A BitTorrent module for Peersim*

Fabrizio Frioli, Michele Pedrolli
{fabrizio.frioli, michele.pedrolli} <at> studenti.unitn.it

February, 2008

Abstract

This document presents the implementation of a BitTorrent module for Peersim, a P2P simulation environment written in Java. It is also provided a basic introduction on the BitTorrent protocol, an explanation on how to configure and use the module and a simple analysis.

1 Introduction

Peersim is a simulation environment for P2P protocols which offers extreme scalability and support for dinamicity. It supports two different simulation models: the *cycle-based* model and the *event-based* one.

Peersim is highly customizable by adding different extendable components, which can be set up through a simple configuration mechanism.

Peersim is developed under the BISON project of the University of Bologna. It is written in Java and released to the public under the GPL open source license.

2 BitTorrent protocol

2.1 Share a file

To share a file using BitTorrent you have to publish somewhere in the web a `.torrent` file containing information such as the file name, its length, hashing info and the *tracker* address.

This special node of the overlay network allows the peers to know other peers that are already interested to the file. Upon request of a peer, the tracker replies providing a set of 50 random nodes which may have a part or the whole file. This set of neighbors can increase if the local peer is in the *peerset* (i.e. the set of neighbors returned by the tracker which are interested to download the file) of another node. If the number of neighbors of a node decreases under 20, this one contacts the tracker to have a new set of peers. A peer can have at most 80 neighbors.

Initially the node who decides to share something has the whole file, then to correctly share a file there must be at least a copy of every piece belonging to it in the network, otherwise the file's download will be never completed. A node with all the pieces of a file is called *seeder* otherwise is called *leecher*. The set of peer sharing the same file is called *swarm*.

*http://peersim.sourceforge.net

2.2 File structure

Inside a peer, a file is split in *pieces* of 256 KB each and every piece is split in 16 sub-pieces (or *blocks*) (every block is 16 KB). If the download of a piece crashes, the piece has to be entirely re-downloaded. Splitting the file in pieces enhances the precision in the estimate of the download time and allows a fast download of the block also from peer with low bandwidth. Different blocks belonging to the same piece can be downloaded from different peers. After the download of a piece has been completed, the peer must send a HAVE message to its neighbors.

2.3 Choice of the piece

2.3.1 Strict Priority

This rule states to download the blocks that belong to a piece until this one is completed. There must be 5 outstanding requests of blocks. When a block is downloaded, another request is sent. When all the blocks of a piece have been requested, an INTERESTED message for a new piece is sent.

2.3.2 Rarest First

This rule lead the choice of the next piece to download. It states that the less frequent piece in the *peerset* has to be downloaded first. This ensures that with only a seeder in the network, every peer doesn't download the same piece simultaneously degrading the performance.

Every peer maintains a set with the number of copies for every piece in its neighbors list. This set is called *rarest pieces set* and is updated every time a piece is downloaded in the whole *peerset* or a neighbor is removed from it. An exception to this rule is used at the beginning of the download, when the local node joins to the network for the first time: when the local node has nothing to offer to the other peers it is important that it has any piece as soon as possible. If the rarest piece was choosed, the download would be slow since that piece, as rare, would be present probably in only one node and thus it wouldn't

be possible to download different sub-pieces from different nodes. To avoid this situation, for the initial four pieces of peer the choice is done randomly.

2.3.3 Endgame mode

This policy is used in the final stage in the download of a file. When a request for every remaining block has been sent, the local node sends a request for the blocks in the pending request queue to every node having that blocks. When a block is downloaded a *cancel* message is sent to the other peers to which the local node has previously sent the request.

2.4 Chocking algorithms

This algorithm manages the downloading rates in the network. Every node tries to maximize its downloading rate and does this by using the "tit-for-tat" philosophy. This technique, born from the games theory, states that a player (in our case a peer) will be cooperative with another one only in the case this one will do the same. It is a principle of collaboration that benefits components interested to the exchange of informations.

The *chocking* is a temporary refuse of uploading pieces to a peer. The main idea is to provide a correspondence between upload and download traffic, achieving connections in both the directions.

2.4.1 Unchoking algorithm

The goal is to find four nodes to which do the upload. The decision is taken given a list with the average downloading rates from every neighbor in the last 20 seconds, in decreasing order. The top 3 peers are chosen. The choice about which peer choke or unchoke is taken every 10 seconds.

The remaining peer is chosen with an optimistic unchoke: the node is chosen randomly every 30 seconds from the list of neighbors.

If the local peer is a seeder, then the nodes to unchoke are chosen in a list with the average uploading rates (instead of downloading rates) to every peer in decreasing order. This ensures that the file will be shared faster.

3 Implementation

The BitTorrent module in Peersim is implemented as an event driven protocol. The events defined are the messages exchanged in the real protocol, plus some other timeout event. Every event is an instance of the `SimpleEvent` class which has an integer value to indicate the type of event. Every message is instead an instance of `SimpleMsg` class that inherits the `type` field from `SimpleEvent` and is in turn overridden to obtain message classes with particular requirements (e.g. sender, array, additional fields etc.). Now we are going to describe the particular messages and events exchanged in the protocol.

Messages:

`KEEP_ALIVE` This message is sent by a peer to its neighbor that hasn't been seen for over two minutes. It has a flag indicating if the message is either a request or a reply. If the neighbor is still alive, when it receives this message, it will replies with a `KEEP_ALIVE` response. If the local node doesn't receive a reply, it will remove the neighbor.

`CHOKE` This is sent to a neighbor to notify that its status, respect to the local node, from *unchoked* became *choked* and it can no more request blocks.

`UNCHOKE` From the *choked* status, receiving this message a node become *unchocked* respect to the sender.

`INTERESTED` Sent by a peer to all the neighbors that have the piece specified in.

`NOT_INTERESTED` Sent immediately after an `HAVE` message has been received, to the neighbors to which the local node has previously sent a `INTERESTED` for the piece just completed.

`HAVE` This is sent by a local node to any of its neighbors to inform that it has a new piece just completed available for sharing. It has a field containing the piece ID.

`BITFIELD` There are four types of this message:

1. Request with ack.
2. Response with ack.
3. Response with nack.
4. Request with nack.

The first one, is sent by a peer to the nodes of the list returned by the tracker to ask them their file status and to be inserted in their list of neighbors. The message is an instance of `BitfieldMsg` and has the `isRequest` field set to 1, the `ack` field set to 1 and the `array` field contains the file status of the sender. Sending this message the node increases the value of `nBitfieldSent`. This value is kept at the sender to avoid that it accepts requests from nodes and than, when the responses to its requests arrive, it has no more place to accept the replying nodes.

The *response with ack* message is used as a "positive" reply to the reception of the previous message. It confirms that the receiver was successfully added to the neighbors set of the sender and has in attachment its file status. `isRequest` is 0, `ack` is 1 and the `array` contains the file status of the sender.

If the node that received the *request with ack* message could not add the sender (e.g. it has the list of neighbors full), it replies with a *response with nack* message. At the reception of this reply, the node that requested the addition, decreases the value of `nBitfieldSent`. In this situation the `ack` field values 0.

The last one is sent when a message from an unknown sender arrived. This could happen due to an out of order delivery of the transport protocol. At the reception of this message, the receiver replies with a *Bitfield response + ack*, indeed it has certainly the sender in its neighbors set since previously it has sent a message to it.

`REQUEST` This is the message used by the local peer to request to its neighbor a block to download. It is sent only to the neighbors that have unchoked the local node and have the piece in question. It has an integer value containing an encoding (an unique ID) of the block to request. This ID is obtained taking the number of the piece that the block belongs to, multiplying it for 100 and adding the index of the block in the piece. For example, the fourth block of the piece number 123 will have ID equal to 12304.

`PIECE` When a node receives a `REQUEST` message, It replies sending a `PIECE` message containing the block requested. The integer value contained is the ID, as explained above, of the block provided to the requester.

`CANCEL` Sent by the local node to the neighbors to which had previously sent a `REQUEST` message, to ask them to don't process the request, if they have not already fulfilled.

`TRACKER` This is the first message sent by the nodes at the beginning of the simulation (and the first message sent when a new node enters the network). This is to ask the tracker the list of 50 nodes that the local node has to contact to begin the sharing. This message is also sent to the tracker to get a new list of nodes, when the number of local neighbors decreases under 20.

`PEERSET` Sent by the tracker, contains the set of 50 nodes asked with a `TRACKER` message by a node. If the total number of nodes is greater than 50, the elements are chosen randomly.

Events:

`CHOKE_TIME` This is the event that occurs with the shortest timeout. Every 10 secs an instance of this event is inserted in the queue of events for every node in the network. At the occurrence of this event, the local node checks which neighbors it has to choke or unchoke accordingly with the choking algorithm.

`OPTUNCHK_TIME` Every 30 secs an optimistic unchoke is done.

`ANTISNUB_TIME` Used by the local node to check if it is snubbed by a node. It occurres every 1 minutes. If for over 1 minutes the local node doesn't receive any block from a node, it concludes that it is snubbed by that node and this one is going to be chocked until an optimistic unchoke.

`CHECKALIVE_TIME` Occurres every 2 minutes telling the local node to check if all its neighbors are alive. To do this, it sends a `KEEP_ALIVE` message to all the neighbors to which it didn't send anything for over 2 minutes. If in the next to minutes the neighbor doesn't respond, the local node assumes its neighbor as died.

`TRACKER_ALIVE` This is processed every 30 minutes to check if the tracker is still online: if not, no more `TRACKER` messages will be sent to it.

3.1 Code description

By the point of view of a peer, the *swarm* related to a file is the set of its neighbors. The structure containing the neighbor is the `cache` and its element are instances of `Neighbor` class. Every `Neighbor` maintains a link to the node in the network of the simulator, a status between *choke*, *unchoke* and *snubbed_by* and the interest status to a piece of the node. Moreover in this class is stored the last time the node sent a message or the local node sent one to it (used for the *snubbing* rule and the removal of neighbors).

All the neighbors are also referenced by an `Element` class that are contained in a structure always ordered by index of the node, called `byPeer`. This structure is used to improve the accessing speed to a node in the cache[1]. `Element` class contains also informations used by the choking/unchocking algorithm.

3.1.1 Start up

The network's life cycle starts when the nodes, joining the network, send a `TRACKER` message to the tracker to get the list of neighbors. Receiving this message the tracker prepares a list containing `peerset_size`[2] nodes.

Received the `PEERSET_MSG`, the local node send `BITFIELD req + ack` to every node contained in the list sent by the tracker, attaching its file status (which pieces it has). When the neighbor receives the message, it calls `addNeighbor()` that tries to add the node returning either *true* if the operation completed successfully or *false*. An addition could not be completed successfully in the case

[1] For more details please refer to the code

[2] For the configuration parameters see section 4.1

the node that is trying to add is already a neighbor (because is still contained in the list returned by the tracker further to a request of the local node due to a number of neighbors less than 20) or when the value of `nNodes + nBitfieldSent` is greater than `max_swarm_size`[3]. If the value returned by `addNeighbor()` is *true*, then:

- the distribution of the file in the local vision of the network (made by the neighbors) is updated with the status contained in the `BITFIELD` message just arrived
- the status of the remote node is set to *choked* and *not interested*.
- a reply `BITFIELD resp + ack` with the local status of the file is sent back to the neighbor.

If the returned value is instead *false* because the `BITFIELD` was sent by a node already neighbor, then the local node replies with `BITFIELD resp + ack` otherwise, if *false* due to too many `BITFIELD` requests, it replies with `BITFIELD resp + nack`.

When the remote node will receive a `BITFIELD resp + ack`, it will add the local node to its cache updating its swarm and decreasing the value of `nBitfieldSent`. Instead, if it will receive a `BITFIELD resp + nack` it will simply decrease the value of `nBitfieldSent` (in this way it can accept one node more).

3.1.2 Request of pieces

After the successfully insertion of 10 neighbors, the local node begins to send its `INTERESTED` status for pieces. It calls `getPiece()` that implements the rules explained in section 2.3 and return the ID of the piece to request. It sends an `INTERESTED` to all the neighbors which have the piece with that ID and updates the value of `lastInterested` consequently.

3.1.3 Downloading of pieces

When the `CHOKE_TIME` event occurs, the local node copies the interested neighbor in the `byBandwidth` array which is then sorted in decreasing order of average download rate. The three neighbors at the top will be unchoked. The average download rate from the neighbors is computed for the former 20 seconds. If the number of neighbors to unchoke using this rule is less than 3, the remaining nodes are chosen randomly in the array. When the three nodes are chosen, an `UNCHOKE` message will be sent to them.

The nodes begin to request *blocks* when the first `UNCHOKE` arrives. In this case, the local node updates the list of neighbors for which it was unchoked and fills the queue of `pendingRequest` by sending `REQUEST` messages with the blocks returned by `getBlock()` to the neighbor that has just unchoked it. If `getBlock()` returns -1, a block for a new piece must be request. In this case `getPiece()` is called, `INTERESTED` are sent and `lastInterested` is updated.

Receiving a `REQUEST` a node sends `PIECE`s as follow. If the local node is uploading less than 10 request, than it enqueues this request and begins to extract request previously enqueued until 10 blocks in upload or the queue is empty. At this point a question might arise: what does it mean to send `PIECE`s? Well, it simply means insert a `PIECE` message containing the ID of the request block in the scheduler of the receiver t_{down} time later, with

$$t_{down} = t_{tx} + t_{prop}$$

where t_{down} is the total downloading time of a block, t_{tx} is the transmission time an t_{prop} the propagation delay. The last one is given by `getLatency()` of the transport protocol, while the second one is in particular given by the following formulas:

$$\begin{aligned} t_{tx} &= \frac{\text{block size}}{\text{available bandwith}} \\ \text{available bandwidth} &= min\{\text{local\_rate, remote\_rate}\} \\ \text{local\_rate} &= \frac{\text{maxband\_l}}{\{\#\text{up\_l} + \#\text{down\_l}\}} \\ \text{remote\_rate} &= \frac{\text{maxband\_r}}{\{\#\text{up\_r} + \#\text{down\_r}\}} \end{aligned}$$

where:

- *maxband_*$\{l,r\}$ are respectively the broadband connection speeds of the local and remote node;
- *#up_*$\{l,r\}$ are the current number of blocks in upload respectively at the local and remote node;
- *#down_*$\{l,r\}$ are the current number of blocks in download respectively at the local and remote node.

Upon receiving a `PIECE` message, the local node checks if the piece has already been downloaded from another peer. To do this there is an array `pieceStatus` where the blocks of the current downloading piece (`currentPiece`, updated to `lastInterested` every time a piece is completed) are stored. If the just downloaded block belongs to the current piece then `pieceStatus` is updated, otherwise the block is enqueued and processed later. A `CANCEL` message is sent to neighbors to which I could have sent a `REQUEST` because they had such block. For every block arrived new `REQUEST`s are sent to fill the `pendingRequests` queue. The request of new blocks are done through the method `requestNextBlocks` that sends `INTERESTED` for new pieces if needed. If after the reception of the block a piece is completed, then an `HAVE` message indicating the piece completed is sent to all the neighbors which, upon

[3]This ensures that the local node does not accept more nodes than the actual available space in the cache, which is given by the difference between `max_swarm_size` and the summation of the number of nodes currently present with the number of invitations the local node did sending its `BITFIELD` messages.

the reception, will update them view of the file's distribution (`swarm` matrix). Moreover a `NOT_INTERESTED` message is sent to neighbors to which the local node had previously sent an `INTERESTED` for the piece completed.

When a neighbor will receive `CANCEL`, it will remove the request from the `requestToServe` queue, if present. Upon a `NOT_INTERESTED` the neighbor will set the interested status of the sender to 0 only if this status contains the number of piece for which the current message was sent, otherwise it means that another newer piece is interested (and thus the node could be chosen by the unchoking algorithm).

3.1.4 Sorting methods

In this module we use two sorting methods: `sortByPeer` and `sortByBandwidth`. The first one is an implementation of InsertSort algorithm. The choice of this algorithm is led by the need to have an array of nodes always sorted to find nodes as soon as possible. Since one node at a time is added, we thought this was an efficient solution. As said earlier, the `sortByPeer` method sorts `byPeer` array by *ID* every time a new node is added. `null` objects are put at the end of the array then, to search a neighbor, a dichotomic search is used.

The second method, is used to choose the three nodes with best average uploading rates referred to the local node. It's called every `CHOKE_TIME` period and implements a QuickSort algorithm. To compute the average downloading rate from the nodes, the volume of bytes downloaded in the last 20 seconds has to be computed. This is given by the difference between the current amount of data downloaded from the beginning of the protocol (`Element.valueDOWN`) and the data downloaded 20 seconds[4] before (`Element.head20`).

In the case the local node is a seeder and it has to choose the three nodes which upload to, the average uploading rate to them is instead computed. To do this the local nodes computes the difference between `Element.valueUP` and `Element.head20`[5].

4 Using the BitTorrent module

4.1 Setting up the simulation

The BitTorrent network to be simulated is highly customizable through a simple plain-text configuration file. This file (`config-BitTorrent.txt`) provides a way to manually define the parameters of the network and how to use the desired Controls. The BitTorrent module provides to the user two different Controls: an observer (called `BTObserver`) used to evaluate the behavior of the protocol and a control (called `NetworkDynamics`) used to change dynamically the size of the network by adding and removing nodes. The module works by relying over a transport layer, which defines how messages can ben sent among the nodes of the network. The development has been made using the `UniformRandomTransport` layer, which provides a way to reliably deliver messages with a random delay.[6]

The Protocol

The BitTorrent protocol can be fine tuned through the `protocol` parameters.

The allowed parameters are:

file_size Is the size (in Megabytes) of the file shared by the nodes of the BitTorrent network. Since the BitTorrent protocol is widely used to distribute large amounts of data, proper values for this parameter are usually greater than 100. However, every desired value can be specified.

max_swarm_size Is the maximum size of the swarm for each node; this value specifies the dimension of the cache (for each node) in which the neighbor nodes are stored. The default value is 80.

peerset_size Is the number of nodes contained in the PEERSET message; this kind of message is sent from the tracker to the new nodes in the network and specifies a set of nodes to communicate with. The default vale is 50.

duplicated_requests Is the number of duplicated REQUEST messages that can be sent after an UNCHOKE for the same block to different peers; e.g. if the value of `duplicated_requests` is 3, a node requests every block to 3 different neighbors. The default value is 1.

transport Specifies the transport layer used for the simulation. All the tests has been made using the UniformRandomTransport layer (value `urt`).[7]

max_growth This value defines how much the network can grow when the Control NetworkDynamics is used. While the `network.size` parameter defines the number of nodes in the network at the start time of the simulation[8], this value defines the increment factor of the size of the network; e.g. if the size of the network is `30` and the `max_growth` value is set to `20`, the network can increase its dimension up to `50` nodes.

[4] `head20` is set to `valueDOWN` every two `CHOCK_TIME` events.

[5] In this case is updated every 20 s to `valueUP`

[6] The `UniformRandomTransport` layer is provided with the official distribution of Peersim and it's maintained by its authors. Refer to the original documentation for understanding how this layer works.

[7] Have a look at the basic configuration file to understand how to use the `urt` Transport and how to tune its parameter `mindelay` and `maxdelay`.

[8] Remeber that the `network.size` parameter specifies the number of nodes in the network, including the tracker.

Note that the value of the parameter `max_swarm_size` must be greater than the value of the parameter `peerset_size`, since to be sure that the space for the neighbor nodes is enough.

The Initializer

The Initializer provides a way to setup the nodes participating in the network and it is implemented in the two Java classes `NetworkInitializer` and `NodeInitializer`.

The goal of the class `NetworkInitializer` is the initialization of the whole network. It performs the initialization of the tracker (updating its neighbor list) and then of the all nodes participating in the network.

The initialization of a single node (with the exception of the tracker) is performed by the class `NodeInitializer` through the setting of the bandwidth for the peer and choosing *how much* the shared file has been downloaded, according to the configuration parameters.

The initialization procedure ends with the operation of adding to the scheduler the periodic events for each peer.

The configuration parameters of the Initializer are:

newer_distr Specifies the distribution of the shared file for nodes joining the network. This value refers to the percentage of nodes with no yet downloaded pieces. For example, the value 10 means that the 10% of nodes in the network have 0 pieces downloaded (namelly, are *new* nodes).

seeder_distr Specifies the percentage of seeders in the network. For example, the value 10 means that the 10% of the nodes in the network are seeders.

transport Specifies the transport layer used for the simulation. All the tests has been made using the UniformRandomTransport layer (value `urt`).

Note that the distribution of the shared file for the other nodes participating in the network (namelly the nodes that are neither seeder nor new) is picked uniformly at random between 10% and 90%.

The Observer

The Observer is used to evaluate the behavior of the protocol. It periodically keeps track of the status of the shared file (number of pieces downloaded per nodes), the number of pieces currently uploaded/download by each node, the status of the node (leecher or seeder) and some other values like the average number of neighbors per peer.

The only allowed parameter is **step**, which defines the frequency of the execution of the Observer.

The NetworkDynamics

The NetworkDynamics control is used to change dynamically the size of the network by adding and removing nodes. The allowed parameters are:

newer_distr This parameter has the same meaning as explained in the Initializer section.

minsize Specifies the minimum size of the network (in number of nodes) when the NetworkDyanmics Control is used; the network size can decrease until *minsize*.

tracker_can_die Specifies if the tracker can disappear from the network. Note that if the tracker is alive, new nodes can join the network and participate in the file sharing. Otherwise, if the tracker is not alive, only the existing nodes can continue the sharing phase and no new nodes can be added to the network. The default value is 0, meaning that the tracker cannot disappear. The value 1 means that the tracker can die.

step Defines the frequency of the execution of the NetworkDynamics control.

transport Specifies the transport layer used for the simulation. All the tests has been made using the UniformRandomTransport layer (value `urt`).

add Defines the number of new nodes to be added to the network.

remove Defines the number of nodes to be removed from the network.[9]

4.2 Compiling the module

The whole Peersim distribution with the Bitpeer module can be compiled using the provided `Makefile` (located in the main directory bitpeer/ directory of the archive). The original `Makefile` has been replaced with a newer version, where the script `make_jar` is called.

However, a manual compilation of the software can be performed with the following steps:

- execute the command

```
javac -g \
-classpath .:jep-2.3.0.jar:djep-1.0.0.jar \
‘find . -name "*.java"‘
```

- execute `rm -f peersim-1.0.jar` to remove the old jar archive and then run the script `./make_jar` to create a new jar distribution of Peersim.

[9]The network size cannot go out of bounds. The network can increase its dimension up to the *max_growth* value defined in the Protocol section of the configuration file. If the desired number of new nodes cannot be added to the network, the Control tries to add a less number of nodes. The same happens during the removal phase.

4.3 Running the simulation

Once all the parameters are defined through the configuration file and all the classes are compiled, the simulation can be started with the following command

```
java -Xmx384M -cp \
"peersim-1.0.jar:jep-2.3.0.jar:djep-1.0.0.jar" \
peersim.Simulator example/config-BitTorrent.txt
```

Note that the parameter `-Xmx384M` is used to increase the amount of memory used during the simulation (otherwise the default value would be 32 MBytes).

5 Analysis and Results

A simple analysis has been made to understand the behavior of the protocol in the simulated network. Please refer to the Section 6 for some suggestions and ideas on the possible alternative ways of using the BitTorrent module for Peersim.
The Figure 1 shows the behavior of the BitTorrent protocol over time in a network of fixed size (30 nodes) with a single shared torrent (size of 100 Mbytes). Every colored line in the graph represent the number of downloaded pieces for each node in the network. Note that the distribution of the pieces among the nodes in the starting phase of the simulation is randomly chosen: some nodes (few) start with a high number of yet downloaded pieces, others (the majority) have less than 50 completed pieces. When the simulation starts, nodes share the torrent with its neighbors downloading pieces. Note that not all the nodes start sharing when the simulation starts: some nodes are probably choked and are waiting for its selection (this is the case of the pink node which starts with more or less 120 pieces at time 0).

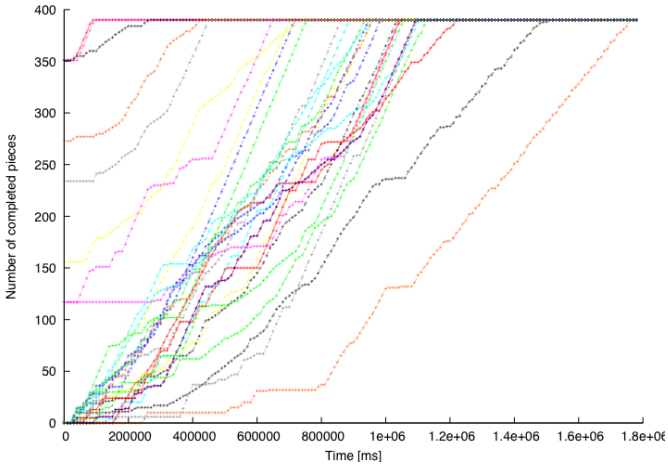

Figure 1: Behavior of the network in a static contest.

6 Future work and Conclusions

This implementation is mainly focused on the core functionalities of the BitTorrent protocol and does not offer advanced features. Further improvement can provide support for the multiple torrent download, the End Game Mode ("get the last few pieces as fast as possible") and Distributed Hash Table (used for allowing peers to download torrents without the use of a standard tracker).[10]

It could be interesting to perform an improved analysis varying the protocol parameters, trying to understand how the behavior of the network change. The default parameter are not always the best choice in particular situations: this module can help to understand how the network performance can be increased.

References

[1] Peersim website, `http://peersim.sourceforge.net/`.

[2] B. Cohen. "Incentives build robustness in Bittorrent". In *Proc. First Workshop on Economics of Peer-to-Peer Systems*, Berkley, USA. June 2003.

[3] A. Legout, G. Urvoy-Keller, P. Minchiardi. "Rarest First and Choke Algorithms Are Enough". In *IMC'06*, Rio de Janeiro, Brazil. October 2006.

[4] G. Wu, T. Chiueh. "How Efficient is BitTorrent?". In *Proc. of SPIE 6071*, January 2006.

[5] D. Erman, D. Ilie, A. Popescu. "BitTorrent Session Characteristics and Models". In *Proc. of HET-NETs*, 2004.

[10] A list of (official and unofficial) protocol extensions, and therefore other new implementation suggestions, is maintained by the theory.org group.