrigatoriamente executadas como:

1. **Criação do simulador** - nesta seção é instanciado o objeto "Simulador".
2. **Seção global** - nesta seção são criadas varias variáveis globais, como:
   (a) Configurações de variáveis que alteram o comportamento da simulação;
   (b) Número de *seeders* no sistema;
   (c) Número de pares no sistema;
   (d) Taxa de *freeriders* no sistema;
   (e) Taxa global de *download* dos pares do sistema;
   (f) Tamanho do arquivo a ser compartilhado.
3. **Criação do *tracker*** - aqui é instanciado o objeto"*tracker*".
4. **Criação dos Pares** - aqui são criados os pares, de acordo com a variável "Número de pares" da seção global.
5. **Agendamento da simulação** - com todas as características do modelo já definidas, nesta seção é agendado o ínicio da simulação e consequentemente a entrada dos pares no sistema

3.5 Validação

De posse dos *traces* do experimento foi feito um tratamento dos dados a fim de filtrar inconsistências provocadas pelos mais diversos motivo como, por exemplo, paradas no servidor *tracker* (falta de energia) e informações enviadas pelos pares e gravadas pelo *tracker* de maneira errada e/ou ilegível. Para a validação, os *swarms* foram escolhidos de acordo com seu número de pares : o maior *swarm* (A), o mediano (B), e o menor *swarm* (C). O objetivo é simular uma amostra para compará-la com os *traces* do ambiente real. Portanto, os pares que interessam aqui são os que entram no sistema, baixam dados, enviam ou não pedaços e saem do sistema. É possível classificar os pares encontrados nos *traces* em três classes:

- **Pares comportados**

Pares que entram no *swarm*, baixam dados, enviam ou não pedaços. Depois de completada a informação saem do sistema. Portanto, um *freerider* pode ser considerado um par comportado nesta classificação.

- **Pares mau comportados**

  Pares que entram no *swarm*, baixam dados, mas não completam a informação, muitas vezes não saem do sistema, e também não ajudam os outros pares do *swarm*.

- **Pares não identificados**

  Pares que entram no sistema e começam a baixar dados, e, de repente, param então de receber e de enviar e simplesmente desaparecem do sistema.

Para a validação foram separadas as informações apenas sobre **pares comportados**. Os outros dois tipos de pares não foram incluídos nesta análise.

Validação com Pares comportados

Como parâmetros de entrada na simulação, foram usados os seguintes valores extraídos da análise dos *traces*:

- Número de pares comportados;
- Média e distribuição dos intervalos entre chegadas de pares;
- Tamanho médio da informação compartilhada;
- Taxa média de *upload*.

O objetivo é comparar tempo médio de *download* dos pares do *swarm* no ambiente real e no ambiente de simulação.

Número de pares comportados

Swarms	Total de Pares	Pares comportados	Porcentagem(%)
A	15315	3219	21,02%
B	9004	1965	21,82%
C	1987	353	17,77%

Tabela 3.4: Quantidade de pares comportados por *swarm*

A tabela 3.4 apresenta o total de pares e a quantidade de pares comportados dos *swarms* A, B e C. Pode-se reparar que os pares comportados correspondem a uma amostra que varia de 17% a 21% do *swarm*. Isto não quer dizer que houve problemas nos *swarms*

e nem que poucos pares participaram do sistema. Apenas que o interesse está em validar o comportamento de pares comportados, já que a simulação só é capaz de representar este tipo de par.

Média do intervalo entre chegada de pares

Com o objetivo de se atingir uma validação mais próxima da realidade, um ponto importante neste trabalho é saber a média do intervalo entre as chegadas de cada par no sistema. A tabela 3.5 apresenta os três dias de cada *swarm* selecionados na seção 3.3.1. Estes dias foram escolhidos por terem o maior número de entrada de pares que entraram durante a coleta.

swarms	**Dias**	**Pares comportados**	**Médias**	**Arquivo (MB)**	***Upload* (KBps)**
A	1	30	707,83	643,90	97,56
	2	50	472,74	611,44	96,07
	3	64	391,05	711,95	169,62
B	1	47	565,60	648,13	43,01
	2	36	663,26	558,84	117,15
	3	34	734,42	602,57	122,60
C	1	9	1820,11	701,91	1,95
	2	8	1901,50	330,50	46,38
	3	4	899,25	659,86	34,07

Tabela 3.5: Dias que mais entraram pares comportados nos *swarms* A, B e C

Para cada dia foi calculada a média dos intervalos entre as chegadas no horário das 15:00hs as 20:59hs. Este horário foi escolhido por possuir o maior número de pares comportados nos três *swarms*.

Tamanho médio da informação compartilhada

Neste tópico é calculada a média do tamanho do arquivo que está sendo compartilhado em cada *swarm* por dia. Aqui é importante ressaltar que o tamanho varia de *swarm* para *swarm*, pois um par comportado pode baixar pedaços da informação, sair e depois voltar para terminar de baixar o resto do arquivo. Isto faz com que a média do tamanho dos arquivos possa ficar menor que o tamanho do arquivo original. A tabela 3.5 mostra o tamanho médio dos arquivos de cada *swarm* nos dias escolhidos para as simulações.

Taxa média de *upload*

A tabela 3.5 apresenta também a taxa média de *upload* dos dias escolhidos nos *swarms*. As taxas são apenas dos pares comportados.

Comparação ambiente Real e Simulado

Neste tópico são apresentados as simulações com as métricas discutidas anteriormente. Foram feitas cinquenta simulações para cada ambiente. A tabela 3.6 apresenta os tempos médios de *download* do ambiente real e do ambiente simulado.

Ambiente Real				Ambiente Simulado
Swarm	Dia	*Download* (s)	Variância	*Download* (s)
A	1	72.546,50	112,80	62.322,64
	2	46.676,92	78,71	59.546,67
	3	13.345,52	46,53	26.637,43
B	1	150.115,87	71,43	153.941,30
	2	49.514,44	142,86	43.412,77
	3	16.889,38	122,61	25315,85
C	1	283.525,22	610,98	298.765,43
	2	19.351,13	583,18	25.379,36
	3	92.607,30	117,70	105.096,40

Tabela 3.6: Comparação em ambiente REAL e SIMULADO

A figura 3.6 apresenta as comparações de um dia de cada *swarm*. Pode-se notar que em todos os *swarms* o ambiente simulado obteve valores próximos aos do ambiente real.

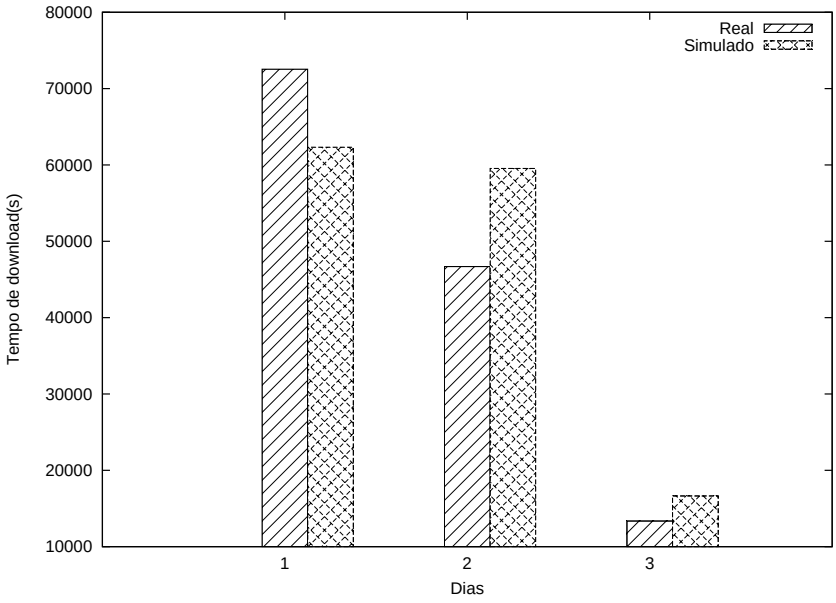


Figura 3.20: Comparação entre ambientes REAL e SIMULADO - *Swarm* A

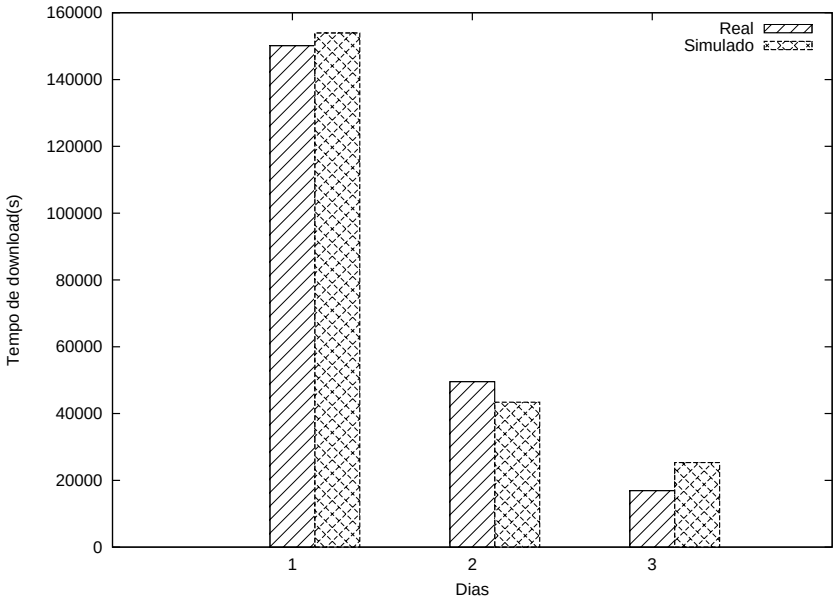


Figura 3.21: Comparação entre ambientes REAL e SIMULADO - *Swarm* B

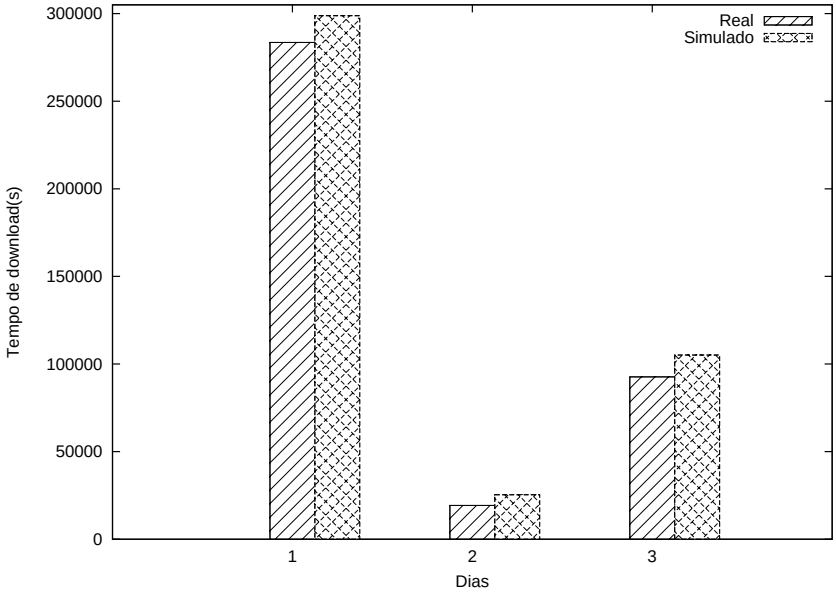


Figura 3.22: Comparação entre ambientes REAL e SIMULADO - *Swarm* C

4 Políticas Propostas

4.1 Introdução

Neste capítulo são apresentadas duas propostas para lidar com *freeriders* em redes BitTorrent. A primeira política identifica e pune os pares que apresentam comportamento não cooperativo em um *swarm*, enquanto a segunda política se preocupa em identificar e dar prioridade no atendimento aos pares que mais contribuem no *swarm*. Estes mecanismos são descritos com mais detalhes a seguir.

4.2 Política 1

O algoritmo "Tit-for-Tat" é implementado no algoritmo seleção de pares e tem como objetivo principal regular a troca de dados entre pares que melhor contribuem (possuem melhores taxas de transmissão). É um algoritmo bem eficiente no tratamento de pares que pouco colaboram, mas tem como característica perdoar esses pares através do "Optmistic Unchoke" (este algoritmo dá uma nova chance a pares que estavam em "Choke" por algum motivo). Esta característica acaba beneficiando os *freeriders* que podem ser perdoados e continuam a receber pedaços.

O objetivo da política 1 é pontuar pares que pouco contribuem dentro de um *swarm*. Quando estes alcançam uma determinada pontuação são marcados como *freeriders*.

Essa política é implementada em dois passos: no primeiro passo é verificada a existência de *freeriders* e no segundo passo os *freeriders* são punidos. A verificação e a punição são feitas durante a execução do algoritmo de "Choking", que segundo a especificação original do protocolo do BitTorrent[2] ocorre a cada 10 segundos.

Para implementar a política 1, cada par precisa manter uma tabela com informações sobre seus vizinhos. Para cada vizinho i no tempo t, o par calcula :

- $d_i(t)$ - total baixado do vizinho i até o instante t;
- $u_i(t)$ - total enviado para o vizinho i até o instante t;
- $r_i(t) = d_i(t)/u_i(t)$ - taxa de contribuição do vizinho i até o instante (t);
- p_i - quantidade de pontos acumulados pelo vizinho i até o instante (t).

Além das variáveis definidas para cada vizinho de um *swarm*, o algoritmo define duas constantes:

- f - limite superior de contribuição, ou seja, limite definido pelo algoritmo para os vizinhos ganharem ou perderem pontos.
- X - constante de definição, valor mínimo de pontos que define um vizinho como *freerider*.

Ao entrarem no *swarm* os pares solicitam ao *tracker* a lista de vizinhos para iniciar as trocas de dados. A escolha pelos melhores vizinhos acontece em intervalos periódicos dentro do algoritmo *Choking*. Com os primeiros vizinhos catalogados iniciam-se as trocas de pedaços entre eles. Também no algoritmo *Choking* ocorrem as descobertas por *freeriders* através da taxa de contribuição. Caso a taxa de contribuição $r_i(t)$ seja maior que o limite superior f, o vizinho i perde um ponto. Caso a taxa de contribuição $r_i(t)$ seja menor que o limite f, o vizinho ganha um ponto. Quantos mais pontos o vizinho tiver mais chance tem dele ser *freerider*. Como a descoberta ocorre durante a vida do par no *swarm* a pontuação dos vizinhos pode melhorar ou piorar, dependendo do seu nível de contribuição no *swarm*. No momento de enviar dados, também dentro do algoritmo *Choking*, se um vizinho está marcado como *freerider*, ele recebe a punição, ou seja, um *Choke*. A partir deste momento o vizinho não recebe mais pedaços até que deixe de ser considerado *freerider*. Ao receber esta punição, o vizinho perde um ponto, pois lhe é dado uma nova chance de mudar a sua situação.

A Tabela 4.1 apresenta o pseudo-código algoritmo de descoberta de *freeriders* e de punição aqui discutido.

Para cada vizinho i no tempo t:

1. Se $r_i(t) < r_i(t-1)$ e $r_i(t) \leq f$
   $p_i = p_i + 1$
2. Se $r_i(t) > r_i(t-1)$ e $r_i(t) > f$
   $p_i = p_i - 1$
3. Se $p_i > X$, então i é um *freerider*.
4. Se vizinho i é *freerider*, então Choke(i) e $p_i = p_i - 1$

Tabela 4.1: Política 1

As constantes X e f são discutidas com mais detalhes na seção 4.4.1 deste capítulo. As simulações feitas neste trabalho mostraram que os melhores valores para estas constantes são 3 e 0,2 respectivamente.

O algoritmo *choking* é responsável por selecionar vizinhos para *unchokes* e assim poder enviar pedaços a estes. A figura 4.1 mostra o funcionamento desse algoritmo com

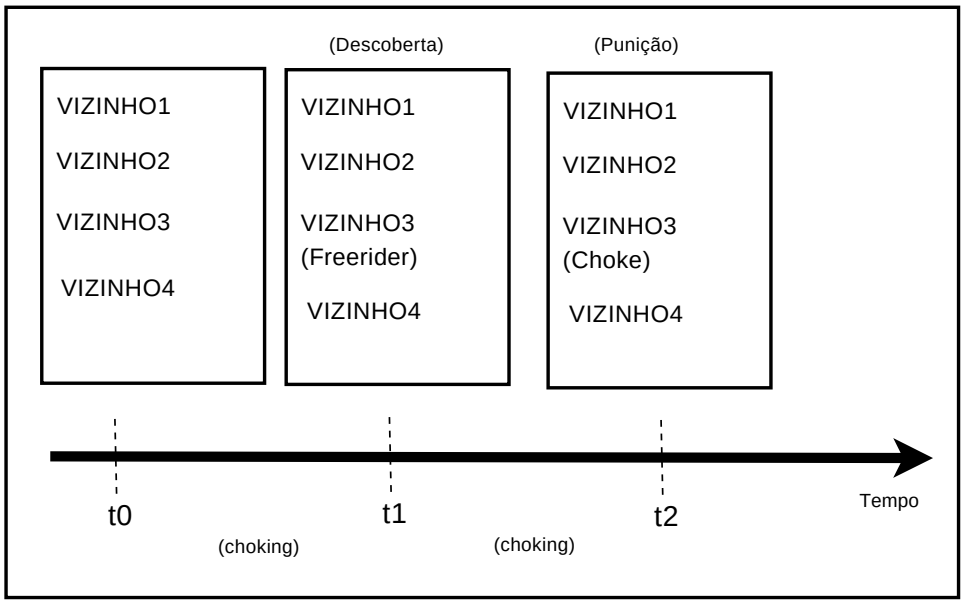


Figura 4.1: Algoritmo *choking* com política 1 habilitada

a política 1 habilitada. No tempo t0, o algoritmo *choking* cria a lista com quatro possíveis vizinhos que receberão pedaços. No tempo t1 o algoritmo descobre que o vizinho 3 é *freerider*. E no tempo t2, o vizinho recebe um *choke*, pois foi marcado anteriormente como um *freerider*.

4.3 Política 2

Esta implementação se assemelha à anterior, com uma diferença: aqui quem recebe os pontos são os bons contribuidores. O objetivo da política 2 é pontuar os pares que mais contribuem dentro de um *swarm*. Quando um vizinho alcança uma determinada pontuação, ele é marcado como Bom Contribuidor. Abaixo será descrito o funcionamento deste mecanismo.

Como na política 1, a política 2 também é implementada em dois passos: no primeiro passo é verificada a existência de Bons Contribuidores e no segundo passo, os pares considerados bons contribuidores são recompensados. A verificação e a recompensa são feitas dentro do algoritmo de *choking* e do algoritmo de *optmistic unchoking*, que segundo a especificação do protocolo BitTorrent ocorrem em intervalos de 10 e 30 segundos, respectivamente.

Na Política 2 cada par também mantém uma tabela com informações sobre seus vizinhos. Para cada vizinho i no tempo t, o par calcula :

- $d_i(t)$ - total baixado do vizinho i até o instante t;
- $u_i(t)$ - total enviado para o vizinho i até o instante t;

- $r_i(t) = d_i(t)/u_i(t)$ - taxa de contribuição do vizinho i;
- p_i - quantidade de pontos acumulados pelo vizinho i.

São também definidos três constantes na política 2:

- f - limite superior de contribuição, o vizinho i que ultrapassar este limite ganham dois pontos. Até este limite o vizinho pode ganhar ou perder um ponto.
- f' - limite inferior de contribuição, o vizinho i que ficar abaixo deste limite, perde dois pontos. Acima deste limite o vizinho poder ganhar ou perder um ponto.
- Z - constante de definição, valor mínimo de pontos que define um vizinho como BOM CONTRIBUIDOR.

As contantes f, f' e Z são explicadas com mais detalhes a seguir no tópico 4.4.2.

Ao entrarem no *swarm* os pares solicitam ao *tracker* a lista de vizinhos para iniciar suas trocas de dados. Dentro do algoritmo *Choking* são escolhidos os primeiros quatro vizinhos. Com os vizinhos catalogados iniciam-se as trocas de pedaços entre eles. Também no algoritmo *Choking* ocorrem as buscas pelos Bons Contribuidoes, para isso o par varifica a taxa de contribuição $r_i(t)$ do seu vizinho i no instante t. Caso a taxa de contribuição tenha aumentado em relação ao intervalo anterior $t-1$ e seja maior que o limite superior de contribuição f, este vizinho recebe dois pontos. Caso a taxa de contribuição $r_i(t)$ esteja entre o limite inferior de contribuição f' e o limite superior de contribuição f, este vizinho recebe um ponto. A idéia aqui é pontuar para quem melhorar sua contribuição entre dois instantes da aplicação do algoritmo. Caso a taxa de contribuição tenha diminuído em relação ao intervalo anterior $t-1$ e seja menor que o limite inferior de contribuição f', este vizinho perde dois pontos. Caso a taxa de contribuição $r_i(t)$ esteja entre o limite inferior de contribuição f' e o limite superior de contribuição f, este vizinho perde um ponto. A idéia é tirar pontos do vizinho caso sua taxa de contribuição venha caindo sucessivamente. Quantos mais pontos o vizinho tiver mais chance tem de ser um Bom Contribuidor. Como a descoberta ocorre durante toda vida do par no *swarm* a pontuação de seus vizinhos pode melhorar ou piorar, variando o estado deste ser um Bom Contribuidor ou não. No mecanismo de *Choking* são escolhidos os pares que serão *unchokes*, o máximo de quatro, caso a lista de vizinho do par seja superior então é feita uma escolha pelos quatro vizinhos com maior quantidade de pontos, ou seja, os Bons Contribuidores. No *Optimistic unchoking*, com vizinhos ainda não escolhidos no algoritmo de *choking*, é criada uma nova lista onde será escolhido o vizinho com mais pontos para receber o *unchoke*, a recompensa é dar prioridade a este vizinho receber dados deste par. O que se pretende é simples:

- Atender primeiro quem tem mais pontos (*Choking*);
- Perdoar primeiro quem tem mais pontos (*Optistic Unchoking*);

A tabela 4.2 apresenta o pseudo-código algoritmo de descoberta de Bons Contribuidores e a da recompensa aqui discutidos.

Para cada vizinho i no tempo t:

1. Se $r_i(t) > r_i(t-1)$ e $r_i(t) > f$
   $b_i = b_i + 2$
2. Se $r_i(t) > r_i(t-1)$ e $f' \leq r_i(t) \leq f$
   $b_i = b_i + 1$
3. Se $r_i(t) < r_i(t-1)$ e $f' \leq r_i(t) \leq f$
   $b_i = b_i - 1$
4. Se $r_i(t) < r_i(t-1)$ e $r_i(t) < f'$
   $b_i = b_1 - 2$
5. Se $b_i > Z$, então vizinho i é um Bom Contribuidor.

   Nos algoritmos de *choking* e *optimistic choking*:

6. Se vizinho i é um Bom Contribuidor, então prioridade no Unchoke(i).

Tabela 4.2: Política 2

As constantes Z, f e f' serão discutidas com mais detalhes na seção 4.4.2 deste capítulo. Entretanto as simulações mostraram que os melhores valores para estas constantes são 5, 0,8 e 0,5 respectivamente.

Na figura 4.2 ilustra o funcionamento do algoritmo *choking* em seus intervalos de tempo com a política 2 habilitada, ou seja, com a descoberta e bonificação para os Bons Contribuidores. No tempo t0, o algoritmo *choking* cria uma lista com quatro possíveis vizinhos que receberão pedaços. No tempo t1 dentro do algoritmo de *choking* ocorrem as descobertas por Bons Contribuidores, ou seja, os vizinhos que mais contribuiram no *swarm* ganham pontos podendo ser marcados como Bons Contribuidores. Neste exemplo, os vizinhos 2 e 3 são considerados Bons Contribuidores. No tempo t2, dentro do algoritmo de *choking* e do algoritmo *optimistic unchoke*, quando o par selecionar mais vizinhos para mandar pedaços terão prioridade na escolha os vizinhos que estiverem marcados como Bons Contribuidores. No exemplo da figura 4.2, os vizinhos 2 e 3 estão no topo da lista para receber *unchokes* e pedaços por terem maior pontuação.

4.4 Constantes usadas nas simulações

Nesta seção são discutidas as constantes usadas nos algoritmos das políticas. As duas políticas usam as constantes controlar a punição dos *freeriders* e a recompensa dos Bons

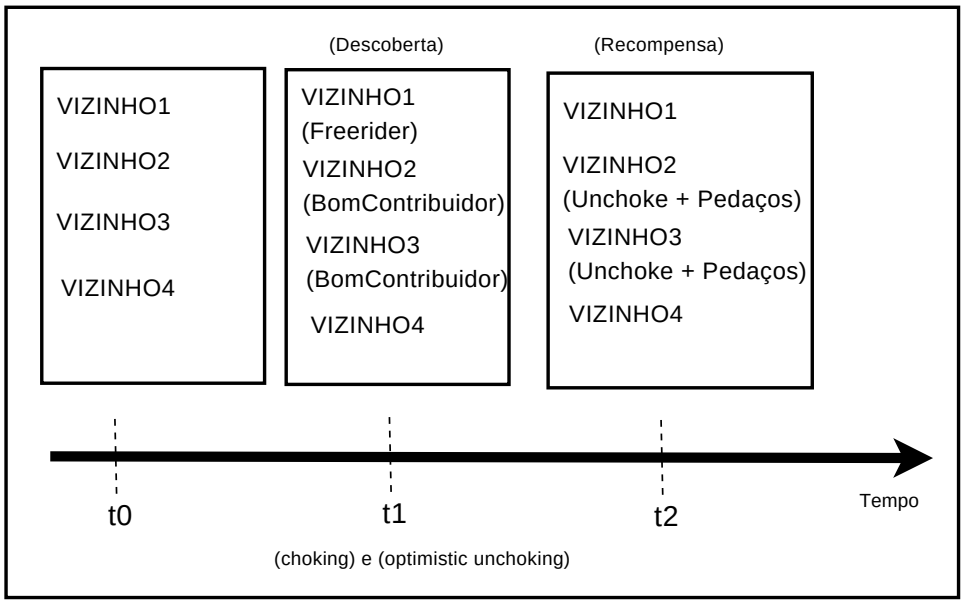


Figura 4.2: Algoritmo *choking* com política 1 habilitada

Contribuidores.

4.4.1 Política 1

Limite superior de contribuição (f) - Este número define se o vizinho irá ganhar ou receber pontos. A idéia é que este número represente a porcentagem de contribuição de um vizinho em relação ao seu par. Por exemplo, 0,3 representa 30% de contribuição. Quando o algoritmo da Política 1 é chamado, este testa a taxa de contribuição $r_i(t)$ do seu vizinho no intervalo atual em relação a taxa de contribuição $r_i(t-1)$ do intervalo anterior. Caso a taxa tenha aumentado e seja maior que f, então o vizinho perde um ponto. Caso a taxa de contribuição $r_i(t)$ tenha diminuído e seja menor que f, então o vizinho ganha um ponto. Quando número de limite f é alterado o número de *freeriders* encontrados pelos pares também sofrem alterações, pois quanto menor for este número, menor o número de vizinhos que marcam pontos, diminuindo o número de *freeriders* encontrados.

Variável de definição (X) - Este número define se o vizinho é ou não um *freerider*, ou seja, se a pontuação do vizinho for superior a este número, ele é um *freerider*. Ao alterar este parâmetro também muda-se o comportamento da política na busca aos *freeriders*, pois se a variável de definição for muito pequena, poucos *freeriders* são encontrados.

Para verificar quais os melhores resultados para as contantes f e X foram feitas cinquenta simulações com 50 e 75 pares, em ambiente com dez *freeriders* (20%) e com vinte *freeriders* (40%). Considerou-se para teste que a constante X tem valor igual a 3.

Pode-se notar que nas figuras 4.3 e 4.4 os melhores resultados aparecem quando se ajusta o limite superior de contribuição f para 0,2, apesar de no ambiente com 75 pares e 40% de *freeriders* o valor de 0,3 ficou muito próximo de 0,2. Em todas as simulações os

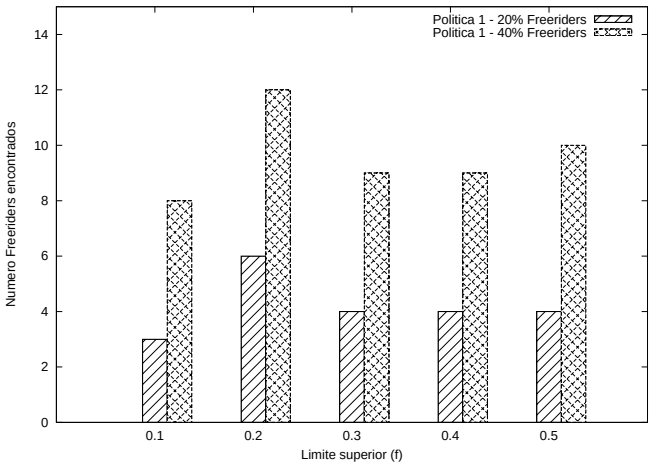


Figura 4.3: Limite superior de contribuição (f) - 50 pares

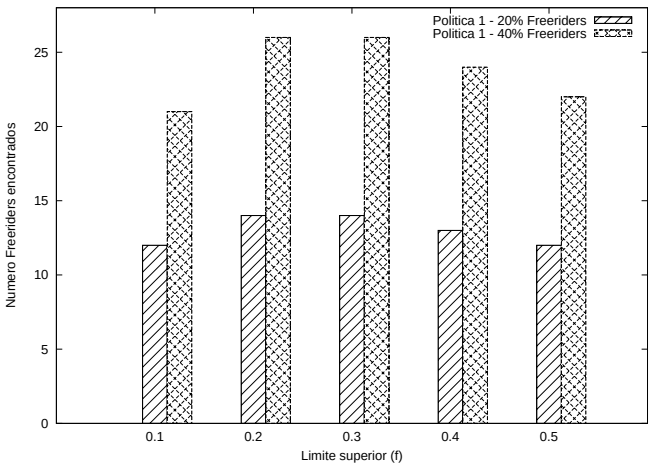


Figura 4.4: Limite superior de contribuição (f) - 75 pares

pares encontraram mais *freeriders* que nos outros valores, tanto no ambiente com 20% de *freeriders* quanto no ambiente com 40%.

Considerando f como 0,2, a constante X foi testada para os valores de 1 a 5. As simulações foram iguais a anterior. As figuras 4.5 e 4.6 mostram que o fator de definição X que mais se encontrou *freerider* foi 3, apesar de no ambiente de 75 pares e 40% de *freeriders* o valor 1 também apresentou bons resultados na busca aos *freeriders*. Este resultado foi melhor tanto no ambiente com 20% de *freeriders* quanto no ambiente com 40%. Em resume os melhores valores encontrados foram 0,2 e 3 para f e X respectivamente.

4.4.2 Política 2

O algoritmo da política 2 é diferente do algoritmo da política 1. Aqui foram criadas duas faixas onde o vizinho poderá ganhar ou perder um ou dois pontos. A idéia é simples: se o vizinho contribui, ganha um ponto, se contribui mais ainda, ganha dois pontos. A lógica é

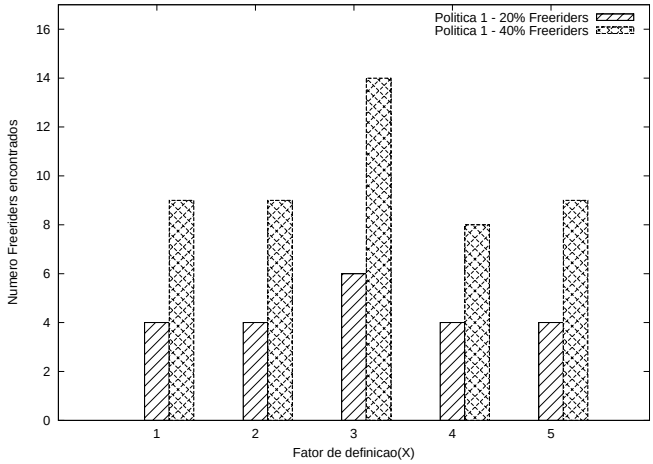


Figura 4.5: Fator de definição (X) - 50 pares

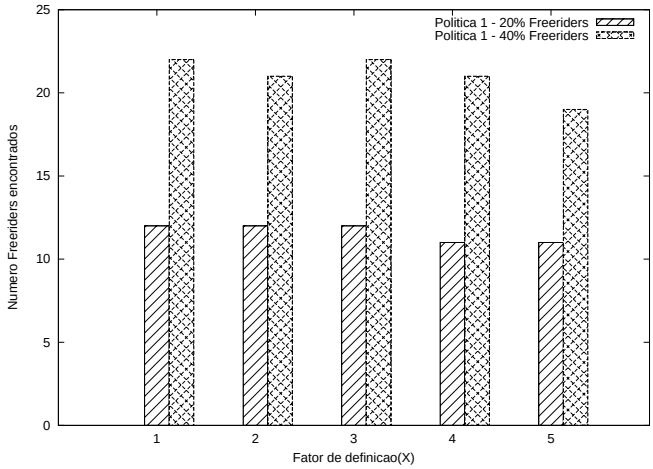


Figura 4.6: Fator de definição (X) - 75 pares

a mesma para perder pontos: se o vizinho não contribui, ele perde um ponto, se o vizinho continua não contribuindo, perde dois pontos. Para implantar esse sistema foram criadas duas constantes f e f' que são explicadas a seguir.

Limite superior de contribuição (f) - Este número define se o vizinho irá ganhar ou receber pontos. A idéia é que este número represente a porcentagem máxima de contribuição de um vizinho em relação ao seu par, para ganhar ou perder pontos. Quando o algoritmo da política 2 é chamado, ele testa a taxa de contribuição $r_i(t)$ do seu vizinho no intervalo atual em relação a taxa de contribuição $r_i(t-1)$ do intervalo anterior. Caso a taxa tenha aumentado e seja maior que f então o vizinho ganha dois pontos. Caso a taxa de contribuição $r_i(t)$ esteja entre o limite inferior f' e o limite superior f então o vizinho ganha um ponto. Quando número de limite f é alterado, o número de Bons Contribuidores encontrados pelos pares também sofrem alterações, pois quanto maior for este número, menor o número de vizinhos que marcam pontos, diminuindo o número de Bons

Contribuidores encontrados.

Limite inferior de contribuição (f') - Este número define se o vizinho irá ganhar ou receber pontos. A idéia é que este número represente a porcentagem mínima de contribuição de um vizinho em relação ao seu par, para perder pontos. Quando o algoritmo da política 2 é chamado, este testa a taxa de contribuição $r_i(t)$ do seu vizinho no intervalo atual em relação a taxa de contribuição $r_i(t-1)$ do intervalo anterior. Caso a taxa tenha diminuído e esteja entre o limite inferior f' e o limite superior f então o vizinho perde um ponto. Caso seja menor que f' então o vizinho perde dois pontos. Quando número de limite f' é alterado, o número de Bons Contribuidores encontrados pelos pares também sofrem alterações.

Variável de definição (Z) - Este número define se o vizinho é ou não um Bom Contribuidor, ou seja, se a pontuação do vizinho for superior a este número, ele é um Bom Contribuidor. Ao alterar este parâmetro também muda-se o comportamento da política na busca aos Bons Contribuidores, pois se a variável de definição for muito pequena ou muito grande, poucos Bons Contribuidores serão encontrados.

Para encontrar os melhores valores para f, f' e Z foram feitas diversas simulações. Estas mostraram como melhores resultados para f e f' são 0,5 e 0,8 respectivamente. Para simplificar o algoritmo e o número de variações nestas constantes, fixou-se como o limite inferior 50% de contribuição, ou seja 0,5 para f', já que abaixo deste valor não teria sentido pois 0,2 já entra na faixa da política 1. Então variou-se aqui os valores do limite superior f de 0,6 a 1,0 e os valores do fator de definição Z de 1 a 5.

Para alcançar os melhores resultados para as contantes foram feitas cinquenta simulações com 50 e 75 pares, com dez *freeriders* (20%) e com vinte *freeriders* (40%).

Foram duas rodadas de simulações: a primeira com os valores f e f' de 0,5 e 0,8 e variando o Z de 1 a 5 e a segunda com os valores f' e Z, variando o f de 1 a 5.

Em todas as figuras, 4.7, 4.8, 4.9 e 4.10 os melhores resultados para constantes Z, f são 5 e 0,8, respectivamente.

4.5 Comparação entre políticas

4.5.1 Ambiente para experimentos

Nesta seção foram realizados vários experimentos com cenários diferentes. O objetivo é estudar a eficiência das políticas no combate aos *freeriders* e a sobrecarga destas políticas no ambiente BitTorrent. Nas simulações, assume-se que os pares entrem no sistema ao mesmo tempo e que no fim de seus *downloads* eles se retiram. A principal métrica de estudo foi o tempo de *download* nos diversos cenários. Foram considerados três cenários para as simulações: pares usam protocolo BitTorrent sem nenhuma política para *freeriders*; pares usam a política 1 para punir *freeriders* e pares usam política 2 para recompensar vizinhos que participam ativamente da distribuição de pedaços. Para cada cenário considerou-se *swarms* de 50 e 75 pares e que a porcentagem de *freeriders* pode variar de zero a 50%. Cada simulação possui *tracker*, um *seeder*, taxa de *upload* de 256 Kbps e tamanho do arquivo de 100MB. Esta definição é também utilizada em Sherman[14], Jun[7] e Zghaibeh[20].

Antes de apresentar os resultados das simulações, é importante observar que o protocolo BitTorrent usa internamente diversos parâmetros que alteram o seu comportamento. Alguns destes parâmetros são apresentados na tabela 4.3. Neste trabalho foram configurados os valores padrões como recomendados por Bram Cohen [2].

Tipo do Par	Parâmetro	Valor
Tracker	max_give	200
Seeder	max_initiate	40
	min_peers	20
	pipelined_requests	5
	num_from_tracker	50
	size_chunk	256 KB
Leecher	max_initiate	40
	min_peers	20
	pipelined_requests	5
	num_from_tracker	50
	check_choking	10s
	optimistic_unchoking	30s
	size_chunk	256 KB

Tabela 4.3: Parâmentos padrões

Abaixo, uma breve descrição dos parâmetros e os valores usados na siulação:

- **max_give** - número máximo de pares que são atendidos em um *swarm*. O padrão é 200.
- **max_initiate** - quantidade de vizinhos conhecidos por um par. Este parâmetro determina quando o cliente deve parar de iniciar conexões com novos vizinhos. O padrão é 40.
- **min_peers** - determina o número mínimo de pares que o cliente BitTorrent deve estar conectado antes de parar de perguntar ao Tracker por mais pares. O padrão é 20.
- **pipelined_requests** - número de conexões máximas simultâneas que um cliente BitTorrent pode fazer com seus pares. O valor padrão é 5.
- **num_from_tracker** - quantidade máxima de pares na lista que o Tracker envia para clientes BitTorrent quando solicitado. O valor padrão é 50.
- **size_chunk** - tamanho do pedaço em KiloBytes.O valor padrão é 256.
- **choking** - intervalo de tempo entre duas execuções do algoritmo *choking*. O recomendado é de 10s.
- **optimistic_unchoking** - intervalo de tempo entre duas execuções do algoritmo *optimistic_unchoking*. O recomendado é de 30s.

4.6 Resultados obtidos

Depois das simulações executadas, iniciou-se as comparações, a primeira no cenário sem política e depois nos cenários com política 1 e com política 2. Em todas as comparações usou-se o tempo de *download* dos pares no sistema como parâmetro para verificar ou não a melhoria do ambiente BitTorrent. O objetivo principal aqui é saber o quanto os *freeriders* prejudicam o sistema, qual das políticas é mais eficiente no combate aos *freeriders* e quanto estas políticas aumentam o tempo de *download* dos pares no sistema.

4.6.1 Cenário sem política

Sistemas com *freerider* X Tempo de *download*

Os experimentos iniciais foram feitos em ambientes com 50 e 75 pares e variando o número de *freeriders* de 0% a 50%. Nenhuma política é usada neste ambiente. Para cada cenário foram executadas cinquenta simulações.

Nas tabelas 4.4 e 4.5 todas as variações o tempo de *download* aumentaram em relação a um ambiente sem *freerider* (0%). Como apresentado no estudo de Sherman [14], sistemas com *freeriders* tendem a degradar seu tempo de *download* para todos os pares:

Free-Riders	Tempo médio download	Aumento do tempo de download
0%	5366,27	0,00
10%	5856,65	9,14%
20%	6312,48	17,63%
30%	6888,54	28,37%
40%	7659,97	42,74%
50%	8433,27	57,15%

Tabela 4.4: Cenário com 50 pares

Free-Riders	Tempo médio download	Aumento do tempo de download
0%	5214,11	0
10%	5383,89	3,26%
20%	5927,32	13,68%
30%	6537,24	25,38%
40%	7062,62	35,45%
50%	8169,22	56,68%

Tabela 4.5: Cenário com 75 pares

nos *swarms* com 50 pares, o aumento no tempo médio de *download* é em torno de 20% em sistemas com 20% de *freeriders* e 60% de aumento em sistemas com 40% de *freeriders*. A tabela 4.4 apresenta valores similares ao trabalho do Sherman[14], as simulações mostram aumento de 17,61% quando há 20% de *freeriders* e 42,74% quando há 40% de *freeriders*. É importante ressalvar que o tamanho do arquivo compartilhado é de 64MB no trabalho Sherman[14] e que o tamanho de arquivo deste trabalho é de 100MB. A tabela 4.5 mostra o tempo de *download* para *swarm* com 75 pares. Note que a degradação do sistema é um pouco menor. Isto se explica pelo fato do *swarm* ter mais pares.

Pares não *freeriders* X *Freeriders*

Outra preocupação é saber quanto os pares normais (pares não *freeriders*) são prejudicados por pares *freeriders*. Com dados nas simulações acima foi comparado o tempo de *download* de ambos os pares.

As figuras 4.11 e 4.12 compara os tempo de *downloads* em um ambiente com 50 e 75 pares respectivamente. Neste cenário os *freeriders* sempre terminam os *downloads* primeiro que os outros pares. Pode-se concluir, portanto, que ser *freerider* em um ambiente sem política é mais vantajoso.

4.6.2 Cenário com políticas para *freeriders*

Com dados sobre os danos que *freeriders* causam ao sistema, as simulações agora habilitam as políticas de combates a eles. Os cenários são os mesmos, a diferença é que agora

as políticas 1 e 2 são habilitadas.

Combatendo os *freeriders*

As figuras 4.13 e 4.14 mostram os resultados das simulações feitas com as políticas habilitadas. O objetivo é saber se as políticas combatem os *freeriders* de forma adequada. Cada uma das figuras mostra o tempo de término de *download* dos *freeriders* em simulações com 10% a 50% de *freerider*, sem aplicação das políticas, com aplicação da política 1 e com aplicação da política 2.

No primeiro e no segundo cenários, representados pelas figuras 4.13 e 4.14 respectivamente, tanto a política 1 como a política 2 obtiveram sucesso no aumento dos tempos de *download* dos *freeriders*.

Na figura 4.13, ambiente com 50 pares, a política 1 obteve um desempenho ligeiramente melhor que a política 2. Pode-se observar que em situações com 20%, 40% e 50% de *freeriders* a política 1 consegue aumentar mais os tempo de *downloads* dos *freeriders* em relação a política 2, já em situações com 10% e 30% ocorre o inverso a política 2 atrasa mais os *freeriders* do que a política 1. Na figura 4.14, ambiente com 75 pares, a política 2 foi melhor em todas as situações, ou seja, em sistemas com 10% a 50% de *freeriders* a política 2 aumentou mais os tempos de *downloads* que a política 1, mostrando que nesses ambientes para combater *freeriders* é mais interessante bonificar e dar prioridade a quem mais contribui do que punir quem pouco contribui.

Neste tópico pode-se concluir que, no combate aos *freeriders*, a política 2 foi mais eficiente que a política 1, mas deve-se levar em conta o quanto este combate prejudicou quem não é *freerider*.

Ajudando os pares não *freeriders*

Neste tópico o objetivo é analisar a sobrecarga das políticas 1 e 2 nos pares não *freeriders*. Para isto são comparados os tempos de *downloads* dos pares, não *freeriders* em presença ou não das políticas 1 e 2.

No ambiente com 50 pares (Figura 4.15), os pares sofrem um aumento significativo em seus tempos no sistema tanto na política 1 quanto na política 2. A política 1 tem desempenho ligeiramente melhor que a política 2, ou seja, os pares não *freeriders* habilitados com a política 1 saem primeiro do sitema do que pares não *freeriders* habilitados com a política 2.

No cenário com 75 pares (Figura 4.16), o resultado é semelhante: a política 1 é melhor em todas as simulações do que a política 2.

A partir dessas simulações, é possível deduzir que política 1, mesmo sendo projetada para punir *freerider*, foi de um modo geral mais eficiente que a política 2 no quesito ajuda ao par não *freerider*.

Políticas X Sistema

Neste tópico é mostrado o quanto a implementação das políticas 1 e 2 aumentou o tempo médio de download no sistema.

Freerider	Aumento Politica 1	Aumento Politica 2
0%	23,56%	28,08%
10%	9,88%	26,45%
20%	9,08%	11,08%
30%	9,17%	13,62%
40%	8,27%	11,18%
50%	8,41%	5,94%

Tabela 4.6: Aumento do tempo no sistema no ambiente com 50 pares

Freerider(%)	Aumento Politica 1	Aumento Politica 2
0%	1,95%	27,77%
10%	2,17%	26,53%
20%	8,99%	31,73%
30%	20,77%	49,91%
40%	36,90%	54,50%
50%	22,76%	31,59%

Tabela 4.7: Aumento do tempo no sistema no ambiente com 75 pares

As tabelas 4.6 e 4.7 mostram o aumento no tempo médio de *download* de todos os pares no *swarm* quando habilitadas as políticas com 0% a 50% de *freerider* nos cenários com 50 e 70 pares, respectivamente.

A tabela 4.6 mostra que em situações com 0% de *freeriders* a política 1 provocou um aumento de 23% no tempo de *download* no sistema e a política 2 provocou um aumento de 28%. A partir de 10% de *freeriders* ocorre um decréscimo deste aumento em ambas as políticas, com destaque a política 1 que em simulações com 50% de *freeriders* houve apenas 8% de aumento no tempo de *download* do sistema.

Já a tabela 4.7 mostra que em 0% de *freeriders* a política 1 provocou um aumento de 2% enquanto a política 2 foi de 27%. A partir de 10% a 30% esse aumento no tempo de sistema mantém uma coerência com a política 1 com aumento pequenos e a política 2 com aumentos maiores semelhantes ao ambiente anterior, mas em 40% tanto a política 1 e política 2 provocam aumentos maiores, acima de 37% chegando a 54%.

Pode-se concluir que a política 1 foi a mais eficiente em ambiente com 50 e 75 pares que a política 2. Em ambiente com 50 pares a política 1 provocou um aumento no tempo de *download* em cenários sem *freeriders*, em seguida com a entrada de *freeriders* este aumento diminui provando uma boa eficiência da política. A política 2 tem um melhor

desempenho em sistema com 50% de *freeriders* onde o aumento no tempo de *download* provocado foi apenas de 6%. Em ambientes com 75 pares a política 1 novamente foi melhor do que a política 2, apesar de cenários acima de 40% de *freeriders* o aumento provocado foi muito alto.

Em resumo a aplicação das políticas 1 e 2 degradam o sistema em todos os cenários, mas quando se compara a degradação do sistema provocado pelos *freeriders* sem habilitar as políticas, mostrado nas tabelas 4.4 e 4.5, a adoção das políticas passa ser vantajoso, pois em todos os cenários a sobrecarga no sistema provocado pelas políticas é inferior a sobrecarga provocada pelos *freeriders*.

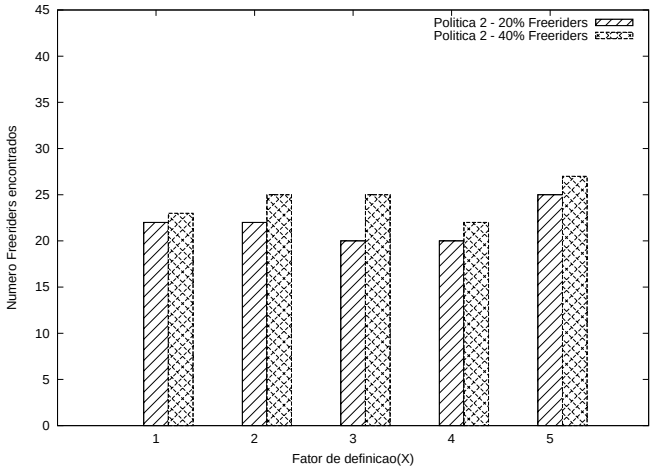


Figura 4.7: Fator de definição (Z) - 50 pares

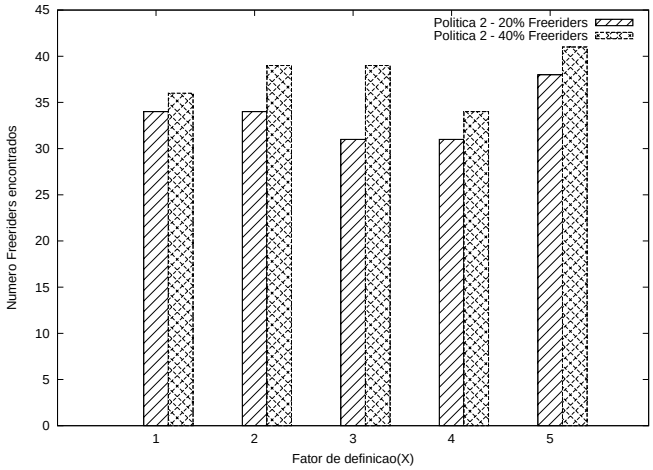


Figura 4.8: Fator de definição (Z) - 75 pares

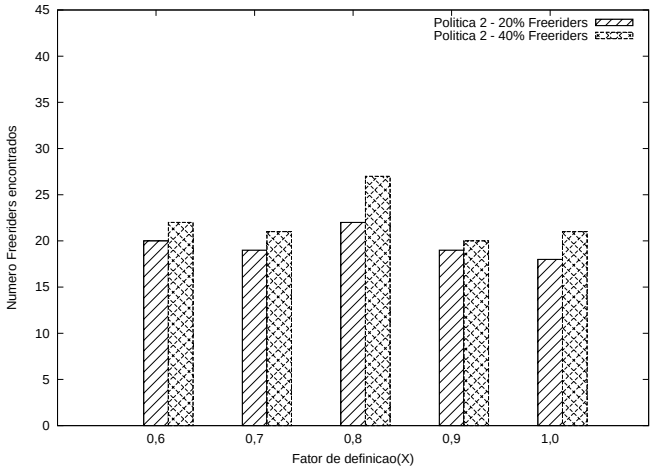


Figura 4.9: Limite de contribuição (f) - 50 pares

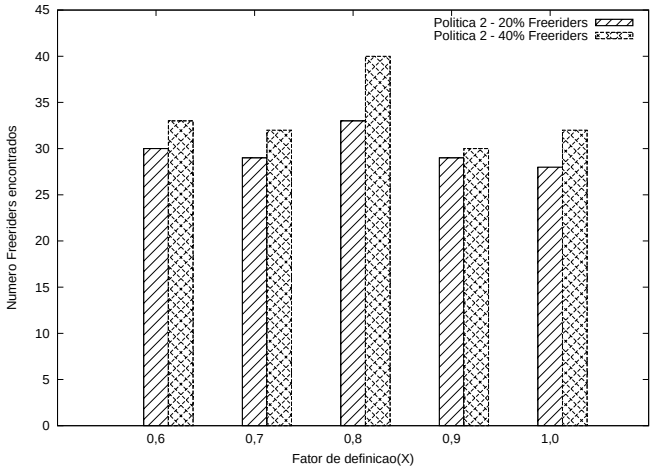


Figura 4.10: Limite de contribuição (f) - 75 pares

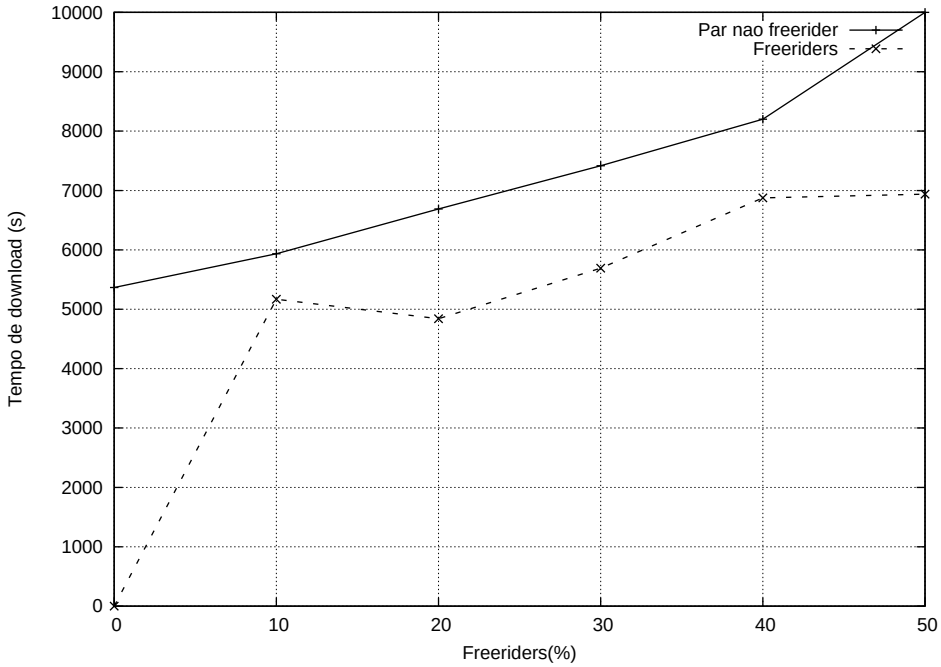


Figura 4.11: Ambiente com 50 pares

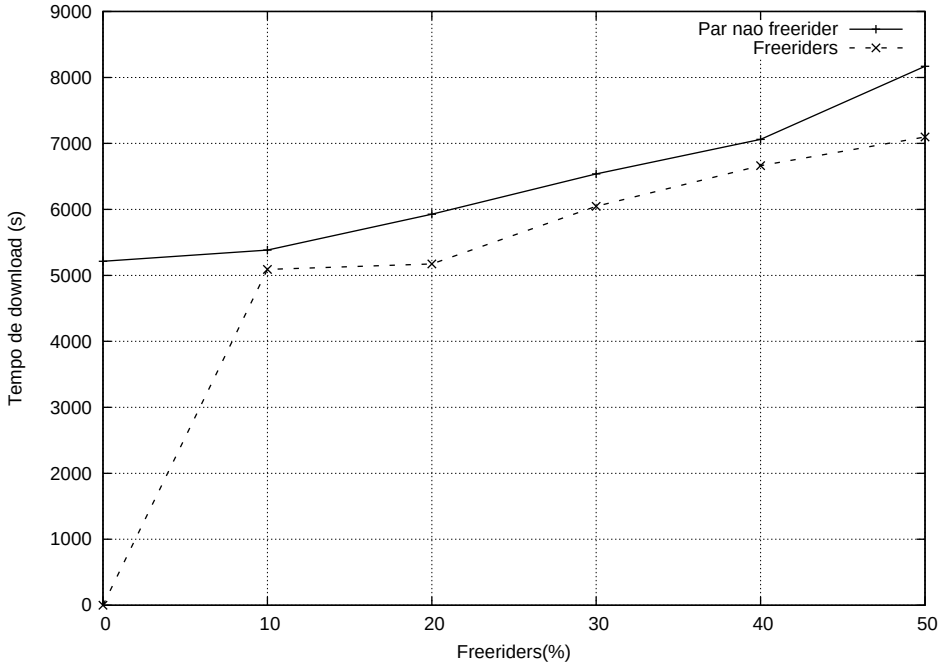


Figura 4.12: Ambiente com 75 pares

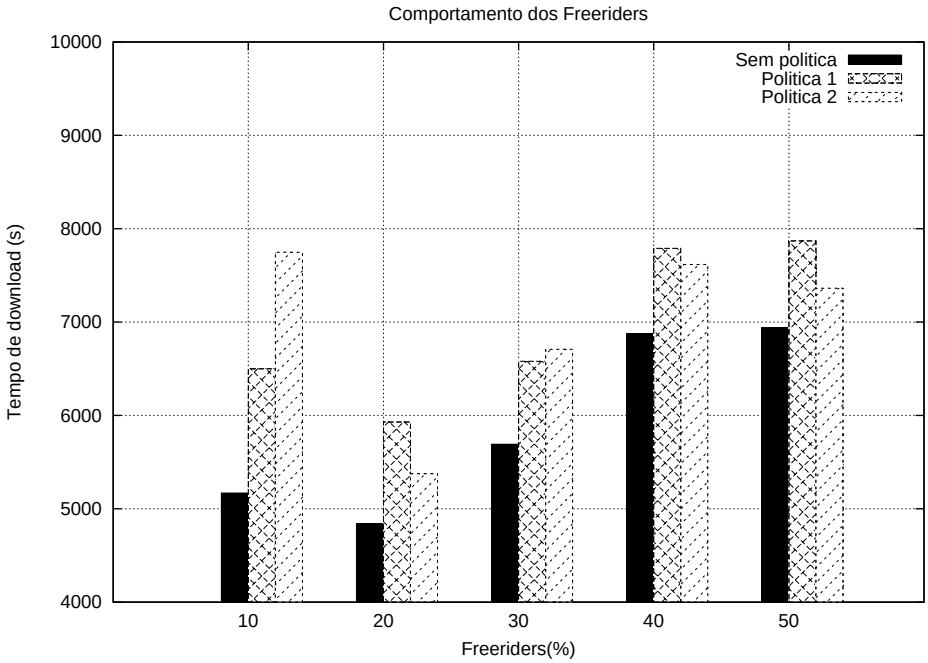


Figura 4.13: Simulação com 50 pares

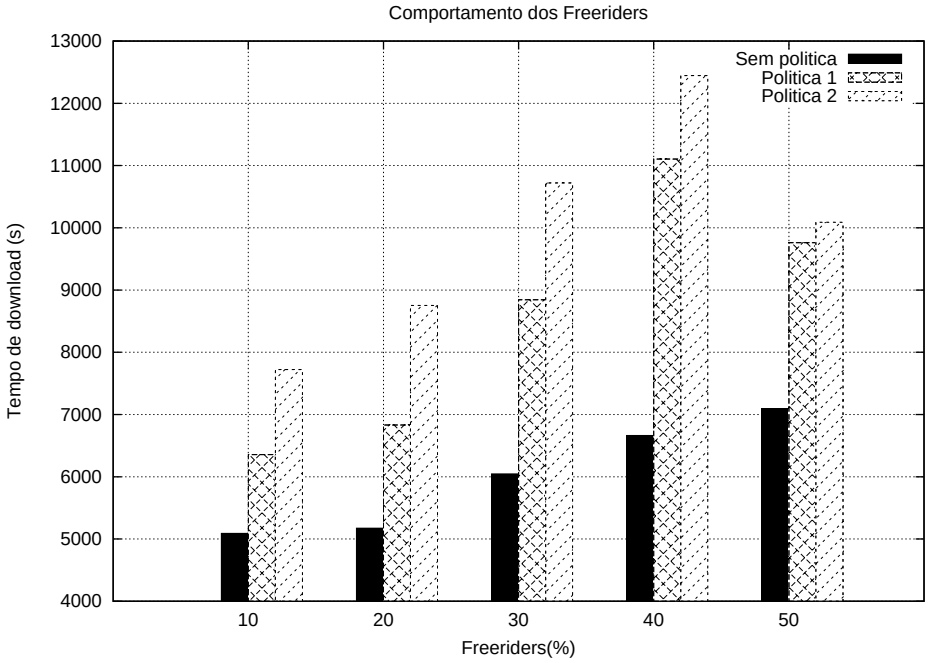


Figura 4.14: Simulação com 75 pares

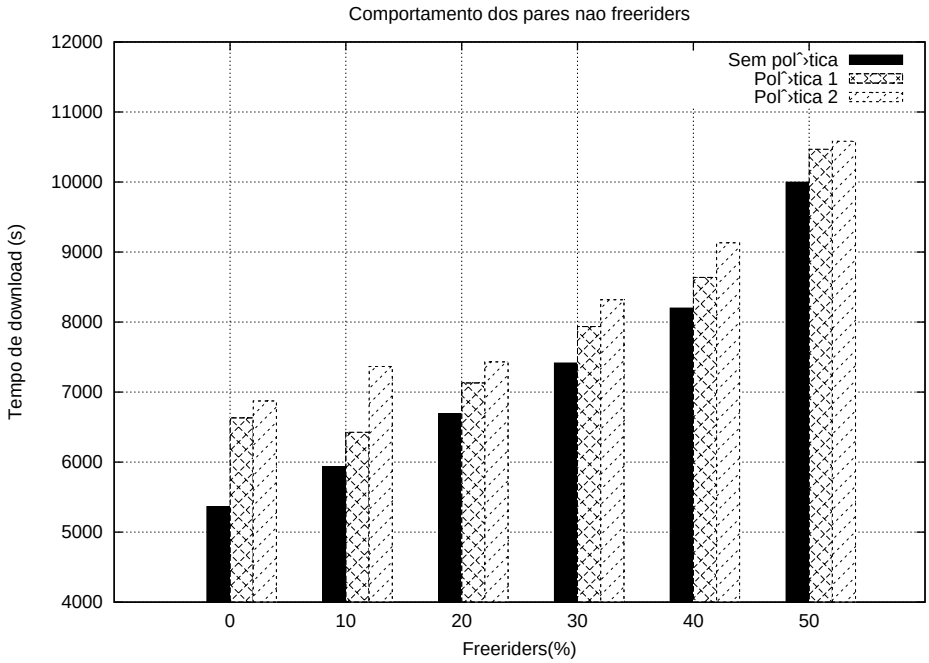


Figura 4.15: Ambiente com 50 pares

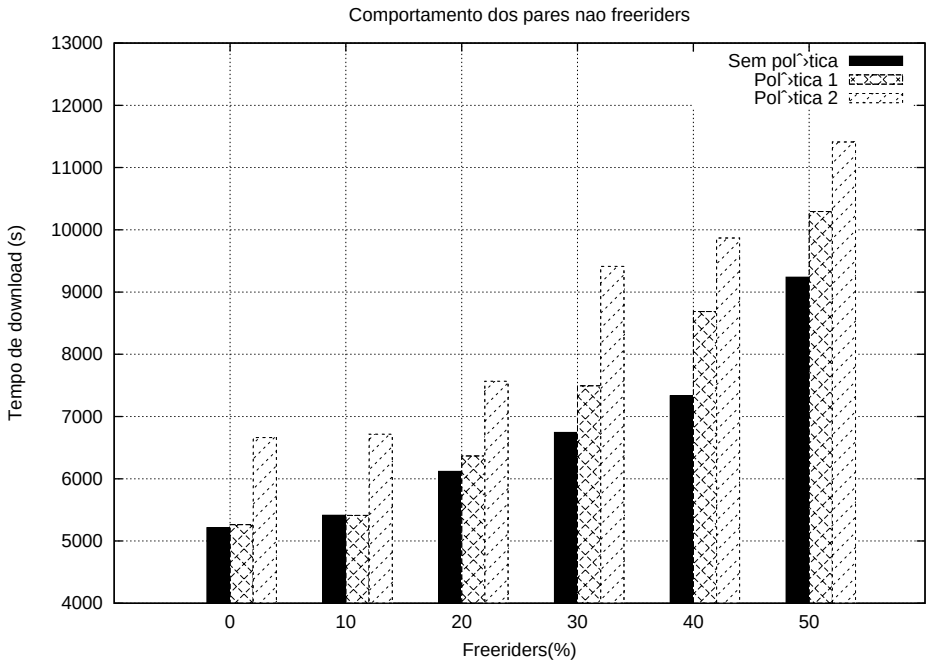


Figura 4.16: Ambiente com 75 pares

5 Conclusão

Este trabalho apresentou duas propostas de política para lidar com *freeriders* em ambiente BitTorrent. A primeira política é usada para pontuar pares que pouco contribuem dentro de um *swarm* e quando esses pares alcançam uma determinada pontuação, são marcados como *freeriders*. A segunda política usa a mesma lógica de pontuação do primeira, mas nesta política quem recebe pontos são os pares que mais contribuem no *swarm* e quando alcançam determinada pontuação são marcados como Bons Contribuidores.

A partir dos resultados apresentados neste trabalho, é possível concluir:

- os *freeriders* degradam o sistema, aumentando o tempo médio de *download* de todos os pares da rede;
- o algoritmo *optimistic unchoking* não é capaz de lidar adequadamente com os *freeriders*;
- é possível implementar mecanismos simples de combate aos *freeriders* nos algoritmos *optimistic unchoking* e *choking*;
- com as políticas habilitadas, a degração do sistema é inferior a degradação do sistema sem políticas e com os *freeriders*;
- a Política 1 teve resultados melhores do que a Política 2 no combate aos *freeriders* na maioria dos casos.

Finalmente, de acordo com a análise feita neste trabalho, as políticas de combate aos *freeriders* se mostraram eficientes apesar de trazer uma carga de trabalho maior ao sistema. As mudanças nos algoritmo *choking* e no método *optimistic unchoking* são viáveis e podem contribuir na melhoria do protocolo.

São propostas de trabalhos futuros:

- A implementação de cenários híbridos com pares usando a política 1, pares usando a política 2 e pares sem nenhuma política em um mesmo *swarm*;
- A implementação das duas políticas em clientes BitTorrent para teste no PlanetLab[1];

[1] http://www.planet-lab.org/

- Fazer uma verificação mais apurada dos parâmetros definidos nas duas políticas;
- Implementar as duas políticas no nível de camada de rede. Isto possibilitará calcular medidas de desempenho como perda e retransmissão de pacotes;
- Implementar as duas políticas na ferramenta Tangram II [5] para possibilitar o cálculo de medidas em estado estacionário e em estado transiente.

Referências Bibliográficas

[1] Mario Barbera, Alfio Lombardo, Giovanni Schembra, and Mirco Tribastone. A markov model of a freerider in a bittorrent p2p network. In *IEEE Globecom*, 2005.

[2] B. Cohen. Incentives build robusteness in bittorrent. In *P2PEcon*, 2003.

[3] Comitê Gestor da Internet no Brasil. Pesquisa sobre o uso das tecnologias da informação e da comunicação no brasil 2007. Website, 2008. http://www.cetic.br/tic/2007/indicadores-cgibr-2007.pdf.

[4] Kolja Eger. Simulation of bittorrent peer-to-peer (p2p) networks in ns-2. Website, 2010. http://www.tu-harburg.de/et6.

[5] Daniel Ratton Figueiredo. O modulo de simulacao da ferramenta tangramii. In *LAND-UFRJ*, 1999.

[6] David Harrison. Bittorrent enhancement proposals. Website, 2009. http://www.bittorrent.org/beps/bep_0000.html.

[7] S. Jun and M. Ahamad. Incentives in bittorrent induce free riding. In *P2PECON*, 2005.

[8] Arnaud Legout, Nikitas Liogkas, Eddie Kohler, and Lixia Zhang. Clustering and sharing incentives in bittorrent systems. In *SIGMETRICS*, 2007.

[9] Thomas Locher, Patrick Moor, Stefan Schmid, and Roger Wattenhofer. Free riding in bittorrent is cheap. In *HOTNETS*, 2006.

[10] Fabrício Murai and Daniel R. Figueiredo. Formacao de clusters em redes p2p por similaridade entre os nos. In *SBRC*, 2009.

[11] NS-2. The network simulator - ns-2. Website, 2010. http://www.isi.edu/nsnam/ns/.

[12] Dongyu Qiu and R. Srikant. Modeling and performance analysis of bittorrent-like peer-to-peer networks. In *SIGCOMM*, 2004.

[13] Axeld R. *The evolution of cooperation*. Basic Books, 1984.

[14] Alex Sherman, Angelos Stavrou, Jason Nieh, and Cliff Stein. Mitigating the effect of free-riders in bittorrent using trusted agents. In *Computer Science Technical Report Series*, 2008.

[15] Kyuyong Shin, Douglas S. Reeves, and Injong Rhee. Treat-before-trick: Free-riding prevention for bittorrent-like peer-to-peer networks. In *IPDPS '09: Proceedings of the 2009 IEEE International Symposium on Parallel&Distributed Processing*. IEEE Computer Society, 2009.

[16] C. Shirky. What's p2p and what's not. In *Internet Publication*, 2000.

[17] Michael Sirivianos, Jong Han Park, Rex Chen, and Xiaowei Yang. Free-riding in bittorrent networks with the large view exploit. In *IPTPS*, 2007.

[18] Luiz Fernando G. Soares, Guido Lemos, and Sérgio Colcher. Network computers. In *LAN, MANs, WANs and ATM*, 1995.

[19] Andrew Tanenbaum and M. V Steen. Distributed systems : Concepts and design. In *4th Edition*, 2005.

[20] Manaf Zghaibeh and Fotios C. Harmantzis. Revisiting free riding and the tit-for-tat in bittorrent: A measurement study. In *Peer-to-Peer Networking and Applications*, volume 1, 2008.

A Simulador NS-2

Neste capítulo são apresentados instalação e estrutura de diretórios e classes do simulador usado neste trabalho, o NS-2.

A.1 Estrutura de diretórios

O projeto NS-2 pode ser baixado em [11], sua instalação é simples e bem documentada. Abaixo é mostrado o local de instalação do NS-2 depois de implementado. Neste projeto o NS-2, versão NS-ALLINONE 2.29, foi instalado em um servidor Intel Core2 Duo de 2GHz com 2 GB de memória com sistema operacional Linux OpenSuse 10.3 e Kernel 2.6.19 . Abaixo na Figura A.1 segue a visão da estrutura de diretórios do NS-2 após ser instalado.

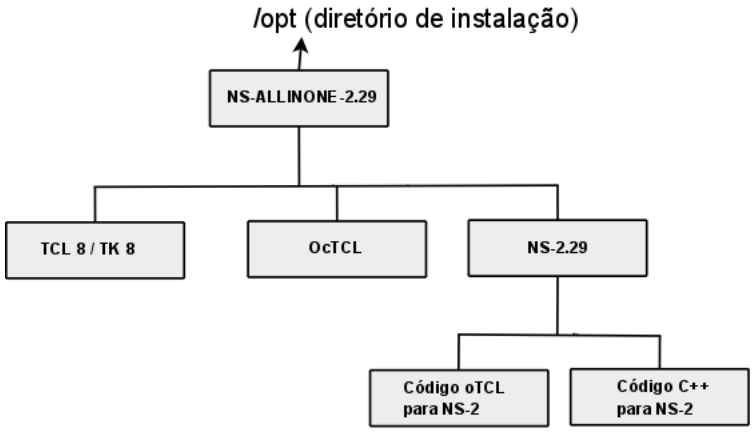


Figura A.1: Estrutura de diretório NS-2

No sub-diretório /opt/ns-allinone-2.29/ns-2.29/ fica toda codificação em C++ dos componentes de rede, protocolos, agentes, geradores de tráfego, agendador de tarefas do NS-2, como também qualquer implementação do usuário deverá ser instalada neste diretório.

A.2 Diagrama de classes do NS-2

Este tópico dá uma dimensão do tamanho do projeto NS-2, a quantidade de linhas de programação e de suas classes. Classes em C++, usada na criação de objetos e base

da programação orientada objeto, o NS-2 é construído com duas linguagens orientada a objetos (já mencionada na seção anterior) o oTCL e o C++ . Ambas com grande poder e flexibilidade na construção de classes, poliformismo, encapsulamentos entre outras. O NS-2 tem uma estrutura de classes bem definida e bem documentada, em [11] todas as classes ,heranças, funções membros e variáveis membros são documentadas. Abaixo será destacada as principais:

1. Tcl

   Esta classe encapsula uma instância do interpretador OTcl e fornece métodos para acessar e se comunicar com o interpretador. Seus métodos são relevantes para programador NS2 que está escrevendo código C++.

2. TclClass

   Esta classe é uma classe virtual pura. Classes derivadas desta classe fornecem duas funções: Construir o espelhamento entre classes do ambiente interpretado para o ambiente compilado e fornecer métodos para instanciar novos TclObjects.

3. TclObject

   Classe base para a maioria das outras classes no tanto no ambiente interpretado quanto no compilado (C++).Cada objeto na classe TclObject é criado pelo usuário no interpretador (oTCL). Um objeto sombra equivalente é criado no ambiente compilado (C++). Os dois objetos estão associados .

4. TclCommand

   Esta classe provê apenas o mecanismo de ns para exportar comandos simples para o interpretador.

5. Handler

   Esta classe é responsável por gerenciar objetos que necessitam de temporizadores.

Nas Figura A.2,A.3 e A.4 uma visão das principais classes do NS-2.

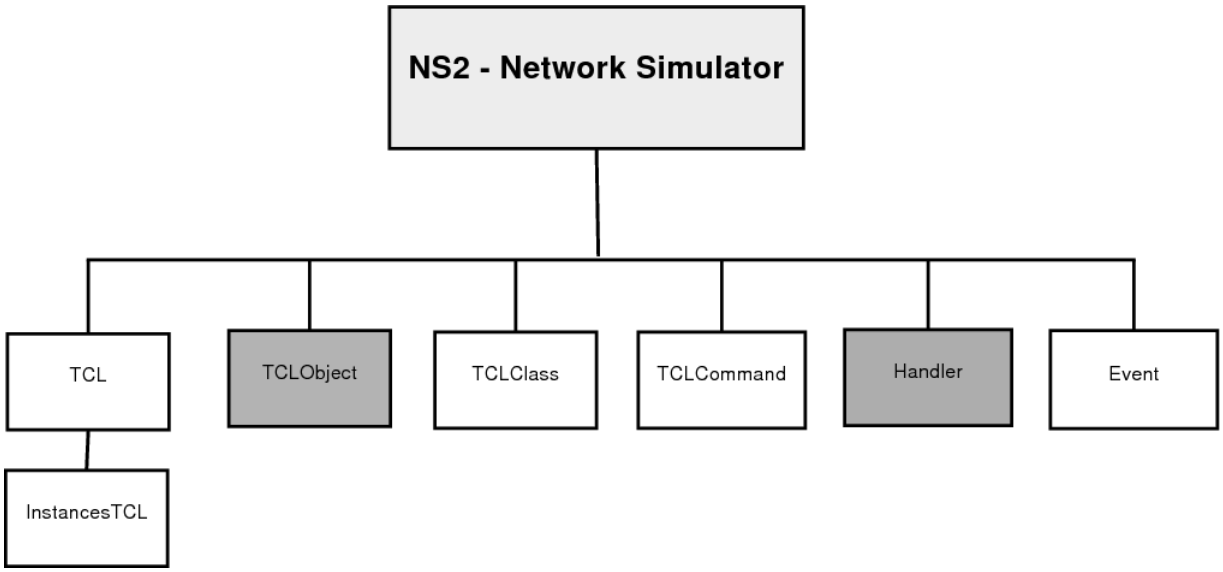


Figura A.2: Estrutura de Classe principal do NS-2

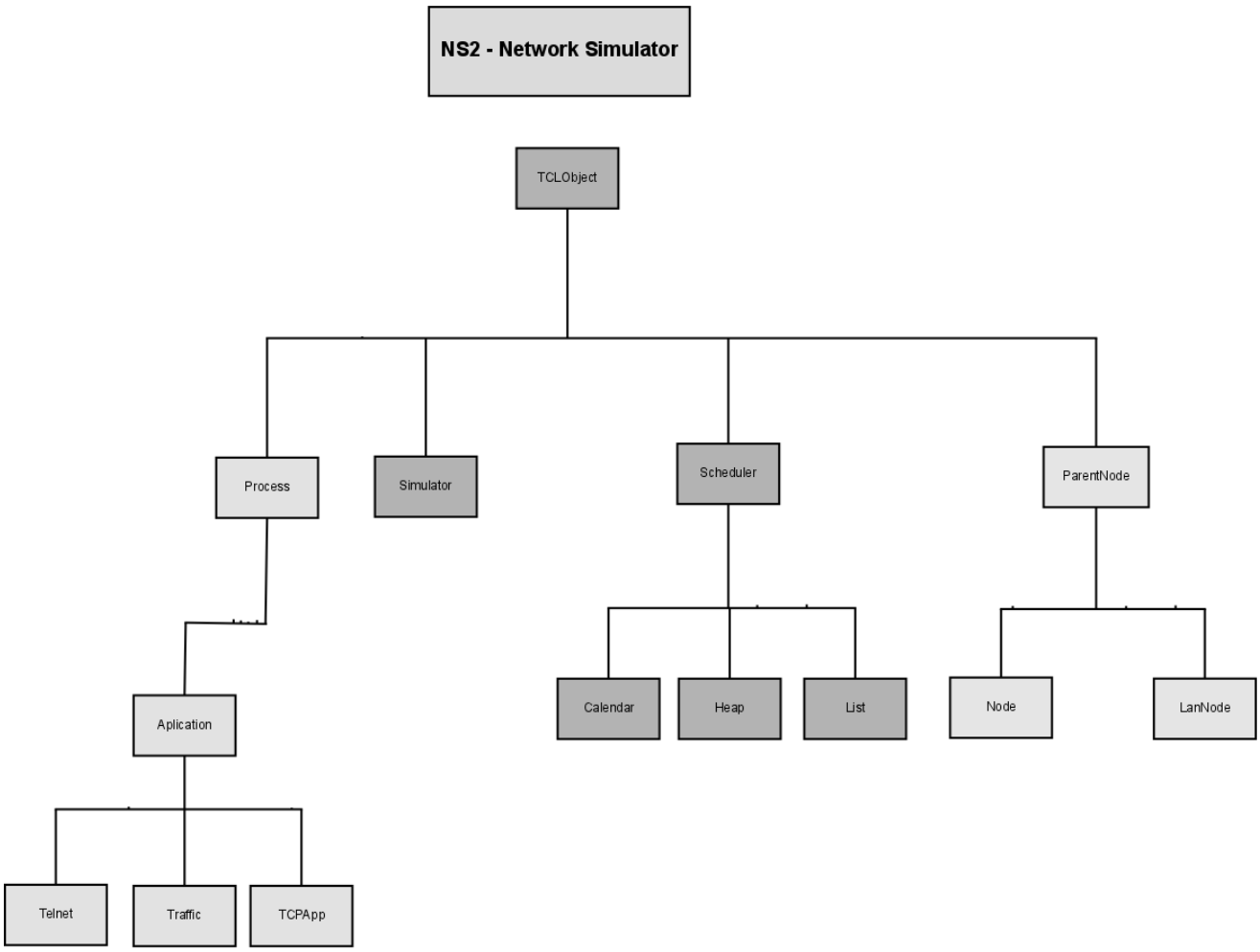


Figura A.3: Estrutura da classe TclObject

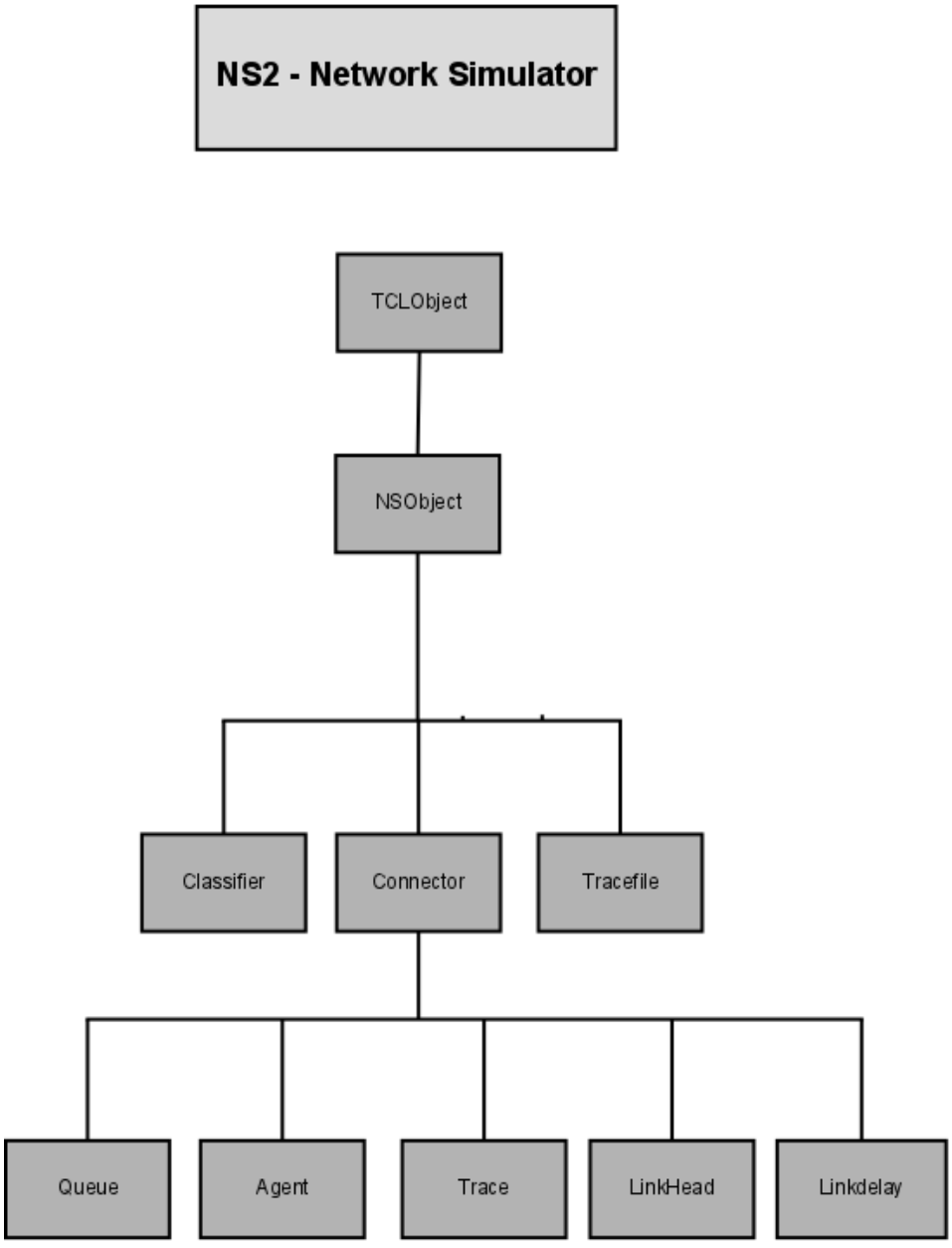


Figura A.4: Estrutura da classe NSObject

B Protocolo BitTorrent no NS-2

Neste capítulo são apresentados as características, instalação e implementação do algoritmo base para este trabalho.

B.1 Estrutura de diretórios do algoritmo

Neste trabalho, toda a implementação do algoritmo BitTorrent e suas modificações foram instaladas dentro de um sub-diretório do NS-2. A figura B.1 apresenta uma visão do NS-2 com o BitTorrent implementado.

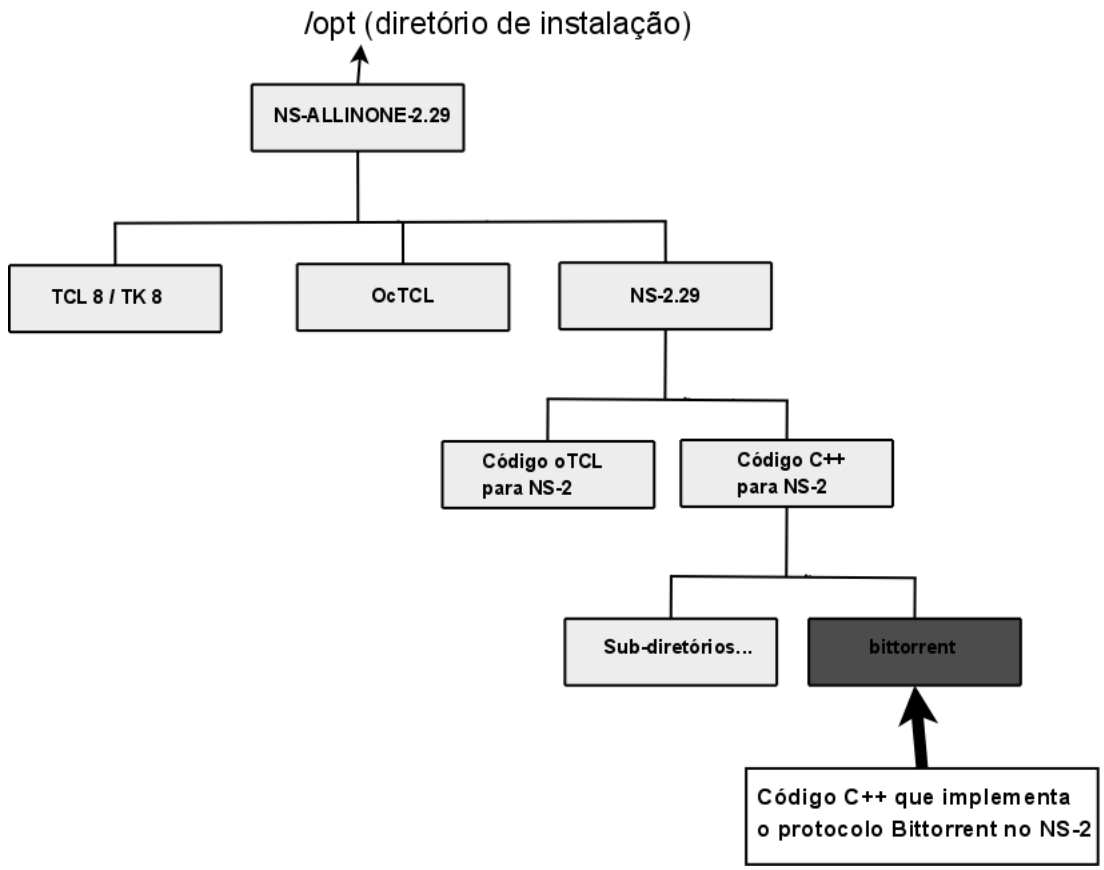


Figura B.1: Estrutura de diretório do algoritimo BitTorrent no NS-2

B.2 Estrutura de classes do algoritmo

B.2.1 Principais Classes

Este tópico aborda a estrutura de classe do algoritmo do BitTorrent usado nas simulações. O algoritmo do sr. Eger foi desenvolvido para ser usado em um simulador NS-2, ou seja, não tem todas as funcionalidades e complexabilidade do algoritmo BitTorrent real, mas para este trabalho se encaixa de maneira razoável. Abaixo seguem detalhes das principais classes na implementação. Existem ao todo 19 classes na implementação BitTorrent para o NS-2:

1. **BitTorrentAPP**

   Uma das classes mais importantes, responsável pela criação do objeto cliente BitTorrent, por todo mecanismo de conexão entre pares, troca de pedaços , algoritmo de seleção dos pares e pedaços, troca de mensagens entre outras. Nesta classe também fica todo controle das conexões TCP e de largura de banda usadas no algoritmo.

2. **BitTorrentAppFlowlevel**

   Esta classe herda muitas funções e variáveis da classe BitTorrentAPP, a única diferença é na conexão entre os pares, pois na classe acima é levada em conta toda negociação a nível camada de rede, já aqui a negociação fica a nível da camada de aplicação, ou seja, o objeto cliente BitTorrent criado irá ter suas funcionalidades consideradas apenas a nível de aplicação. Neste trabalho foi escolhido trabalhar com conexão orientada à camada de aplicação.

3. **BitTorrentConnection**

   Esta classe é responsável por todo controle de conexão a nível da camada de rede entre os pares. Para este trabalho não foi utilizada.

4. **BitTorrentData**

   Esta classe é responsável por criar os objetos mensagens do protocolo BitTorrent, como:

   (a) Bitfield;

   (b) Cancel;

   (c) Choke;

   (d) Handshake;

   (e) Have;

   (f) Interested;

(g) KeepAlive;

(h) Not_Interested;

(i) Piece;

(j) Request;

(k) Unchoke;

5. **BitTorrentTracker**

   Esta classe é responsável pela criação e implementação do objeto Rastreador no simulador. Lembrando que não foram implementadas todas as funcionalidades de um Rastreador real para efeito de simplificar a simulação. Suas principais funcionalidades são : Criar e gerenciar o enxame (Swarm), receber os registros dos pares que desejam ingressar no enxame e mandar lista de pares para os novos pares que ingressam no enxame.

6. **BitTorrentTrackerFlowlevel**

   Esta classe herda funções e variáveis da classe BitTorrentTracker. As principais diferenças são os métodos que auxiliam os pares na comunicação entre eles e na implementação de um método próprio para sorteio e escolha dos pares que serão Freeriders.

7. **CheckFreeriderTimer**

   Esta classe foi implementada por este trabalho , ela herda funcionalidades do Handler, classe responsável pelos temporizadores no NS-2. CheckFreeriderTimer é responsável por chamar o método que busca por Freeriders dentro do enxame.

8. **CheckGoodPeersTimer**

   Esta classe foi implementada por este trabalho, ela herda funcionalidades do Handler, classe responsável pelos temporizadores no NS-2. CheckGoodPeersTimer é responsável por chamar o método que busca por pares bins contribuidores dentro do enxame.

9. **ChokingTimer**

   Esta classe herda funcionalidades do Handler, classe responsável pelos temporizadores no NS-2. ChokingTimer é responsável por chamar método Choking, um dos mais importantes do algoritmo, uma de suas principais funções é controlar o "Tit-for-Tat" no enxame.

10. **ConnectionCloser**

    Esta classe é responsável pelo fechamento das conexões entre pares , tanto a nível de rede quanto a nível de aplicação.

11. **ConnectionTimeout**

    Esta classe é responsável por controlar o evento Timeout, que pode ocorrer entre conexões de pares dentro do enxame. Aqui estas conexões podem ser encerradas ou reiniciadas.

12. **KeepAliveTimer**

    Esta classe herda funcionalidades do Handler, classe responsável pelos temporizadores no NS-2. KeepAliveTimer é responsável por manter ativas conexões entre pares, enviando requisições "Handshake"multilaterais.

13. **LeavingTimer**

    Esta classe herda funcionalidades do Handler, classe responsável pelos temporizadores no NS-2. LeavingTimer é responsável por fechar todas as conexões dos pares e retirá-los do enxame.

14. **TrackerRequestTimer**

    Esta classe herda funcionalidades do Handler, classe responsável pelos temporizadores no NS-2. TrackerRequestTimer é responsável por manter os pares periodicamente trocando dados com Rastreador (Tracker) do enxame.

15. **UploadQueueQuery**

    Classe responsável por controlar a quantidade de envio de dados de um par a outro. Também implementa um controle de fila para dados que não foram enviados ou recebidos.

A figura B.2, mostra o diagrama de classes da implementação do BitTorrent no NS-2.

As classes mais usados nas modificações propostas por este trabalho herdam diretamente de classes do NS-2. Ex.: BitTorrentTracker herda TCLObject.

B.2.2 Principais Métodos

Na implementação do protocolo BitTorrent no NS-2 exitem centenas de funções membros espalhadas pelas dezenove classes do projeto. Abaixo segue as classes com seus principais métodos usados por este trabalho, todos inerentes a nível de camada de aplicação:

1. **Classe BitTorrentAppFlowlevel**

    Esta classe, conforme citado anteriormente, herda variáveis e funções membros da classe BitTorrentApp , a maioria não pertence ao escopo deste trabalho. As funções abaixo citadas são implementadas através de polimorfismo para classe BitTorrentAppFlowlevel, referente apenas a camada de aplicação foco deste trabalho. São estes:

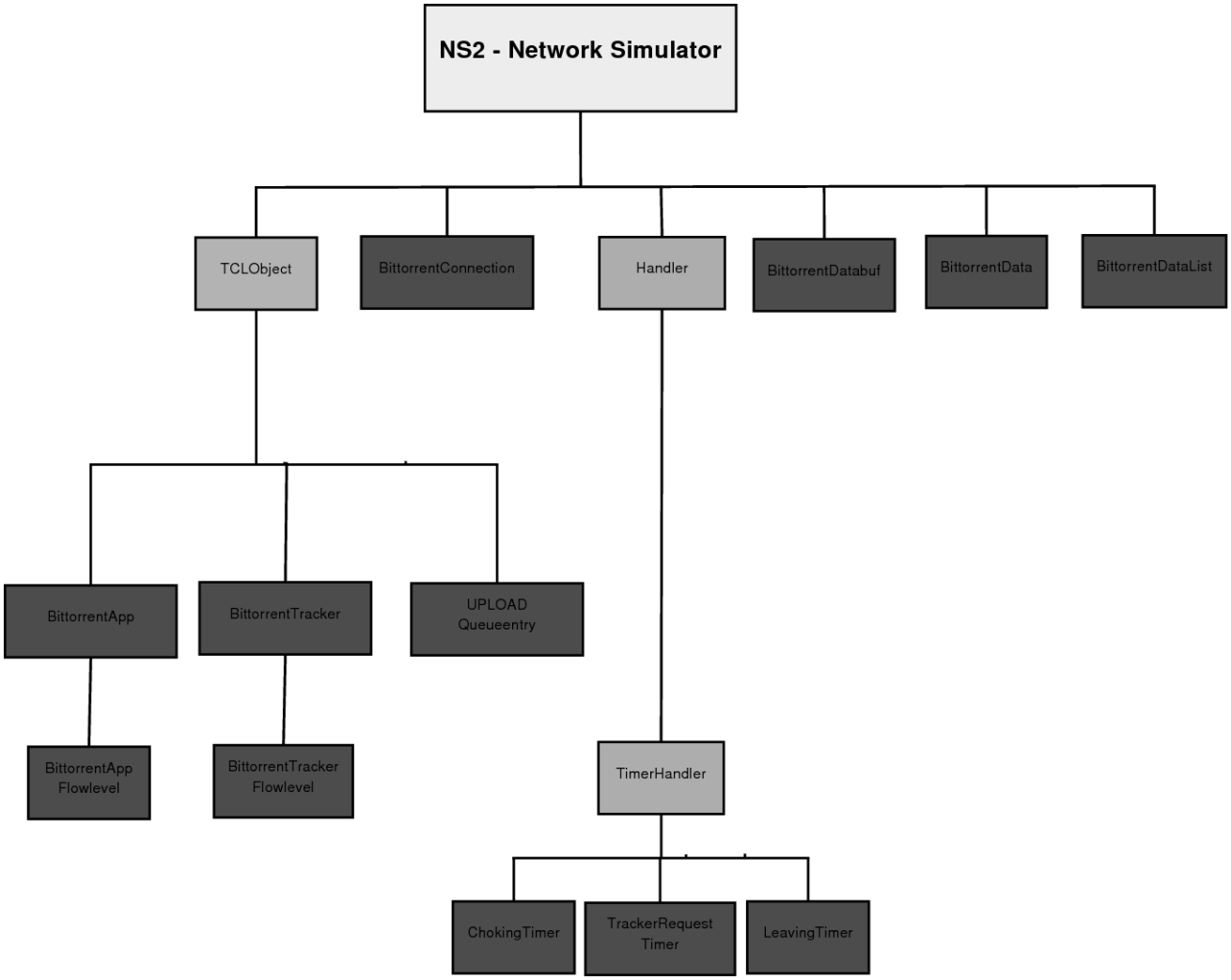


Figura B.2: Diagrama de classe da implementação BitTorrent no NS-2

(a) **start()**

Este método é responsável por criar o objeto Peer. Aqui as principais variáveis são inicializadas.

(b) **stop()**

Este método retira o objeto Par do sistema. Aqui são geradas todas as informações sobre a troca de dados do objeto durante o tempo que esteve no sistema.

(c) **get_ids_from_tracker()**

Este método é responsável por requisitar ao objeto Tracker a lista de pares que estão no sistema. Esta requisição é agendada para acontecer periodicamente.

(d) **check_connections()**

Este método verifica a quantidade de conexões que estão abertas no objeto

Peer. Aqui pode ser controlado o comportamento de cada par no atedimento a requisições por pedaços de dados.

(e) **handle_recv_msg(BitTorrentData,int)**

Este método é responsável por interpretar cada mensagem recebida pelo objeto Peer na troca de informações com seus vizinhos.

(f) **make_new_peer_list_entry(long)**

Este método é reposável por montar a lista de vizinhos que cada objeto Peer precisa para iniciar troca de dados.

(g) **send(BitTorrentData,int)**

Este método é responsável por enviar mensagens do objeto Peer para seus vizinhos.

(h) **check_choking()**

Este método é responsável pelo funcionamento do algoritmo do "Tit-for-Tat", bem como a montagem da fila de pedações que cada Peer poderá enviar ao ser requisitado por seus vizinhos.

A figura B.3 apresenta a classe com todas suas funções membros e variáveis, aqui são demonstrados detalhes da classe BitTorrentAppFlowlevel com todas suas variáveis e funções menbros.

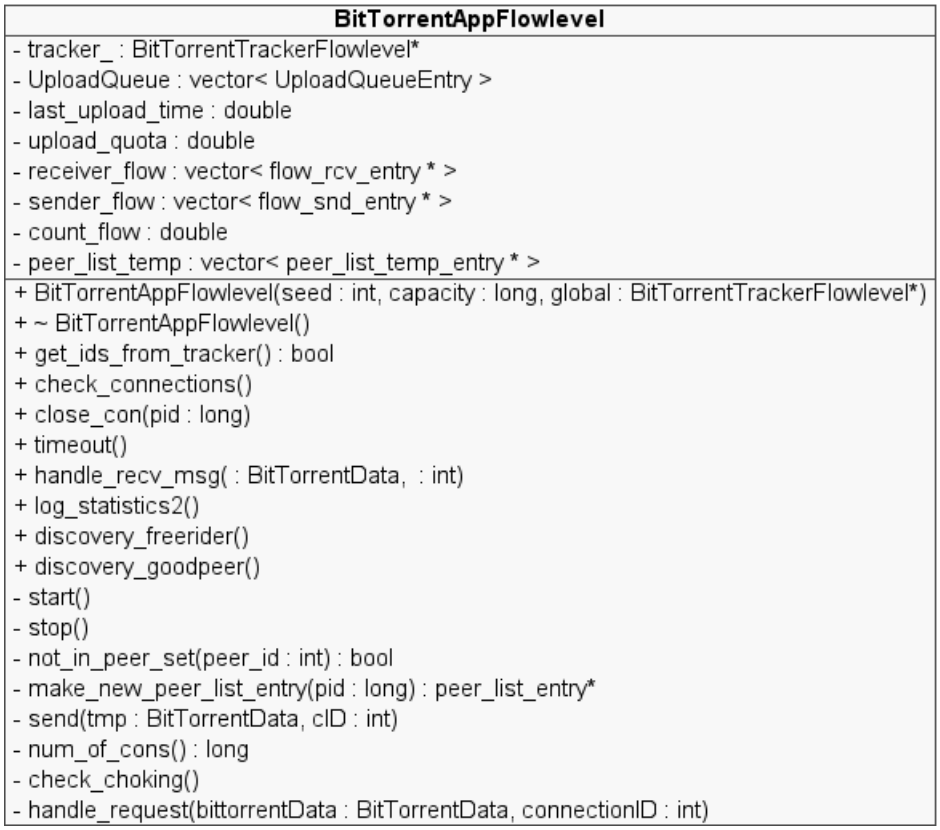


Figura B.3: Diagrama da classe BitTorrentAppFlowlevel

2. **Classe BitTorrentTrackerFlowlevel**

Como na classe acima, BitTorrentTrackerFlowlevel, também herda funções membros e variáveis da classe base BitTorrentTracker, mas para este trabalho somente os métodos referentes a camada de aplicação são alvo de detalhes:

(a) **reg_peer(BitTorrentAppFlowlevel)**

Este método é responsável por registrar todo objeto Peer que ingresse no sistema, dentro do objeto Tracker.

(b) **get_peer_set(int)**

Este método monta uma lista com número limitado de objetos Peer que estão no sistema no momneto.

(c) **sendmsg(BitTorrentData,long,long)**

Este método tem como função em auxiliar a troca de mensagens entre objeto Peer e seus vizinhos.

(d) **close_con(long,long)**

Este método é responsável por auxiliar o fechamento de uma conexão entre o objeto Peer e seus vizinhos.

A figura B.4 apresenta a classe com todas suas funções membros e variáveis.

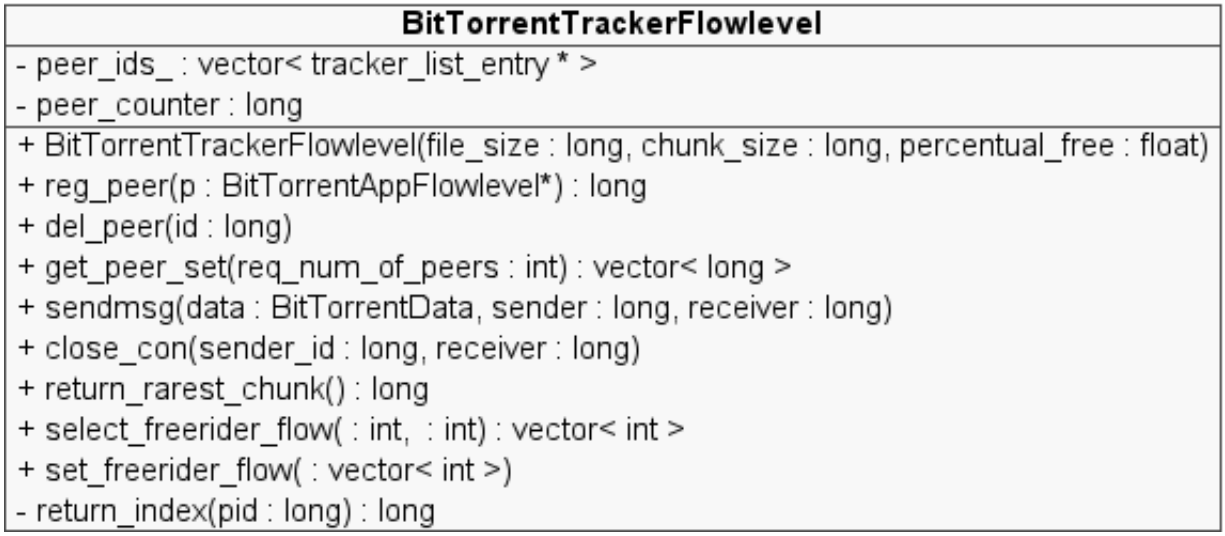


Figura B.4: Diagrama da classe BitTorrentAppTrackerFlowlevel

Na figura acima é demonstrada detalhes da classe BitTorrentAppFlowlevel com todas suas variáveis e funções menbros.