CoopViz: Real Time Visualization of BitTorrent Swarm Dynamics

Stefano Barbieri
CoopViz: Real Time Visualization of BitTorrent
Swarm Dynamics

Master’s Thesis in Computer Engineering

Parallel and Distributed Systems group
Faculty of Electrical Engineering, Mathematics, and Computer Science
Delft University of Technology

Stefano Barbieri

3rd September 2009
Author
Stefano Barbieri
1397052
S.f.barbieri@gmail.com

Title
CoopViz: Real Time Visualization of BitTorrent Swarm Dynamics

MSc presentation
September 09, 2009

Graduation Committee
dr. ir. D.H.J. Epema (chair) Faculty EEMCS, Delft University of Technology
dr. ir. J. Pouwelse (University Supervisor) Faculty EEMCS, Delft University of Technology
dr. ir. G. Gaydadjiev (Committee Member:) Faculty EEMCS, Delft University of Technology
Abstract

Peer to peer networks are only successful when a significant amount of users is willing to share their resources. In order to have highly efficient file sharing systems, each user should upload to the other peers a number of bytes at least equal to the number of bytes he has downloaded from the community. Unfortunately many users show a selfish behavior, refusing to offer their upload bandwidth to other peers, thus limiting the efficiency of the networks. P2P clients implement different mechanisms in order to mitigate this misbehavior and improve cooperation; often, though, these policies are not enough to enforce altruism in peer-to-peer communities. This thesis explores a new mean to stimulate users to give their fair contribution to the P2P community; the final scope is to generate an emergent self-policing system where users take active part in determining the bandwidth allocation policies. This is achieved by means of a graphical interface, CoopViz, that provides users with up-to-date insights into cooperative behavior of peers and allows the latter to vote on each other, showing their appreciation or disapproval. Several simulations are presented to support the development choices and to demonstrate the real time behavior of the visualization.
Muse make the man thy theme, for shrewdness famed
And genius versatile, who far and wide
A Wand’rer, after Ilium overthrown,
Discover’d various cities, and the mind
And manners learn’d of men, in lands remote
- The Odyssey, I.I-IV -
To my parents,
support, guidance and inspiration
throughout all my wanderings.
Preface

This Thesis is part of my Master of Science in Computer Engineering at Delft University of Technology. The research was performed at the Parallel and Distributed Systems Group of the Faculty of Electrical Engineering, Mathematics, and Computer Science of Delft University of Technology.

First and foremost I would like to express my gratitude to my supervisor, Dr. Johan Pouwelse, for his support throughout the project. Thanks to his energy and feedbacks I have always been motivated during the past nine months. I am also greatly thankful to all the Tribler research group for their ideas, contributions and for providing a nice work environment. In particular I would like to thank Dr. Arno Bakker and Tamas Vinko for the code review and thesis feedbacks.

Special thanks go to my parents, who allowed me to embark in this great international experience and to my sisters for their continuous support and precious advices. Last but certainly not least, I would like to thank all my colleagues and friends; in particular Adi, Dhara and Navin for making The Netherlands a home away from home.

Finally I would like to thank Dr. Ir. Epema for chairing the examination committee, and Dr. Ir. Georgi Gaydadjiev for participating in the examination committee.

Stefano Barbieri

Delft, The Netherlands
3rd September 2009
# Contents

Preface ix

1 Introduction 1

1.1 P2P Networks Architecture 2
1.1.1 Centralized P2P Networks 3
1.1.2 Decentralized P2P Networks 4
1.1.3 Hybrid P2P Networks 4
1.2 P2P File Sharing Protocols 6
1.2.1 BitTorrent 7
1.2.2 Tribler 8
1.3 Thesis Outline 9

2 Problem Description 11

2.1 Improving Cooperation 11
2.2 Requirements 13
2.3 Related Work 14

3 Design and Implementation 17

3.1 Graphics Framework Prototyping 18
3.2 Algorithm Prototyping 19
3.3 Peers Selection Prototyping 24
3.4 Implementation with Peers Live Data 25

4 Experiments and Measurements 29

4.1 Prototype Phase Measurements 29
4.2 Implementation in Tribler, Time Behavior 33

5 Conclusions and Future Work 37
List of Figures

1.1 Daily bandwidth Share in Download (a) and Upload (b) [1] 2
1.2 Centralized P2P network 3
1.3 Decentralized P2P network 4
1.4 Flooding algorithm [2] 5
1.5 Hybrid P2P network 5
1.6 Distribution of P2P Protocols [3] 6

2.1 Rank ordering of Peers by Number of Files Shared [4] 12
2.2 Swarm Visualization in Bits On Wheels [5] (a) and Azureus (b) [6] 15

3.1 Continuous Prototyping Methodology 17
3.2 Flow Chart for the Basic Prototype 19
3.3 Screenshot of Basic Prototype, wxPython (a) and PyOpenGL (b) 21
3.4 Flow Chart for the Advanced Prototype 22
3.5 Screenshot of Advanced Prototype, wxPython (a) and PyOpenGL (b) 23
3.6 Flow Chart for the Implementation with Live Data 26
3.7 Screenshot of CoopViz Implementation in Tribler 27
3.8 Pop-Up with User Sharing Statistics (right) and Peers Cooperation Information (left) 28
3.9 Pop-Up with Voting Options 28

4.1 CoopViz, Time Behavior of Basic Prototype wxPython (a) and PyOpenGL (b) 30
4.2 CoopViz, Time Behavior of Advanced Prototype wxPython (a) and PyOpenGL (b) 32
4.3 Response Time of Peers Information Retrieval, wxPython VS PyOpenGL 33
4.4 Time Behavior of CoopViz Implementation in Tribler 34
4.5 Zoom of CoopViz Time Response 34
List of Tables

1.1 Overview of most diffused P2P protocols. . . . . . . . . . . . . . 6
3.1 CoopViz Non Functional Requirements, wxPyhton VS PyOpenGL 19
Chapter 1

Introduction

Before the introduction of P2P (peer-to-peer) technology, internet users mainly relied for files download on FTP (File Transfer Protocol) and HTTP (HyperText Transfer Protocol). These two protocols, while proved to be both robust and reliable, are keen to suffer from congestions when the traffic becomes heavy. Servers that rely on these transfer methodologies provide the possibility of content downloads for an overwhelming number of users, while the upload depends on a single server. It is easy to see how inefficient this architecture, the so-called server-client model, can be; when the traffic to and from the server is not heavy, users experience a steady file transfer, when the traffic becomes severely high the download performance decreases fast and the risk of bandwidth overload arises.

These very drawbacks inspired the architecture of peer-to-peer networks. As suggested by the word peer, defined as one that is equal to another [7], P2P networks are nonhierarchical networks composed of equal peers that behave at the same time as client and server. The unique feature of peer-to-peer networks lies on the usage of all the peers’ bandwidth, so that the total bandwidth available grows with the number of peers connected. Therefore P2P is built on the resources of the users, relying on the correct behavior of the ’men behind the machines’.

It is popular belief that P2P networks belong to a rather new brand of technology architecture; this is in reality far from true. Peer-to-peer networking has been used by universities such as Stanford Research Institute and UC Santa Barbara for more than 30 years [8]. The concept of peer-to-peer networks ’as we know it’, though, has been introduced in mid 1999 by Shawn Fanning, a student at Northeastern University, USA. He is the mind behind Napster, freeware software that enabled users to share MP3 files in a P2P fashion. By the beginning of the 21st century, Napster was a real phenomenon, mostly thanks to the social environment its developer belonged to (College and Universities) that favored the spreading of Napster popularity by means of the most powerful marketing tools, word of mouth advertisement. Another common belief is that P2P networks’ sole purpose is that of file sharing. This is again a wrong conviction, probably due to the popularity of peer-to-peer for file sharing that has shadowed the other purposes of these net-
works. Many services, such as telephony, discussion forums and video streaming are based on this architecture and several companies use it to share knowledge or computing power. To see how powerful are these other applications of P2P, the Intel’s NetBatch can be taken as example; this distributed computing system uses a peer-to-peer architecture to detect and use spare cycles on the company’s servers to solve problems. Intel had noticed that their workstations were idle for 70% of the time; hence in 1990 they started the NetBatch project with a few hundred systems that have grown from 2000 to 10,000 computers. NetBatch utilizes 80 percent of the company’s total computing power, solves 2.7 million problems a month, and in ten years has saved Intel half a billion dollars in equipment costs [9].

To highlight the popularity of P2P networks, Figure 1.1 depicts the daily bandwidth usage in download and upload segmented by the typology of service used.

![Figure 1.1: Daily bandwidth Share in Download (a) and Upload (b) [1].](image)

It can be seen that P2P represents approximately 22% of bandwidth usage in download and an astonishing 61% of the total upload bandwidth consumption. These figures are indeed remarkable, especially considering that P2P networks have been known to the greater public for only six years.

### 1.1 P2P Networks Architecture

A peer-to-peer network is a particular typology of network that allows a group of users to connect with each other and share resources [10]. Examples of resources are bandwidth, information, files and computation power. This thesis focuses on peer-to-peer networks for data sharing, meaning that the resources shared on the network are, essentially, files. P2P networks can be divided in two groups, namely structured and unstructured networks.

In structured networks, every joining peer connects to neighbors chosen according to a predetermined function. Also, the location of every object in the network is established according to this function [11]. The function ensures that the search for information across the network has logarithmic complexity (O(logN), where N is the number of nodes in the network). All files are stored in a specific location, identified and reachable by routing tables [12]; this ensures that any user can successfully run a search for a peer that is sharing the desired file, even if this file is extremely rare. However the routing tables must be updated when peers join
and abandon the network, leading to a non-negligible time overhead (O(logN) for joining peer and O(log^2 N) for leaving peer) [12].

In unstructured networks, a joining peer will connect to other random peers that are already active in the system. The topology of the network changes dynamically when peers join or abandon the network; this kind of network, dynamically built by the peers takes the name of overlay network. This term describe a network that, because of the level of abstraction allowed by TCP/IP, is independent from the physical network [13]; furthermore this kind of networks can be hierarchically organized and routing algorithms can be defined so to implement searching and exchange of control messages. Focus of this thesis are P2P unstructured networks; these will be further explored in the following paragraphs.

1.1.1 Centralized P2P Networks

Centralized networks use a central server to route the traffic between peers, Figure 1.2. The most famous example of client that uses this architecture is Napster.

![Figure 1.2: Centralized P2P network](image)

Upon joining the network, the peer connects to the server providing a list of all the files that it wants to share with the other peers. The server organizes this information by means of a central, dynamic database that contains the relationship between files and IP addresses. Every time a peer adds or removes a file from its sharing list, the server must be notified; this way the indexing and routing are managed by the server alone while the transfers are handled directly by the peers. The advantage of this solution lies in the search function; if a file is being shared on the network it can be surely identified. Nevertheless, this architecture presents several drawbacks. First of all it involves a single point of failure; if the server stops working, the whole P2P system cannot work. Scalability is also a problem; the server must keep track of all the files shared by thousands of peers and, at the
same time, handle the search requests.

1.1.2 Decentralized P2P Networks

Peers belonging to decentralized P2P networks all have same responsibilities and roles, regardless of their geographical position and resources. They work at all times as server and client without any server to coordinate them, Figure 1.3.

![Decentralized P2P network](image)

Figure 1.3: Decentralized P2P network

Peers belonging to a decentralized network do not communicate the files they wish to share; these can only be found through queries. When a peer initiates a query, the request is propagated to the other peers of the overlay network by means of a flooding algorithm as shown in Figure 1.4; the host sends a message to all its known peers that, in turn, forward this message to their known peers until a fixed number of forwards is reached (Time To Live). This strategy has linear complexity in the number of hosts and fails to be efficient in terms of scalability; since the load on each node grows with the number of queries and the system size, peers are keen to get overloaded [14].

This problem has been partially overcome by means of a reduced TTL. This reduced propagation of messages has a positive influence on the traffic but as a drawback, it reduces the efficiency of the queries; the user might not be able to find a file that is in fact present on the network but unreachable because of a low TTL value. Due to the lack of a central server, a peer that wants to join a decentralized P2P network has to connect directly to other peers; it is the peers’ responsibility to keep the overlay network that connects them updated.

1.1.3 Hybrid P2P Networks

In hybrid P2P systems, nodes do not have equal roles; certain peers have more responsibilities because they satisfy certain criteria, such as available bandwidth, time spent connected to the network, computation power, available memory etc.
These nodes take the name of super peers and act as server for their neighbor peers; also they are responsible for establishing a network of super peers by connecting to other super peers, Figure 1.5. This way a two level hierarchy is generated; this provides a better scalability and efficiency if compared to decentralized systems.

A peer, upon connection to the network, will connect to a super peer and will inform the latter about the files it wishes to share. The super peer network handles the search requests; this way, to find a file, every peer only has to maintain a single connection to a super peer. Furthermore the reliability of the systems is guaranteed by the presence of multiple points of failure; if a super peer is unreachable, the network can still work, thanks to the other super peers.
1.2 P2P File Sharing Protocols

Since the release of Napster in 1999, several new protocols have been designed in order to share files and hundreds of clients were released to support these protocols. Figure 1.6 shows the most popular P2P protocols and Table 1.1 gives an overview of their characteristics and correspondent clients.

![Distribution of P2P Protocols](image)

**Table 1.1: Overview of most diffused P2P protocols.**

<table>
<thead>
<tr>
<th>P2P Protocol</th>
<th>Year of release</th>
<th>Description</th>
<th>P2P Clients</th>
</tr>
</thead>
<tbody>
<tr>
<td>Napster</td>
<td>1999</td>
<td>Predecessor of all P2P protocols. Napster is a centralized P2P protocol. Central servers keep track of files shared and connected peers while transactions are handled directly by peers. This protocol and its client, responsible for the booming of P2P networks, were shut down after only 2 years of service. Napster is now a commercial service.</td>
<td>Napster</td>
</tr>
<tr>
<td>eDonkey</td>
<td>2000</td>
<td>The eDonkey protocol relies on servers in order to localize the files on the network. The servers are, hence, subjected to heavy traffic and the whole network cannot work unless at least one server is active.</td>
<td>eDonkey, eMule, Shareazaaz, Lphant, FrostWire</td>
</tr>
<tr>
<td>Gnutella</td>
<td>2000</td>
<td>Gnutella is a pure (completely decentralised) P2P protocol. There are no servers and all peers act at the same time as client and server.</td>
<td>Gnutella, Gnucleus, Shareazaaz, GTKGnutella, Morpheus, FrostWire</td>
</tr>
<tr>
<td>BitTorrent</td>
<td>2001</td>
<td>BitTorrent, nowadays the most popular P2P protocol, enables fast downloads and minimal usage of internet bandwidth. It relies on metadata, called torrents; every torrent corresponds to a file on the network and contains information on the file as well as the address of the servers able to localize the sources of this file.</td>
<td>BitTorrent, uTorrent, BitComet, ABC, BitLord, TurboBT, Azureus</td>
</tr>
<tr>
<td>FastTrack</td>
<td>2003</td>
<td>FastTrack improves the efficiency and scalability of Gnutella by means of hybrid architecture. Certain nodes (super peers) have greater responsibilities and act as server for their neighbor peers.</td>
<td>KaZaA, MLDonkey, Filepipe, giFTFastTrack</td>
</tr>
</tbody>
</table>

The next two paragraphs will give further insights into BitTorrent and Tribler.
that, being at the center of this work, deserve special consideration.

1.2.1 BitTorrent

BitTorrent, accounting for the majority of peer-to-peer traffic on the internet (Figure 1.6), is the most popular peer-to-peer protocol that allows to distribute and share files through the internet. The protocol was designed to reach one main goal: build and provide a system to distribute the same file to the largest number of available users, thus coordinating a large number of computers in order to bring maximum benefit to the community. In order to understand the BitTorrent protocol, it is necessary to introduce the following key terms:

- **Leecher** - Peer that has not yet completed the download of the file.
- **Swarm** - The group of peers collectively connected for a particular file.
- **Seed** - A computer that is sharing a complete copy of a file.
- **Tracker** - A server that assists in the communication between peers. It is also the only major critical point of the BitTorrent architecture, since clients are required to communicate with the tracker to initiate downloads. After the download starts, clients communicate with the tracker periodically to refresh the list of available peers; however, after the initial reception of peer data, peer communication can continue without a tracker.
- **Chunk** - To ease the transmission of data, the original file is split into small parts (chunks), typically of a size that varies between 64 KB and 4 MB, which will be reordered at destination.
- **.torrent file** - A small metadata file containing information about a shared file. The .torrent file acts as an index and contains information on where to find the tracker for the file, the full size and filename of the file. It also contains the number of chunks the original file has been divided in. A checksum is recorded for every chunk using the SHA1 hashing algorithm; this is used by clients to verify the integrity of the data received.

The way BitTorrent works is quite simple. Users browse the web to find a torrent of interest, download it, and open it with a BitTorrent client. At this point the client contacts the tracker indicated in the .torrent file and, if the tracker server is found and seeds are available, it launches a download process of the file. Once the user has downloaded part of the file, it starts uploading its chunks to other peers in the swarms, relieving this way the bandwidth consumption of the seeders.

Fairness in the BitTorrent system is implemented by enforcing tit-for-tat exchange of content between peers. A peer maintains two states for each peer relationship, interested and choked. The peer wishing to download a file notifies the peers that have chunks of needed content by mean of an interested message. At
this point it is on choked status and will not receive any data from the peers until unchoking occurs. Each BitTorrent peer always unchokes (uploads data to) a fixed number of other peers simultaneously; this number usually varies between 4 to 6. BitTorrent’s bandwidth allocation policy foresee that leechers must assign their slots to the most generous peers; thus those peers that currently provide the highest upload rate in return. Seeders, on the other hand, must assign their upload slots to those peers that have the fastest download rate, so to rapidly spread content and increase the number of seeders. An extra slot is reserved for optimistic unchoking; this is assigned randomly via a 30 seconds round-robin shift over all the interested peers regardless of their upload rate. This mechanism allows peers that have no content to join the swarm and, after they have received at least one chunk, start sharing.

1.2.2 Tribler

Tribler is an open source BitTorrent client developed by the Parallel and Distributed System group of the Faculty of Electrical Engineering, Mathematics and Computer Science at the Technische Universiteit Delft. The name Tribler originates from the word Tribe, referring to the fact that the software implements a sense of community and allows to mark specific users as online friends. These can be used, upon their permission, to increase the download speed of files by using their upload capacity [15]. This is only one of the features that make this client unique. Tribler also use a gossip protocol to add keyword search ability to the BitTorrent file download protocol, removing the need for central elements in order to find content. Furthermore it includes the ability to recommend content to users; after a dozen downloads the Tribler software can roughly estimate the download taste of the user and make recommendation.

To prevent free riding, the client deploys a distributed mechanism called Bartercast [16] that allows to allocate bandwidth according to the long term behavior of peers. This is a step forward from Tit-for-Tat, limited by the fact that bandwidth allocation policies are based exclusively on the short term behavior of peers. Tribler, through the Bartercast message exchange protocol, introduces a long-term based reputation metric that allows a distinction between sharers, freeriders, and neutral peers. The reputation value, scaled between -1 and 1, permits to measure the altruism levels of peers, defined as the amount of uploading versus downloading of a user. New peers entering the network and peers that upload as much as they download get a neutral reputation value equal to 0. The reputation based mechanism foresees that peers only assign their upload slots to peers that have a reputation which is above a certain threshold, which has a fixed value below zero. When a peer shows a free-riding behavior, downloading more than it uploads, its reputation will fall below the threshold value. From then on, downloading files will take the user more and more time. Therefore, this mechanism encourages peers to seed each downloaded file until they have uploaded at least the size of the file to others, obtaining this way a neutral reputation. The Bartercast messages are saved on a
local database so that the reputation is computed on a long term basis rather than on a one session basis.

1.3 Thesis Outline

The thesis is structured as follows. Chapter 2 deals with the problem of stimulating cooperation in peer-to-peer networks; furthermore an overview of the work related to CoopViz is presented. Chapter 3 introduces the design and implementation of CoopViz, graphical interface aimed at highlighting health and contribution of peers. This chapter offers a detailed description of the design decisions taken during the nine month thesis project. In Chapter 4 the results of the experiments performed are presented; these show the time behaviour of CoopViz at prototype stage and when implemented in Tribler. Conclusions, ideas for future research as well as possible further investigation into peers behavior and cooperation are presented in Chapter 5.
Chapter 2

Problem Description

In Section 2.1 the problem of stimulating cooperation in peer-to-peer networks will be described. Particular emphasis will be laid on explaining the importance of cooperation, describing the current level of cooperation in P2P networks and how CoopViz can help improve this situation. In Section 2.2 the non functional requirements set for the development of CoopViz will be discussed. In Section 2.3 the related work will be presented; the swarms visualization currently implemented in two BitTorrent clients will be analysed and compared to CoopViz.

2.1 Improving Cooperation

File sharing systems are only successful when a significant amount of users is willing to share resources. Unfortunately, while some users share their bandwidth and files altruistically, others only share when they are encouraged to do so and might freeride otherwise [16]. The phenomenon of free riding is one of the main issues of P2P networks, originated from those users that refuse to share their files or limit their upload bandwidth and yet download content from other peers taking advantage of other users’ resources. This kind of actions go against the very foundation of P2P systems, communities where every peer must cooperate with each other in order to keep resources available. In order to have highly efficient peer-to-peer files sharing networks, each and every user should upload to the other peers a number of bytes at least equal to the number of bytes it has downloaded from the community. It is, therefore, imperative to stimulate the cooperation between peers and reverse their independent and selfish behavior before incurring in what is known in the literature as 'The Tragedy of The Commons' [17]. This is the name of an article written by Garrett Hardin in 1968 that describes a community where each individual tries to exploit as much as possible of a common resource, managing to destroy it even if it is in the collective best interest to keep it intact. Bandwidth allocations policy have been implemented in P2P protocols to improve the level of intra-peers cooperation; eMule, for example, uses a point/credit systems so that the users who share their resources the most are given download precedence. Unfortu-
nately, surveys show that no policy implemented so far has succeeded in reducing this trend. Users still do not share their resources and sometimes even create or obtain cheating clients to prevent having to do so. To support these facts, in [4] is shown that in a popular P2P client, Gnutella, 70% of peers does not share any file (Figure 2.1) and a mere 1% provide the community with 47% of the total service.

![Figure 2.1: Rank ordering of Peers by Number of Files Shared [4]](image)

If these figures still represent the P2P reality after 8 years from their initial boom, how is it possible to enforce policies to encourage people to disengage from their selfish behavior? If client dictated policies have not accomplished this goal so far, is it possible that more flexible rules, based on users inputs, could improve the current free riding situation? This thesis explores a new mean to stimulate users to give their fair contribution to the P2P community; the final scope is to generate an emergent self-policing system where users take active part in determining the bandwidth allocation policies. The main idea behind the work is to stimulate cooperation between peers by inculcating in the users a sense of belonging to a community. Therefore, a user interface has been designed and implemented in Tribler to show the bandwidth contributions of peers. In this way, peers can see that the bandwidth they are sharing is donated to other users, real people of whom they can retrieve information, such as status, name and sharing behavior. The act of sharing bandwidth loses its abstraction and becomes tangible; users can, at any moment, visualize the peers they are connected with, have an overview of their sharing behavior and reputation. In [18] it is shown that in private BitTorrent communities the download performance is higher than in public communities; a larger number of peers share their resources because of a centralized sharing-ratio enforcement. This mechanism foresee, in example, the banning of peers that upload too little compared to what they download [16]. Therefore, it appears that stricter policies are effective in stimulating users to obey to the rules. However the implementation of a central technology to monitor and sanction the peers would
require the trust of users in an unknown authority. This thesis evaluates the possibility to allow users to enforce their own policies; in this way they do not have to trust or depend from a central authority. After visualizing health and contribution of the peers they are connected to, users can implement themselves stricter policies. These could involve the banning of free riding peers and the promotion of the generous ones. In this case, though, there is the risk of users ‘playing’ with the interface and randomly blocking peers or blocking peers to avoid sharing bandwidth. Hence, on a less drastic note, it has been decided to allow peers to vote on other users, attributing ‘thumbs up’ or ‘thumbs down’ to manifest their appreciation or disapproval. This voting system can then have weighted influence on bandwidth allocation policies.

The Visualization can also be used for research purposes; it allows to have a graphical representation of the evolution of the swarms as time passes and capture at every instant all the properties of the peers. This topic is nowadays at the center of numerous research works; scientists are trying to gain more insights on what exactly happens inside a swarm during its life, what is the behavior of the peers, how much bandwidth the users share, how long they seed their files and which kind of social behavior they exhibit. An example of this kind of research can be found in [19, 20, 21].

2.2 Requirements

The following fundamental non-functional requirements have been set at the beginning of the project.

- Portability. The Swarm Dynamics Visualization is designed to be integrated with the Tribler software. Therefore it must be projected in order to work with the different platforms supported by the client, namely Windows, Linux and Mac OS.

- Efficiency. The interface must be easy to use; the actions that can be performed by the user must be accomplishable in an intuitive way.

- Learnability. The average user must be able to understand the purpose of the interface, even at the first run. The Visualization must represent the dynamics of the swarms in such a way that users can capture at a glance the contribution of peers. Also, the average user wants to collect practical information about the peers he is connected to; in the Tribler implementation, only basic and intuitive properties of peers must be shown.

- Performance. The Visualization must have short response time. The peers population inside the swarms varies continuously; these dynamics must be observable without jitters. Because of rapid changes, the retrieval and display of peers’ information must occur in real time; a long response time
would imply not up-to-date information on peers or even information about peers that do not belong to the swarm anymore. The Swarm Dynamics visualization can also be used as a tool of research to investigate the peers’ behavior and the evolutions and changes that take place inside the swarms; in this case a short response time in visualizing the peers and gathering their information is even more significant. Since users do not have a PC dedicated to file sharing clients and run the software simultaneously with other applications, a reasonable utilization of computing resources must also be achieved.

- Satisfaction. The Swarm Visualization interface is not only a way to show the current bandwidth exchanges on the P2P network and stimulate cooperation. It is also a screen at which users can look while exchanging their files, and it should therefore be aesthetically pleasing.

### 2.3 Related Work

The problem of cooperation in P2P networks is a very active research topic. Several studies investigate robust incentive mechanisms to achieve more efficient P2P systems and devise fairer bandwidth allocation policies [16, 22, 23, 24].

Another kind of related work is represented by the swarms visualizations recently implemented in two different BitTorrent clients. Azureus (recently renamed to Vuze) [6] and Bit On Wheels [5], currently provide a swarm visualization interface; the features of this related work will be here discussed with particular focus on how it differs from Triber’s Swarm Visualization.

Both clients offer the users the possibility to browse between active swarms and retrieve a visual representation of them. The swarm is represented in 2D as multiple circles (the peers) rotating around a fixed circle (the user). Both clients also offer a 3D representation of the swarms, realised in OpenGL, which can be manually set, allowing the user to change the level of zoom and the perspective of the representation.

Figure 2.2(a) is a screenshot of the three dimensional swarm visualization implemented in Bit On Wheels, while Figure 2.2(b) shows Azureus’ 2D implementation.

The main differences between the work here presented and the visualizations offered by these two clients reside in the scope of the visualization and in what can be inferred from it. The information displayed in Azureus and Bit On Wheels depict seeds and leechers, which percentage of the file the leechers have already transferred, and which peers are currently sending and receiving data to/from the user. The visualization proposed in this work focus, instead, on representing health and contribution of peers so to highlight their sharing behavior. Bit On Wheels provide the visualization for entertainment purposes only. Azureus inserts the visualization in the context of many detailed information collected from the swarms. Several insights are offered, such as connected peers data including IP addresses,
the speed at which the user is downloading and uploading to/from them, the BitTorrent client they are using, the total average swarm speed and information over the file chunks, their distribution and which of them are currently being transferred. These insights though, are not reachable or summarized in the visualization itself, but are available in text format in separate tabs. Also there is no way in which the user can influence the sharing-ratio enforcement mechanism on the basis of these information.

The Swarm Visualization in Tribler is, then, somehow unique. It gathers the properties of the peers connected to the swarms, summarizes the most important of these in the visualization itself and allows the user to retrieve the others with a simple click on a peer. Moreover it has the scope of highlighting the peers’ behavior and giving users the possibility to influence the bandwidth allocation mechanisms.

From a performance point of view, both Bit on Wheels and Azureus visualizations run smoothly in 2D but present a slow refresh rate in 3D; in this work, instead, particular attention has been paid to the refresh rate and response time of the visualizations, so that users do not experience jitters and can interact with the interface in real time.

From a learnability perspective, the visualization implemented in Azureus represents the swarm status in a very comprehensible way, while Bit On Wheels lacks in clarity; especially while using the 3D visualization it is very difficult to interpret the information visualized on the screen. This is very different from the two and three dimensional visualizations proposed in this work, where the ease in interpreting the representation has been put amongst the primary objectives.
Chapter 3

Design and Implementation

The Swarm Dynamics Visualization has been developed using the incremental, iterative approach of continuous prototyping. The requirements and final goal of the project have been devised at the beginning while design and implementation were quite flexible. Hence the prototyping methodology allowed starting with the development of the visualization without precluding the possibility for modifications or new ideas. In Figure 3.1 is represented the approach followed.

Several concepts have been developed, which lead to two major prototypes. This chapter presents the evolution of CoopViz, from concept to implementation. Section 3.1, 3.2 and 3.3 present the development choices at prototype stage, what has been learned from them and which of these have been adopted for the final version of CoopViz. Section 3.4 deals with the implementation of CoopViz in the Tribler peer-to-peer client, thus obtaining a coupling with real live peers data.
3.1 Graphics Framework Prototyping

Deciding which programming language was best for the Visualization was straightforward, given that Tribler is fully coded in Python. Since drawing functions were needed to develop the Swarms visualization, the hardest development choice was to decide which graphics libraries were better suited for the purpose of the project. After a careful analysis of the different possibilities, two versions of each prototype have been developed; one using WxPython and one using PyOpenGL.

WxPython is a GUI toolkit for the Python programming language that allows to create Python programs with a robust, highly functional graphical user interface. WxPython is implemented as a Python extension module that wraps the wxWidgets cross platform GUI library, which is written in C++ [25]. WxPython is Open Source and runs on multiple platforms, such as Microsoft Windows, most Unix or unix-like systems, and Macintosh OS X.

PyOpenGL is the cross platform Python binding to OpenGL and related APIs provided under Open Source license. OpenGL (Open Graphics Library) is a very popular and flexible software interface to graphics hardware. The interface offers several function calls which can be used to draw complex two and three-dimensional scenes from simple geometric primitives. The library also includes routines for rendering the scenes with control over lighting, object surface properties, transparency, anti-aliasing and texture mapping. OpenGL is a hardware-independent interface to be implemented on many different graphics hardware platforms [26]. No commands for performing windowing tasks or obtaining user input are included in OpenGL; these will be, in this work, implemented in Python.

Both libraries were suitable for developing the Swarms Visualization; they both run on multiple platforms, provide functions to draw and can be integrated in Python code. WxPython supports GUI related widgets such as boxes, pop-ups and menus but does not allow to create fancy graphics. PyOpenGL, as previously mentioned, does not include any command for windowing tasks or user input but permits the creation of graphically impressive 3D scenarios. Therefore it has been decided to develop two versions of each of the two prototypes, so to evaluate both libraries and then use the best and more conform to the requirements for the implementation in Tribler.

Taking into account all requirements set in Section 2.2, WxPython proved to be the best implementation. The graphics allowed by its drawing libraries is not as impressive as that allowed by OpenGL, but still appreciable. What makes WxPython the best solution, though, are the learnability and performance factors (Table 3.1). From a learnability point of view, the 3D implementation fails to emphasize and represent clearly the properties of peers. The third dimension renders the distance in between the spheres less accentuated; this way the perception of the different peers speeds is reduced, especially when multiple peers have similar speed. From a performance perspective, PyOpenGL proves to be drastically slower than WxPython. The response time of the different implementations will be discussed in further details in Chapter 4.
### Table 3.1: CoopViz Non Functional Requirements, wxPyhton VS PyOpenGL

<table>
<thead>
<tr>
<th>Requirements</th>
<th>wxPython</th>
<th>OpenGL</th>
</tr>
</thead>
<tbody>
<tr>
<td>Portability</td>
<td>V</td>
<td>V</td>
</tr>
<tr>
<td>Efficiency</td>
<td>V</td>
<td>V</td>
</tr>
<tr>
<td>Performance</td>
<td>V</td>
<td>X</td>
</tr>
<tr>
<td>Learnability</td>
<td>V</td>
<td>X</td>
</tr>
<tr>
<td>Satisfaction</td>
<td>V</td>
<td>V</td>
</tr>
</tbody>
</table>

#### 3.2 Algorithm Prototyping

After several trial concepts, two final prototypes have been developed; these will be referred to as basic and advanced prototype. The basic prototype was developed taking simplicity as fundamental requirement, so to have an easy to read, easy to use graphical interface showing the fundamental traits of peers. The advanced prototype, instead, was designed to represent all properties of peers and, at the same time, allow the user to focus on a particular property through interaction with the interface.

![Flow Chart for the Basic Prototype](image)

The prototypes had to be representative of the final implementation in Tribler, so the first step in the algorithm is the generation of random peers. To handle the peer attributes, the Peer Class has been developed, so that all properties of peers can be set and retrieved. Once the peers information have been set, the peers are grouped...
in a swarm. For this purpose the Swarm Class has been designed; via the functions of this class it is possible to access all the peers in the swarm and modify and read their attributes.

Figure 3.2 shows the algorithm for the basic prototype. The peers are visualized on screen as circles, or spheres in 3D, disposed around a central peer that represents the current users. After carefully analysing the possible alternatives, it has been decided that the most relevant property of peers the users want to have an insight on is the upload speed, that denote how generous a peer is. Hence the randomly generated peers are sorted according to their current bandwidth in upload and the distance of the peers from the center describes this speed; the more the peers are near the center, the greater is their shared bandwidth in upload. The upload speed is scaled, so that all the peers have coordinates that fall inside the range of the screen resolution.

\[
SUpBandwidth_i = \text{MaxD} - \frac{\log(UpB_i) \times \text{MaxD} - \text{MinD}}{\log(\text{MaxUp})} \tag{3.1}
\]

where \(\text{MaxD}\) and \(\text{MinD}\) are respectively the maximum and minimum distance from the center of the screen allowed by the monitor resolution, \(UpB_i\) is the bandwidth in upload of the i-th peer and \(\text{MaxUp}\) is the greatest upload speed between all active peers.

The position \((x_i, y_i)\) of the center of the circles that represent each peer is assigned according to the following formulas.

\[
x_i = x_s + SUpBandwidth_i \times \cos(t) \tag{3.2}
\]
\[
y_i = y_s + SUpBandwidth_i \times \sin(t) \tag{3.3}
\]

where \((x_s, y_s)\) are the coordinates of the center of the screen, \(SUpBandwidth_i\) is the scaled bandwidth of the i-th peer and \(t\) is a variable \(0 <= t <= 360\) used to dispose the peers in a circle.

The above formulas are only valid for the wxPython code; in OpenGL there is no exact way to specific the coordinates of the spheres; though their distance from the center still represents their bandwidth in upload, since they are translated in space proportionally to their upload speed.

The size of the peers represents instead the reputation; this is, as described in 1.2.2, a unique property of Tribler peers that highlights their past and present sharing behavior. The reputation of the peers is used in the visualization to establish the ray of the circles, or spheres in three dimensions. After coordinates and rays have been assigned, the most generous 50 peers are visualized on screen, Figure 3.3.

The algorithm for the advanced prototype include multiple visualizations and the possibility for the user to customize the interface, Figure 3.4. Three different visualization modes have been implemented.
Figure 3.3: Screenshot of Basic Prototype, wxPython (a) and PyOpenGL (b)

- Latency VS Upload - Peers are scattered across the screen according to their latency; their size is computed considering their current bandwidth in upload
- Latency VS Reputation - Peers are scattered across the screen according to their latency; their reputation is represented by the size of the circle
- Upload VS Reputation - Peers are scattered across the screen according to their current bandwidth in upload; their size is computed according to their reputation

The visualization screen is divided into quarters; every quarter focuses on a particular property of peers. This way it is possible to discern which peers in the swarm have the highest bandwidth in upload, which the best latency and which have been the most generous and uploaded the most. In the last quarter are visualized, instead, random peers. The user can choose the visualization mode, the number of
peers to visualize on screen and which of the above properties have to be considered. This way it is possible to have an exhaustive overview of all the best and more cooperative peers in the swarm or obtain a detailed view by focusing on a particular property. The visualization achieved is depicted in Figure 3.5.

As stated before, the prototypes had to be representative of the final implementation in Tribler; when downloading and uploading files using the client, the peers
a user is connected to are not always the same, but vary with the time. Some peers leave the swarm and others join; to reflect this situation the prototypes are continuously refreshed using an idle function. New random peers are generated and the algorithm is followed again in a loop.

For the implementation in Tribler, the final Coopviz takes its simplicity from the basic prototype; the different options available in the advanced prototype would have resulted too insightful for the average user. On the other hand, it has been decided to maintain, as in the advanced prototype, the division between BitTorrent peers and Tribler peers, fundamental to show users which peer belong to the same community and build a sense of belonging. Also, from the prototypes it has been learned that the peers’ reputation does not represent intuitively the degree of peers cooperation; hence, in the implementation in Tribler, the size of the circles that represent the peers will describe the MegaBytes uploaded rather than the reputation.
3.3 Peers Selection Prototyping

The algorithms and classes previously described have been implemented in both WxPython and PyOpenGL prototypes. The major differences between these two versions lie in the visualization function and the mechanism to implement the selection of peers on the screen. This procedure, called ‘object picking’, had to be implemented to give the users the possibility to visualize a pop-up window with the information on the selected peer when clicking on it.

With WxPython two different ways of implementing the object picking procedure have been explored.

The first approach is the ‘by-the-book’ WxPython procedure of drawing the objects on the screen and at the same time record the drawings in an operations list. The operations are, then, tagged with an id in order to associate them with the specific object; in this way when the user selects a peer, the unique id of the selected object is used to identify, through a for loop, the corresponding peer and retrieve and print its information on screen. This procedure is effective but very code intensive; it also lacks precision since the shape of the object in the operation list is approximated with a rectangular area. Hence the effectiveness of this methodology varies according to the shapes drawn; in the Swarm Visualization case, where the objects are circles and can be very near to each other (i.e. if they all have a high bandwidth in upload), this procedure proved not to be 100% adequate. In case two peers were near, a click in the proximities of one of the two peers often resulted in the visualization of the other peer’s information.

Because of this, it has been decided to devise an original way to map the peers on screen. Each peer is assigned two coordinates \((x_i, y_i)\) and a diameter \((\text{Reputation})\). Then, through a for loop, it is possible to establish which peer the user has clicked on; this will be the only one whose coordinates satisfy the following inequality:

\[
(x + x_i)^2 + (y + y_i)^2 < (\text{Reputation}_i)^2
\]  

(3.4)

where \((x, y)\) are the coordinates of the point on the screen clicked by the user, \((x_i, y_i)\) and \(\text{Reputation}_i\) are respectively coordinates and ray of the \(i\)-th peer.

This methodology proves to be much more precise, less code intensive (only one line to attribute the coordinates to the peers) and still shows a very good time performance. Hence it has been chosen as the object picking technique for the final CoopViz implemented in Tribler.

In OpenGL, since objects drawn on the screen undergo multiple rotations, translations and perspective transformations, the use of coordinates to determine which object a user is selecting in a three-dimensional scene is not a viable option. OpenGL, though, provides a selection mechanism not very different from the one offered by WxPython able to detect which objects are drawn inside a specified region of the window. This mechanism can be used together with a special utility routine to determine which object the user is picking with the cursor. In order to accomplish this, first the scene must be drawn on the screen; then, the scene must be redrawn
in selection mode. During this procedure, a name stack is constructed by loading a name whenever an object is redrawn. Once redrawing is complete, for every object selected by the user with a mouse click, OpenGL returns a selection hit. Since every selection hit reports a name from the name stack and every name in the name stack has a unique correspondence to a drawn peer, it is possible to recognize which peer has been clicked on by the user and show its properties. This object picking mechanism works very well; it is very precise and quite flexible; for example, a special utility is available to twick the ’picking precision’, allowing to restrict or enlarge the area picked by the mouse or discern between multiple objects drawn in the same area by specifying their color or shape. The main drawback is the time efficiency of this procedure; since the prototype is continuously refreshed and the peers on screen change, in practice it is necessary to redraw the scenery every time a user click on the screen. This, as it will be discussed in the following chapter, is a very time expensive procedure.

3.4 Implementation with Peers Live Data

This section describes the implementation of CoopViz with the Tribler client. CoopViz, being a graphical interface, has been integrated with the vwxGUI code of Tribler. A new menu has been created in the client’s main GUI to allow the launch of the Visualization from any page of the Tribler interface. Figure 3.6 illustrates the final algorithm. When CoopViz is launched, a preliminary check on the status of the peer is performed; if the user is active in at least a swarm, he gets the ’active’ status otherwise he is assigned the ’inactive’ status.

In case the peer is inactive, the user is proposed with an overview of the network according to its previous sessions. CoopViz will retrieve all peers in the Bartercast database, will sort them and assign them a size according to the amount of MegaBytes they have transferred to the given peer; then it will visualize them on screen. Only the most 50 generous peers are graphed. The user can click on the circles that represent the peers and retrieve their properties; these include MegaBytes uploaded, MegaBytes downloaded, Tribler name, how many times a connection has been established with that user and if the latter is a Tribler friend or not. The bartercast database does not contain all these crucial information; hence more properties about each peer are fetched by cross-referencing the peerId property with the Tribler’s Peer database.

The active/inactive check is performed every 150 msec. Therefore, as soon as the user starts downloading or seeding a file, CoopViz will stop visualizing locally stored data and will switch to live data. This is sent directly from Tribler’s main module to CoopViz through a shared variable containing a list of all user’s active swarms. Using the getpeerlist function available in the DownloadState module of the Tribler Core code, it is possible to retrieve from this variable, for every swarm a list of dictionaries, one for each connected peer, containing all the properties for that peer. Peers’ properties are handled, as in the prototypes, through the Peer and
Examining the role (leecher or seeder) of the user in the active swarms, the user is assigned another status, 'downloading' or 'seeding'. This status is necessary to adjust the visualization parameters. If the user is downloading, all active peers in the swarms are sorted and scattered on the screen according to their current bandwidth in upload; also, the size of the circles that represent them is computed according to the MegaBytes the peers have already uploaded to the user. If the user is seeding, the current bandwidth in download and the downloaded MegaBytes are
considered, instead.

BitTorrent and Tribler Peers are separated one from each other and represented by different colors; yellow for Tribler and purple for BitTorrent. This division is important not only for statistics purpose, but also because Tribler members can have interactions with each other (add as friend, boost bandwidth) that are not possible with peers running other BitTorrent clients.

A representation of the peers encountered during the past sessions is given also in the active mode; these peers are retrieved as described for the inactive status. The on-screen visualization for the active mode represents, then, the best 25 Bartercast peers and the best 25 live peers (Figure 3.7).

At this point the active/inactive check is performed again. If the user is inactive, only the Bartercast peers will be displayed on screen as described above. If the user is still active, CoopViz will check if new data has been made available by the Tribler Main module. If so, the algorithm will go through the whole process; on the other case, the same 50 peers as before will be displayed on screen. This operation has been devised to avoid data starvation; in case the Main module does not have new peer data to send, the CoopViz screen still displays information.

Thanks to the continuous data refreshes, the user has at all times an up-to-date representation of the network. He can infer at-a-glance how ‘healthy’ (bandwidth speed) are the peers he is connected to by looking at how near to the center these are represented. At the same time the user can see how ‘generous’ (size of the circles when downloading) or ‘greedy’ (size of the circles when uploading) the other peers are. By clicking on the peers on screen, further insights into the contribution of peers are available, Figure 3.8. CoopViz also allow the user to retrieve his
personal statistics. By right click of the mouse on the red circle at the center of
the screen, that represents himself, a pop up appears on screen with all the current
user information (Figure 3.8). On the basis of peers’ behavior, as inferred from the
on screen visualization and from the pop-up information, the user can establish
whether the selected peer is a contributor, a neutral user or a free rider. At this
point, with a right click of the mouse on the peer, users can vote on the peer with a
‘thumbs up’ or ‘thumbs down’ Figure 3.9.

Figure 3.8: Pop-Up with User Sharing Statistics (right) and Peers Cooperation
Information (left)

Figure 3.9: Pop-Up with Voting Options

The final version of CoopViz implemented in Tribler is fully coded in Python and
wxPython; the modularity of the client’s code has allowed a seamless integration of
the Visualization. In order to maintain consistency with the rest of the Tribler code,
CoopViz has been integrated as the rest of the client’s GUI. Hence the majority of
modifications were made to the wxGUI Tribler code.

When clicking the CoopViz button on the Tribler’s Top Bar, the GUIUtility class
recognizes which button has been selected and passes the information to the StandardOverview class. Here the CoopViz Overview is loaded by mean of a xrc file
that specifies the position of the Visualization panel inside the Tribler frame, its
size, background color and the function to call to start CoopViz. The Visualization
runs on a dedicated thread. In order to catch the continuous variations in
the swarms, the interface is refreshed by means of an idle function that calls the
CoopViz main function every 150 milliseconds.
Chapter 4

Experiments and Measurements

The following Sections propose an overview of the experiments performed on CoopViz at prototype and implementation stage. The measurements on the prototypes have been taken with the scope of evaluating which of the development choices was best to be adopted in the implementation with peers live data. The experiments performed on CoopViz implementation have the dual scope of highlighting the quality of the implementation and verify if the Visualization is compliant with the real time constraints established at the beginning of the project.

All measurements have been taken under Ubuntu 9.04. The average P2P user does not own a high performance computer; hence for all measurements a 2.0 GHz Intel Centrino laptop with 2 Gbytes of RAM has been used.

4.1 Prototype Phase Measurements

The time behavior of the prototypes has been measured by varying the number of peers visualized on screen. Since part of the requirements was to achieve a Visualization with reasonable consumption of resources, the CPU usage has been graphed along with the response time. In order to better explain the time results, three times have been measured and graphed:

- Total Response Time - Time of all the algorithms’ phases, from peers generation to on-screen visualization.
- Operations Time - Time spent on operations on peers such as generation, properties retrieval and sorting.
- Visualization Time - Time necessary to scatter the peers on screen, attribute the color, shape and draw them.

For every prototype, the wxPython implementation was compared with that coded in PyOpenGL, so to establish which of the two libraries was more compliant to the real time requirements of CoopViz final implementation. Finally, the object picking procedure, described in Section 3.3, has been timed; from this procedure
depends the time the user has to wait before visualizing the statistics of other peers and a short response time is, therefore, crucial.

Figure shows the time behavior of wxPython and PyOpenGL implementations of the basic prototype. The difference in total response time between the two im-

![Figure 4.1: CoopViz, Time Behavior of Basic Prototype wxPython (a) and PyOpenGL (b) implementation is striking; the GL basic prototype is averagely 10 times slower than the wx one. The algorithms are identical; the library used for the visualization is the only variable. This explains why the Operations Times are very similar, while the
Visualization Times are sensibly different. The visualization process in PyOpenGL is a very heavy operation, observation supported by the CPU usage reported in the above graph.

These time results are one of the main reason why CoopViz implementation has been coded in wxPython; the time measurements of the corresponding prototype show a real time behavior regardless of the number of peers visualized on screen; the CPU consumption, with peaks of maximum 40% utilization, is also very reasonable. The OpenGL prototype lacks in flexibility; it shows a real time behavior for a visualization of less than 100 peers, but shows an unacceptable response time otherwise. The resulting CPU utilization is also higher; this is due to the greater complexity of the 3D OpenGL graphic.

The same observations are valid for the Advanced Prototype, Figure 4.2. In these graphs is also highlighted how the increased complexity of the Advanced Algorithm influences the response time. The Operations Times become longer than in the Basic Prototype because of the multiple sorting operations and the augmented peers’ properties to retrieve and set. It can also be seen that the Visualization Times are higher than those shown in Figure 4.1, while the number of peers visualized is the same. This is due to the the use of many if statements in the code, in order to take into account the users’ input; these conditional statements induce extra jumps and branches in the low level code which take extra time to compute.

The last experiment on the prototypes concerns the response time of peers information retrieval, Figure 4.3. As explained in Section 3.3 the process of selecting an object on screen and retrieving its properties depends from the drawing library used. In wxPython it is sufficient to retrieve the coordinates of the point clicked by the mouse and verify if these fall into the area of any object on screen. In PyOpenGL it is necessary to redraw the entire scene and objects in the selection buffer and verify if the click has generated a selection hit. The graph in Figure 4.3\(^1\) shows the different time behavior of these two implementations when increasing the number of objects that can be selected on screen. The Visualization was run forty times, increasing at every iteration the number of peers by fifty. The response time describes the time employed to discern which peer has been selected, retrieve its properties and visualize them on screen via a pop-up. wxPython presents at all times a real time peers information retrieval; PyOpenGL presents, again, a real time behavior only when the objects to handle are less than 100. The response time of the object picking procedure varies in both implementations with the number of objects on screen. With wxPython these variations are minimal, due only to the increased number of iterations in the for loop that effectuates the coordinates matching. In PyOpenGL, instead, the response time grows almost linearly with the number of peers; not only the for loop that takes care of the hit-to-object matching increases in size, the number of peers that have to be redrawn in selection mode grows too.

---

\(^1\)Note that the y axis of the wxPython timings shows values multiplied by $10^{-3}$.
These experiments demonstrate the clear superiority of wxPython implementation over PyOpenGL’s; for a large number of on-screen peers, the wx object picking procedure proves to be more than 1000 times faster. The algorithm for the implementation of CoopViz with real live data foresees the visualization on screen of only 50 peers, for which both implementations show a real time behavior. This number of visualized peers, though, is very easy to modify via the CoopViz code and is likely to be changed in the future. It was hence necessary to adopt the library who not only proved to be efficient but also flexible to these variations.

Figure 4.2: CoopViz, Time Behavior of Advanced Prototype wxPython (a) and PyOpenGL (b)
CoopViz has been implemented on the pre-release 5.0 version of the Tribler client. The experiments have been performed in a live environment; the variation in CPU usage of the machine and the response time of the Visualization have been registered while increasing the number of peers handled by CoopViz. To achieve this, gradually, more and more files have been downloaded thus increasing the number of connections with peers established by the client.

The response times registered describe the time to perform the whole CoopViz algorithm, from data retrieval to on screen visualization. In Figure 4.4 the results obtained are graphed.

The first thing that has to be highlighted is that the real time behavior of CoopViz has been preserved also in the implementation with live data. The response times fluctuates between 11 and 160 milliseconds with an average of 50 msec. Also, by looking at the response time graph, it is clearly visible that the time behavior is highly non linear. The inspection of the CPU variations alone does not explain these peaks and lows; to gain a better understanding, the timings have been taken again, this time marking at every iteration if new data had been made available by the Tribler Main module or not. Figure 4.5 presents a 20 seconds snapshot of this experiment.

The peaks in the graph correspond to a situation when new live data is accessible, while the lows correspond to no new data available. As explained in Section 3.4, the Visualization has been coded so to avoid data starvation and consequent jitters; hence if CoopViz refreshes before Tribler has had the time to provide new live data, the Visualization draws on screen the same peers it had displayed the instant before. In this case CoopViz does not have to perform any pre-visualization processing, such as retrieve the new data, sort the peers and set their properties. The
resulting response time is, hence, shorter. There are several reasons why Tribler
could not have any data ready to send. As stated before, the number of peers has
been increased by adding more and more files to the download list; hence, during
the 20 minutes experiment, some files have finished downloading and others
started downloading at a later time. The connection to a new swarm and the com-
pletion of a file require handling time by the client that might delay the supply of
data to CoopViz. The variations in CPU utilization also have an impact on the per-
formance of the client and, consequently, of CoopViz. Note that the high resources
utilization illustrated in Figure 4.4 is due to the very high number of connections.
that were being handled by the client during the experiment and is not representa-
tive of a normal Tribler session. In fact all BitTorrent clients, for network and CPU
performance reasons, suggest to limit the maximum number of concurrent active
torrents to 8 and the number of active connections to 100. During the experiments
the client was active at the same time in more than 25 swarms and handled more
than 1200 connections.
Chapter 5

Conclusions and Future Work

For the survival of a P2P file sharing system, peers should upload to the community a number of bytes at least equal to the downloaded data. As discussed in Chapter 2, current figures show that the average peer’s behavior is still very far from this goal. Current bandwidth allocation policies, implemented in P2P clients, are apparently not enough to push users towards cooperation. In order to help solve this problem, this thesis presents the design and implementation of Coopviz. The graphical interface offers file sharing systems’ users insights into health and contribution of peers they are exchanging data with. Information is often a first step to stimulate people to face a problem; CoopViz, by showing the behavior of peers, exploits the function of information channel. On a more psychological note, it visualizes concretely the data exchanges and shows that P2P networks are real communities. CoopViz also offers a novel approach to sharing policies, allowing every single user to be a moderator and be active part in the establishment of bandwidth allocation policies. Finally, the proposed Visualization allows to gather crucial information on the status of the swarms and of the participant peers.

With regards to the requirements highlighted in Section 2.2, Coopviz implementation in Tribler manages to satisfy all the requisites set at the beginning of the project. The interface is graphically pleasant, extremely easy to use and offers a clear overview of peers cooperation. The graphics libraries used allow users of different operative systems to run the modified version of Tribler. Finally and most importantly, CoopViz offers a real time representation of the swarms dynamics and real time peers properties retrieval regardless from the number of active peers in the swarms.

CoopViz is a potentially revolutionary user interface. So far, the only moderated networks are private and show a level of cooperation sensibly higher than traditional file sharing systems. If users use CoopViz with caution, attributing their votes fairly, two important changes in P2P networks could take place. First, bandwidth allocation policies would be better tuned thanks to the users’ inputs. Second, a voting system is much more concrete and ‘scary’ than theoretical bandwidth allocation mechanisms, of which some users have even no knowledge; the constant
free rider, after seeing its reputation going down vote after vote, would probably start thinking about sharing at least a little bandwidth.

During the development of this thesis the following investigations and possible modifications have been left for future work:

- **Dual Visualization, wxPython and PyOpenGL.** Looking at the swarm visualizations offered in other peer-to-peer clients, for example in Azureus [6], and examining the reaction of the public to the implementation, it is evident that the first thing the users seek in a graphical interface is the so-called ‘wow factor’, while performance and learnability are somehow pushed into background. These two requirements have been set as essential in CoopViz, so that users could really use the interface to investigate the contribution of the other peers and cast their vote. Hence the implementation in Tribler only includes a 2D visualization of the swarms, that better satisfies the performance and learnability factors but lacks in graphical impressiveness. It would be advisable to offer the user the possibility to switch between the two and three dimensional visualizations, so to have on one side the functional interface and on the other the more graphically fancy and recreative one.

- **Establishment of Users’ Approach to CoopViz and Vote’s Weight on Bandwidth Allocation Policies.** Before allowing the users to really take active part in the establishment of bandwidth allocation policies, it would be very important to deploy the modified version of Tribler on large scale and monitor the users’ approach to CoopViz. Once these information have been collected, the influence of their vote on Tribler’s bandwidth allocation mechanisms can be established. If the figures show that the public tends to play with the interface out of curiosity, the voting factor should have a minimal impact on the establishment of peers’ reputations. If the majority of peers demonstrates, instead, a fair use of the interface, it might be even possible to give them the privilege of moderators, allowing them to block freeriding peers.

- **Connection with Add as Friend.** Tribler offers the possibility to add specific users as online friends. These can be contacted to boost the download speed of files by using their upload capacity. CoopViz highlights at all times the most generous Bartercast peers; users can access information about the long time behavior of these users, such as how many connection they have accepted and how many MegaBytes they have uploaded. CoopViz is, hence, the best way to establish which peers would be more keen in donating their bandwidth and it would be very practical if users could add peers directly from the interface by right click of the mouse.

- **Latency Monitoring.** The problem of discovering the closest peers in terms of latency is a very active area of research [27, 28, 29, 30]. BitTorrent is responsible for over one third of all Internet traffic [31], of which a large
amount is in fact inter-continental. This causes inefficient network usage and longer download times; it would be more effective to try avoid connection between peers at very different locations. The architecture of CoopViz is very flexible and can easily be modified to represent the latency of peers. Therefore, at every instant, the latency of connected peers within a swarm can be monitored and studied. This would simplify the establishment of bandwidth allocation algorithms that also take into account the latency of peers. This kind of algorithms are proved to increase the networks’ efficiency and shorten the download times. In [30], for example, by biasing BitTorrent communication towards nearby peers, average reductions in download time of up to 25% are achieved along with 12% less traffic going over global transit networks.
Bibliography


