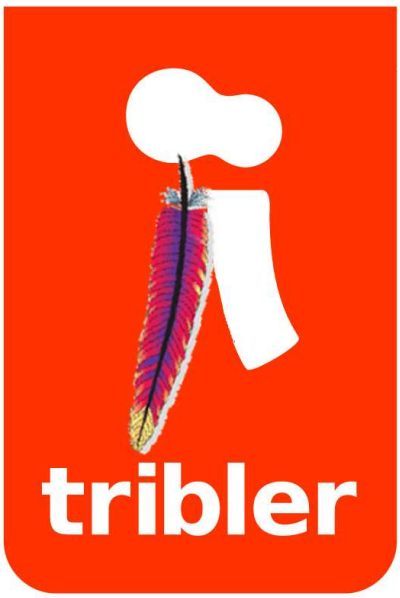
Merging the best of HTTP and P2P

Version of January 11, 2011



Diego Andres Rabaioli

Merging the best of HTTP and P2P

THESIS

submitted in partial fulfillment of the
requirements for the degree of

MASTER OF SCIENCE

in

COMPUTER ENGINEERING

by

Diego Andres Rabaioli
born in Rome, Italy

Parallel and Distributed Systems Group
Department of Software Technology
Faculty EEMCS, Delft University of Technology
Delft, the Netherlands
www.ewi.tudelft.nl

Merging the best of HTTP and P2P

Author: Diego Andres Rabaioli
Student id: 1340794
Email: drabaioli@gmail.com

Abstract

The World Wide Web is growing fast. Web content is growing as well. To cope with this trend, server infrastructures must be able to serve a huge amount of traffic. When this cannot happen, service denial is the consequence. Small content publishers are the most affected by this phenomenon. At the same time, BitTorrent is leading the file sharing world, generating a big part of the network traffic. Therefore combining HTTP and BitTorrent is a candidate solution for dealing with phenomena like *flashcrowds*. But how to merge the HTTP and BitTorrent protocols? How to leverage the P2P network when the only input is the web content URL? This document proposes an approach to support HTTP with the BitTorrent protocol. The architecture of *HTTP2P*, a tool capable of hybrid download, is described in its details. It is shown that the main problem to face when creating such an hybrid is preventing content pollution attacks. The solution to this problem is the Pollution Prevention algorithm presented in this work. The experiment results show that the achieved performance when combining HTTP and BitTorrent is the best of both protocols: slow start-up delay, improved download speed and server load reduction.

Thesis Committee:

Chair:	Prof. dr. ir. H.J. Sips, Faculty EEMCS, TU Delft
University supervisor:	Dr. ir. J. Pouwelse, Faculty EEMCS, TU Delft
Committee Member:	Dr. S. Dulman, Faculty EEMCS, TU Delft

Preface

This thesis is part of my Master of Science in Computer Engineering at Delft University of Technology. This document describes my thesis work performed at the Parallel and Distributed Systems Group of the Faculty of Electrical Engineering, Mathematics, and Computer Science of Delft University of Technology.

I would like to express my gratitude to my supervisor Dr. Johan Pouwelse, for supporting me throughout the project period, enriching me with academical experience. Also I would like to thank Freek Zindel, for introducing me in the thesis project and guide me in the right direction, and Gertjan P. Halkes, Nazareno Andrade and Nitin Chiluka for their precious feedback on my thesis. Finally I would like to thank the Tribler group, that I consider as an inspiration for the P2P technologies world.

Diego Andres Rabaioli

Delft, The Netherlands
January 11, 2011

Contents

Chapter 1

Introduction

The *Web* is growing fast. From the first static pages containing mainly text, a few links and pictures browsed by few users per hour, the World Wide Web evolved to dynamically-changing and media-rich content pages, browsable by sophisticated tools at a rate of nearly a million requests per day.

Web content size is also growing hand in hand with connection speed. One of the recent steps forward in modern browsers has been the integration of video content rendering directly in pages (HTML 5) and not through the use of an external plug-in [11]. This marks the trend of including video content (one of the largest types of content) in web sites. For popular web content distributors this means that the delivery infrastructure has to be able to provide a large upload bandwidth for a huge number of users. Today, technology offers high performance servers and the possibility to cluster these powerful machines in large scale content delivery systems. However, the costs to set up and maintain such systems are very high, and hence not affordable by all content publishers.

Besides web content size, popularity is another factor that could require the content provider to set up expensive content-delivery systems. When a specific web page or web content becomes very popular in a short time (for example *Slashdot* [9] or *Digg*) the delivery infrastructure gets overrun with user requests. This consistent growth in user requests is called *Flash Crowd*. Most of the small web content hosts can not cope with this huge increase of traffic and become temporarily unavailable.

At the same time, peer-to-peer (P2P) protocols are succeeding in the file-sharing world due to their capability to be highly scalable in environments with a large number of users. Files of the magnitude of 1 GB can be spread between 100.000 users in one day. This example gives the idea of the performance of such a network and the potential it can achieve if combined with a web content delivery system. By combining P2P and HTTP protocols, a small web server can handle *Flash Crowds* and satisfy the request for a huge amount of data without any extra hardware cost.

1.1 Flash Crowds

To understand the advantage of merging HTTP with P2P download, the concept of *Flash Crowd* has to be introduced. A *Flash Crowd* is a phenomenon that occurs when a server hosting a web site or a web resource catches the attention of a large number of users, and gets an unexpected overloading surge of traffic [23][14]. To satisfy such a huge amount of requests in a short time the web provider has to supply an adequate amount of hardware resources.

The common strategy to deal with this phenomenon is to add a *mirror* to the web infrastructure. An exact copy of the web content is spread over different machines so that the user requests are distributed and the load is balanced over several hosts. As an example of mirroring, we can consider the well-known SourceForge website that hosts open-source software projects; to maintain download availability to the users, its infrastructure is composed of many mirror sites.

If we consider the cost for setting up and maintaining such an infrastructure, we can conclude that preventing a *Flash Crowd* effect is very expensive and not affordable for every web content provider [28]. Hence, the small publishers are the most affected by the risks of a *Flash Crowd*, where, if not properly prevented by a robust infrastructure, the consequence is running out of service. This problem is even more pronounced when dealing with large content, since the available upload bandwidth of the server could run out with a small increase in user requests.

1.2 Load distribution

Due to the high scalability property of the P2P networks, the BitTorrent protocol can handle the *Flash Crowd* problem very well. In a BitTorrent network, the larger the number of users trying to retrieve the same content, the better the network performs.

The reason why P2P systems are good candidates to solve network congestion problems related to the HTTP protocol is that its good scalability allows the network to adapt itself when congestion is arising; if many users are accessing the web server to retrieve the same content, the P2P integration would allow the users to interact in a cooperative download, bypassing the server in the task of central content distributor and therefore distributing the requests between the users. By its nature, P2P has a low cost infrastructure: no expensive hardware is needed like in the case of a mirror, since the load of each request is spread among the peers in a cooperative way. Last but not least, P2P allows decentralization: when a server providing the desired content is down, the content is unavailable. In the P2P network the content is available as long as there is at least a small set of peers connected.

This is the reason why P2P is succeeding in the file sharing world and why it suits for an integration with the HTTP protocol in the task of web content retrieval.

1.3 HTTP

HTTP is a client-server paradigm protocol designed to retrieve *hypertext* documents (web pages and media content in general) [18]. Its messaging system is based on requests from client to server and responses from server to client. In an HTTP session, the client establishes a (TCP) connection with the server, then it sends the request for a document, the server replies with a response (generally containing the document), and at the end of the session the connection is closed. The request is immediately satisfied by the server with a positive or negative response. The file transfer in a conventional network starts immediately after the client request and remains more or less constant. Figure 1.2 describes the bandwidth behavior of an HTTP file download. The resource to be accessed by the client is defined by the URL which is mapped by the server into the physical file or dynamically generated content.

Figure 1.1: *Client-Server architecture*

1.4 BitTorrent

BitTorrent is a P2P file sharing protocol designed to distribute digital content between peers [3]. Unlike HTTP, in a P2P network architecture, a peer is both a client and a server at the same time. It usually trades pieces of information with other peers. In the case of BitTorrent we can simplify that a peer trades part of a file with another part of the same file. When a peer wants to download a file, it joins a swarm (a group of peers sharing the same file) by asking the tracker (a server

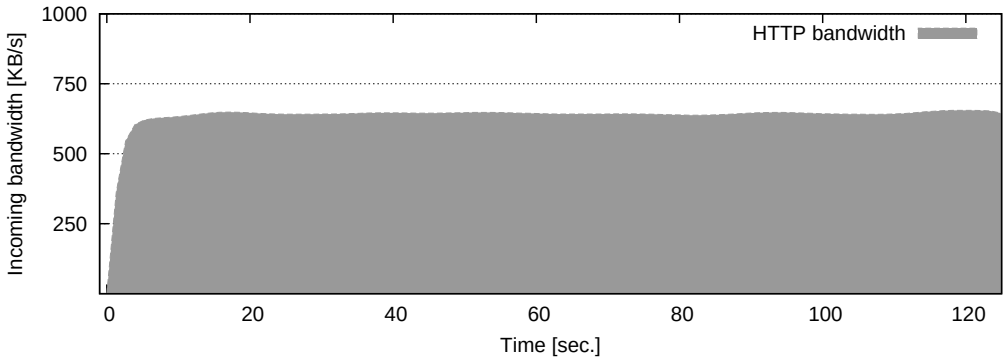


Figure 1.2: *HTTP bandwidth behavior*

keeping information about peers currently connected) the network address of peers in that swarm. The peer then connects to a set of peers participating in the swarm and starts trading pieces of the file.

Unlike HTTP, in BitTorrent, the start of a download is delayed by many steps: first connecting to the tracker and retrieving the peers' addresses, then connecting to the peers in the swarm, finally waiting to receive pieces from the peers when they are interested in trading.

The resource that a client-peer wants to access is described in a torrent file, containing information about the tracker address, file size, piece size, file name and other information needed to instantiate a download session. From the *info* field in the torrent file, the *info-hash* is computed. This value uniquely identifies the resource over all the BitTorrent network. For this reason, this parameter is used during the protocol hand-shake: when two clients initiate a connection, and during the *swarm discovery* step, when the peer asks the tracker the addresses of the peers participating to the interested swarm. Therefore, the *info-hash* can be viewed as the swarm identifier.

Since the peers in a swarm could be potential attackers distributing fake content, a content integrity mechanism is needed. The hashes of each piece, computed during the torrent creation, are contained in the torrent file; once a piece has been retrieved by a peer, the hash of that piece is computed and compared to the same hash contained in the torrent file. The hash checking step reveals whether the piece is fake or not.

The advantages of a P2P network over a client-server infrastructure are:

- distributed content: the content does not rely on a single central entity but on many distributed entities. This makes the content availability more robust during huge surges of requests, in contrast with the web servers' stability. When a web server can not supply to the number of requests, the service is denied and the content unavailable.

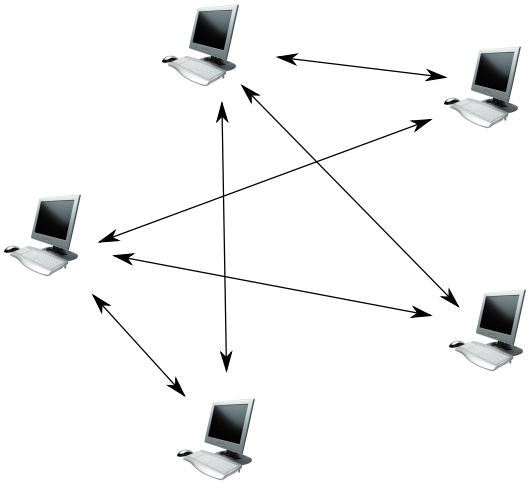


Figure 1.3: *Peer to Peer architecture*

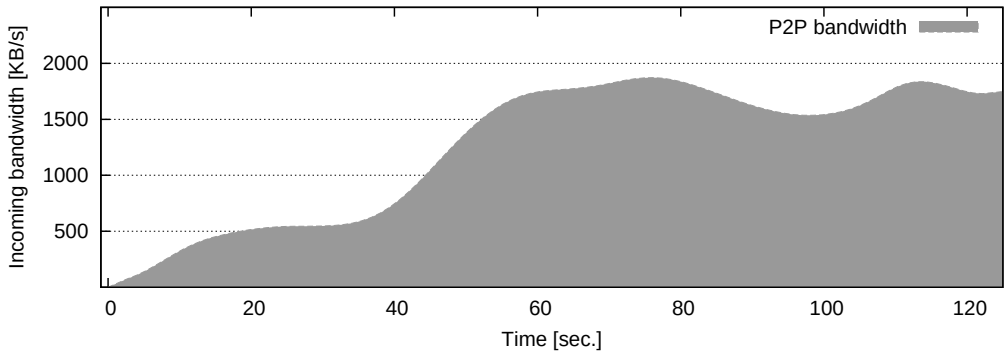


Figure 1.4: *P2P bandwidth behavior*

- scalability: the more peers participate in a swarm, the better the network performs. In the case of a web server, a big number of clients result in a high server load, reducing the delivery performance.

- cost effective: building up a system able to satisfy an amount of requests of the magnitude of YouTube would have a strong economical impact with a client-server architecture. The same amount of requests can be satisfied with a P2P solution at extremely low costs.

1.5 The best of two worlds

This thesis project aims to support HTTP with the BitTorrent protocol in the task of web content retrieval. In particular, the target is to combine the qualities of the

two protocols in such a way that the resulting performance matches the best of both worlds. The following table describes which are qualities of each protocol that we want to keep:

HTTP	BitTorrent
Simplicity	Highly scalable
Low latency data transfer start-up	Low cost infrastructure
Fast convergence to full-speed	Decentralization

The result of this merge will allow a client to start a P2P download with the start-up latency typical of an HTTP file transfer, a download speed equal to the sum of both bandwidths, and will allow a low performance server to provide large web content while satisfying a high number of concurrent requests. Figure 1.5 shows the resulting architecture of an HTTP/P2P hybrid system.

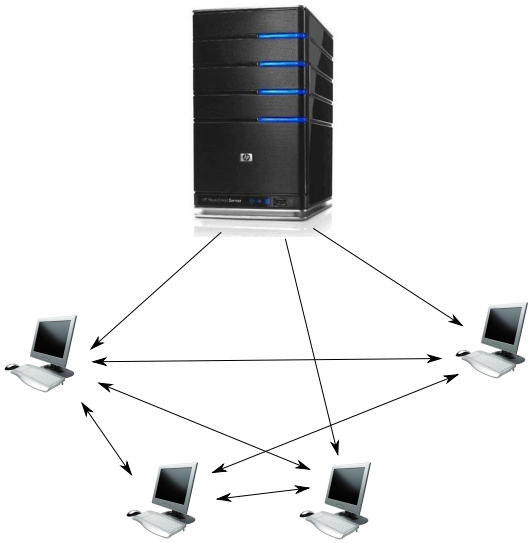


Figure 1.5: *HTTP/P2P Hybrid architecture*

1.6 Thesis outline

The next chapter will focus on the technical problems faced when creating a tool able to combine HTTP and BitTorrent. Chapter 3 presents the architecture of HTTP2P, a HTTP/BitTorrent download tool, focusing on the main implementation details. Chapter 4 represents the scientific contribution of this thesis work: the *Pollution Prevention* algorithm, a technique able to guarantee security over the received untrusted data in a torrent-less and tracker-less environment. Chapter 5

analyses some of the main implementations created during the project period and shows the results of the experiments performed with these tools. Finally, the last chapter presents the conclusions.

Chapter 2

Problem Description

This chapter explains the challenges that are faced for building a system that combines the best of the HTTP and BitTorrent worlds. Even if the two protocols are designed to retrieve files from internet, the environment in which they operate is different. This introduces some incompatibilities in the messaging system of the protocols. While HTTP is designed to retrieve a file from a central repository, BitTorrent is designed to retrieve and retransmit pieces of a file in a distributed environment formed by many entities. How to join the divergences presented in this chapter is the scope of this thesis work.

This chapter starts with a list of goals, representing the requirements to meet, followed by an overview on the previous attempts in the task of creating an HTTP/P2P hybrid. The challenges that are faced to address these goals are presented in detail. The chapter concludes with the description of some techniques that will be the key of success toward the creation of an HTTP/P2P hybrid.

2.1 Goals

The goal of the thesis is to build a system that is able to scale on the number of clients, handling flash crowds efficiently and that is low cost for content publishers. Following are some of the advantages of a system that combines the best of the HTTP and BitTorrent protocols:

- **Server load reduction**
  This is one of the most significant benefits that the P2P-HTTP combination would bring. The more clients try to download the same resource, the more peers are available to share content and the less data needs to be retrieved from the web server.
- **Increased download performance**
  If the user's download bandwidth is higher than the server upload bandwidth, then the user can add the P2P download speed to the HTTP one.

Lets consider an example on which the server bandwidth limit is 200 KB/s and the P2P download bandwidth is 400 KB/s (pretty common values). Hence the total speed that is possible to achieve in this case it is 600 KB/s, 3 times faster than the server bandwidth limit.

In general the total download speed cannot be lower than the HTTP one.

- **Low cost publication**
  As it has been explained in the first point, by adopting this download strategy, the server load would be extremely reduced, therefore it is no more necessary to have expensive server infrastructures to deliver large content.

  Therefore, low-cost publishing is possible. Setting up a video or large content delivery system is just a matter of delegating the task of distributing the large content to the P2P network and delegating the task of providing just the web page and the content access key to the web server.

- **Easy configuration**
  Server configuration requires minimalistic changes such as limiting the upload bandwidth to the desired value, if that is considered necessary. Once this limit is set, all the excess bandwidth requested by users is compensated automatically by the P2P network. In general, no server adaptation is required and backward compatibility is guaranteed for servers and clients.

- **Zero-delay P2P video streaming**
  As Tribler demonstrates [10], the BitTorrent protocol can also be used for video streaming [25]. The weak point in P2P video streaming is the video playback startup time, because discovering and getting download bandwidth from other peers introduces a large delay. By implementing our new architecture, the playback of a video could start as soon as the web server gives the stream, while the torrent engine is still in the swarm discovery phase (looking for peers in the swarm). Once the BitTorrent bandwidth is fast enough, the web server can be offloaded. This example highlights the 'best of both worlds' quality of this approach: the responsiveness of HTTP and the load distribution of P2P.

- **Backward compatibility**
  This technique does not need any support on the server side. This also means that clients unaware of the P2P support can still access the web server in the conventional way. This would lead to an easy and fast adoption of this download strategy without relevant impacts.

As a last remark, it is to be emphasized that this download solution is intended to have a higher advantage for the server in situations where the number of client

requests is high, since the load on the server is distributed among the clients. If the number of requests is low, the server is sufficient in most cases to satisfy clients' requests.

2.2 Prior work

There have been many attempts to support the HTTP protocol download with P2P. The most significant one proposes to include an *X-Torrent* header in the HTTP protocol [7]. The *X-Torrent* header specifies a link pointing to a torrent file that provides the same content that the user is trying to download through the browser. If the client does not receive bandwidth because the server load is too high, the browser automatically switches the download source to the BitTorrent one, offloading the server from supplying the data. The weakness of this approach stands in the fact that the HTTP protocol has to be extended generating backward compatibility issues for servers and clients not able to deal with the new HTTP header. Moreover, it is not possible to merge the data retrieved from the different protocols but just one of them must be chosen at any given time; in short, if the web server gets overloaded by a *Flash Crowd* effect while the user is downloading through HTTP, the download has to be started from scratch through the BitTorrent protocol.

The other solution trying to support HTTP with P2P that deserves mention is the FireCoral Network Project [5]. FireCoral is intended to enable P2P exchange of browser caches between clients. Its architecture consist of browser clients (or peers), origin servers that publish content, trackers that store peering information and content metadata, and cryptographic signing services that authenticate content. The drawback of this design is that it is not a completely distributed architecture; the peer discovery process is delegated to a central tracker and the authentication of the data received from other peers is performed by asking a cryptographic signing service for the signature of each specific block. On the one hand this technique reduces the load of the web server, on the other hand it moves the load to the tracker and mainly to the signing service, responsible for sending each piece's hash to each peer. If the tracker or the signing service is down the whole system cannot work.

These examples are mentioned here to show that P2P is considered to be a solution to network congestion problems and that many efforts are concentrated in trying to combine HTTP and P2P [19] [22] [27] to build a scalable system able to resist to phenomena like *Flash Crowds*.

2.3 Metadata and swarm discovery challenges

In this section we explain what challenges are faced when creating a tool that supports HTTP with the BitTorrent protocol in the task of web content retrieval. As explained later in section 2.4.1, *Web Seeding* solves the problem of supporting P2P

with HTTP; this is simpler since the presence of a torrent metadata file includes information useful to start a connection to the server in a simple way. When supporting HTTP with BitTorrent, the torrent file is missing due to the nature of HTTP. The lack of information contained in the torrent metadata makes the process of using the P2P network more difficult. This section gives some more details about these challenges.

2.3.1 Request translation

Since HTTP and BitTorrent are two different protocols designed to operate in two different environments, there is some incompatibility to solve when trying to merge them. The nature of HTTP is to send only one request to the server and to download a file sequentially from the beginning to the end, while BitTorrent downloads a file in scatter order depending on some policy, typically rarest first, and sending one request for each piece of the file to download.

2.3.2 Swarm discovery

In a conventional BitTorrent session, the client joins the swarm, as previously explained, by asking the tracker for a list of peers currently participating to the swarm, and establishing a connection with each of them by performing a hand-shake. During the hand-shake several pieces of information are exchanged; most important of all is the info-hash. Both the tracker address and the file info-hash are contained in the torrent metadata file. In this scenario, since the only input is the URL of the resource that the client wants to download, there is no torrent file and therefore no tracker address and no info-hash. This is critical information to join a swarm and to start a connection.

2.3.3 Security and pollution prevention

Hash checking is a difficult issue to solve in a scenario where there is no torrent file containing piece hashes. Since the hashes are not known, a strategy to ensure content integrity and to prevent the injection of polluted data into the system is required. On one hand, the content integrity is a requirement for ensuring that the data we are downloading is correct. On the other hand, it is also needed to ensure that the data we are injecting in the system is not polluted.

For this purpose, it is important to understand the concept of a *trusted authority*. The trusted authority in a conventional download is the torrent file itself since it both contains the swarm identifier to be used to join the swarm and the piece hashes to use to ensure content integrity. In a torrent-less and tracker-less environment the only trustworthy authority is the web server; comparing a piece received from a peer against the same piece retrieved from the server, reveals whether the content received from the peer is good or not.

It is obvious that we cannot compare all the data received through the P2P network against the web server's content, since this would lead to data download duplication, conflicting with the goal of this thesis. Ensuring content integrity while retrieving the minimum amount of data from the server is perhaps the biggest challenge faced in this environment.

2.4 Existing techniques

The following techniques are used to face the problems previously described. These techniques are presented here to prepare the ground for the next chapter in which the architecture of HTTP2P, a tool able to support HTTP with the BitTorrent protocol, is explained in detail.

2.4.1 Web Seeding

Web Seeding is the ability of a server to act as a BitTorrent seeder [3]. There is an important distinction to remark here: if *Web Seeding* is meant to use HTTP to support P2P, this thesis intends to use P2P to support HTTP. However the solution that is going to be proposed in the next chapters will make use of the *Web Seeding* technique to retrieve the data from the HTTP side.

There are currently two main specification proposals for implementing *HTTP Seeding*: the GetRight style [6] and the John Hoffman style [21]. The first one is transparent to the web server but includes a heavy algorithm to download the biggest gaps (adjacent pieces); the reason, as explained in the main specification, is that many requests of the same file can be considered an attack by the server. The John Hoffman's style is included as part of the BitTornado client and consist of translating a piece request into a URL string, concatenating info-hash, piece index and offset in the URL query; however it requires a script on the server side to parse the request, raising backward compatibility issues.

2.4.2 Merkle hashes

The building block of the security architecture is the technique of ***Merkle hashes***. Hence, to fully understand how the *hash checking algorithm* works, we need to introduce some background on *Merkle hashes* [15].

In BitTorrent, piece checking is needed to guarantee content integrity. Each piece is checked against the piece's hash contained in the torrent meta-data file. If the piece's size is too small compared to the file size the meta-data file will be large, increasing the load of the server distributing that particular torrent. On the other hand, very large pieces decrease the ability of peers of bartering since a larger time is required to download enough data to start a trade.

A solution to these two problems is to replace the list of digests with a single Merkle hash. A Merkle hash can be used to verify the integrity of the total content file as well as the individual blocks via a hierarchical scheme. From the content we construct a hash tree as follows. Given a piece size, we calculate the hashes of all the pieces in the set of content files. Next, we create a binary tree of sufficient height, using the piece hashes as leaves. Finally, we calculate the hash values of the higher levels in the tree, by concatenating the hash values of the two children and computing the hash of that aggregate. This process ends in a hash value for the root node, as shown in figure 2.1, which we call the root hash [1].

The root hash along with the total size of the content file and the piece size are now the only information in the system that needs to come from a trusted source. A client that has only the root hash of a file set can check any piece as follows: it first calculates the hash of the piece it received. Along with this piece it should have received the hashes of the piece's sibling and of its uncles. The highlited block of figure 2.1 correspond to the hashes received within a message. Using this information the client recalculates the root hash of the tree, and compares it to the root hash it received from the trusted source.

2.4.3 DHT

A distributed hash table (DHT) is an hash table distributed over several nodes. Each node maintains a part of the hash table and a list of other nodes addresses (routing table). The totality of all the nodes forms the overlay network on which the DHT operates, and the totality of all the content maintained by each node is the DHT itself. The DHT holds a set of (key, value) pairs and provides the possibility to store new pairs and a look-up mechanism to retrieve a *value* given its *key*. The main DHT property, apart from the obvious scalability and decentralization properties, is the ability of dynamically change its topology, guaranteeing fault tolerance over nodes arrival, departure and failure [4].

DHT technology has been adopted as a component of BitTorrent to implement *Distributed Trackers*. This technique uses DHTs to store (swarm_ID, peer_list) pairs in an overlay network. In this way the DHT can be used to replace the centralized tracker in the task of swarm discovery. There are a few advantages in this technique: the load of the tracker server is distributed over the nodes, avoiding service denial in flash crowd situations, and there is no need to know which tracker stores the peer list for a particular torrent. In this way any swarm can be joined without the need to know the tracker address, usually stored in the torrent metadata. This is one step ahead towards decentralization.

Typical functions of DHT implementations are [2]:

[1] The hashes are computed using the SHA1 cryptographic hash function which generates 256^{20} (2^{160}) different values. Therefore the probability of a collision when computing the root hash of different content files is minimal

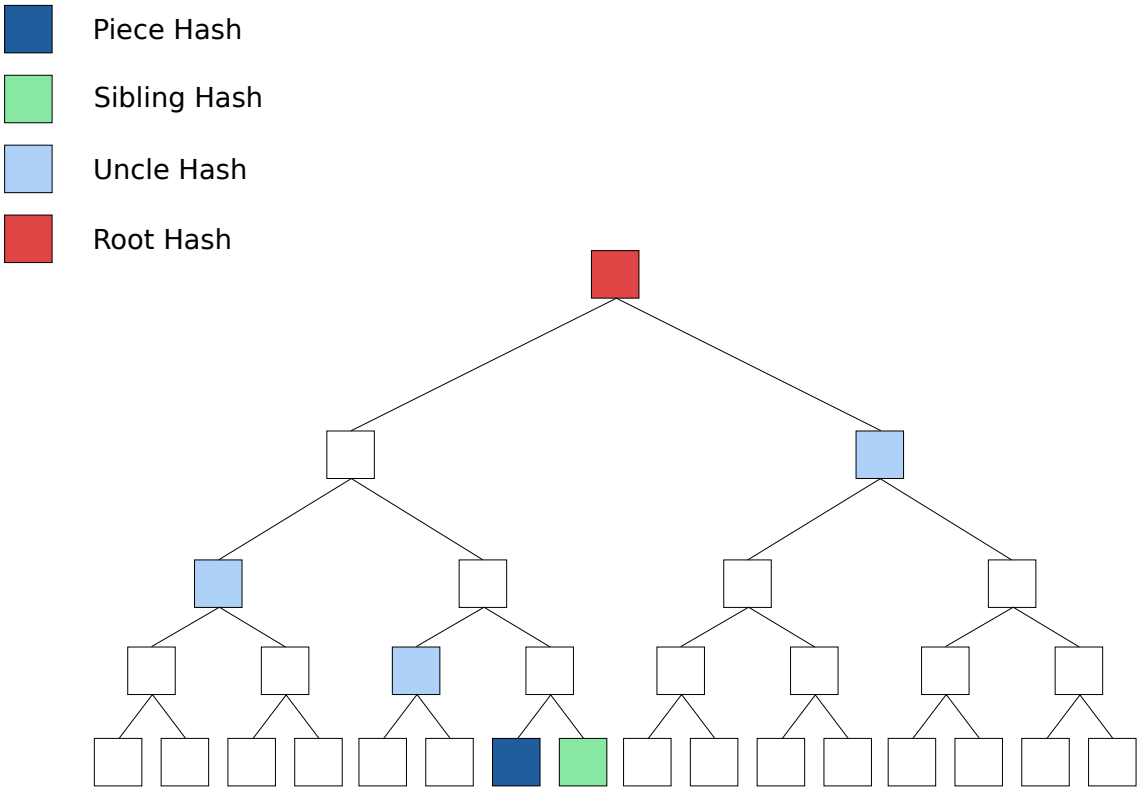


Figure 2.1: *Hash, sibling, uncles and root hash received along with the relative piece.*

- announce_peer : announce to the DHT that the peer joined a particular swarm.
- get_peers : retrieve from DHT the list of peers that are participating to a swarm.

As you can see these are also the main functionalities of a tracker. After the first adoption of a DHT in 2005 in the Azureus client [1], all main P2P clients now support DHTs as look-up system for swarm discovery, even if central tracker is still the most used technique for swarm discovery.

Chapter 3

HTTP2P Architecture

This chapter describes the architecture of *HTTP2P*, a tool that supports HTTP download with the BitTorrent protocol. HTTP2P is based on the existing techniques described in the previous chapter: web seeding for HTTP request translation, DHT for swarm discovery and Merkle hashes for pollution prevention. An implementation of each of these techniques is integrated in the Tribler core; the base of this software system.

Our solution is based on two main components: a browser plug-in and a background process. The plug-in handles the user interaction, while the *Background process* is responsible for the file download.

The first section explains how the challenges presented in the previous chapter are addressed. The general design guides are then presented. Finally, the architecture of HTTP2P is described in detail.

3.1 Metadata and swarm discovery solutions

3.1.1 Request translation

In the previous chapter, we presented the two main specifications for dealing with *Web Seeding*: the *Hoffman's style* and the *GetRight style*. In particular, the *GetRight style* specification document explains how many requests to the same resource can be considered as attacks by the web server. For this reason, implementations based on the *GetRight's style* perform HTTP requests with a byte-range as wide as possible, with the intent of performing the smallest number of requests. A few experiments, however, revealed that modern web servers (Apache was taken into account) allows several byte-range requests to the same resource, without considering the client as an attacker and blocking the connection. A *Hoffman style* implementation of the web seeding feature was included in the Tribler core before this project. However, as already mentioned, the *Hoffman's style* requires server side support.

The final implementation of the HTTP seeding module is the result of combining

the two styles: starting from the *Hoffman's style*, because its implementation is already included in the Tribler core, and converging it toward *GetRight's style*, since it does not require any server support. The result is a Python module that performs byte-range requests to the server, where in each range, the amount of data asked is equal to the piece size.

Exploiting the *Web Seeding* technique to retrieve HTTP content implies that the web server is considered as a peer by the BitTorrent engine, translating BitTorrent protocol requests into HTTP requests. The server responses are then parsed and the content retrieved is merged with that of BitTorrent. Following is how the BitTorrent *request* message is translated into a HTTP request:

BitTorrent request message:

```
<len=0013><id=6><index><begin><length>
```

HTTP request message:

```
GET /content_file HTTP/1.1
Host:  web_server_address
Range:  bytes=start_byte-end_byte
```

with:

```
start_byte = ( index * piece_size ) + begin
end_byte = start_byte + length
```

3.1.2 Swarm discovery

In a conventional BitTorrent environment, the swarm discovery process is performed by retrieving a list of peer addresses from the tracker. Which tracker to contact is specified in the torrent metadata file. An environment without a torrent file introduces the problem of finding the correct tracker to join a particular swarm.

Distributed tracking is an alternative solution when the tracker is unknown. As explained in the previous chapter, a DHT is a table of key/value entries distributed over the network [24]. The BitTorrent infrastructure exploits this mechanism to store the mapping of info-hashes/peer-addresses in the same way it does with trackers. It is easy now to understand that the information needed to join a swarm, is a key that the DHT will map into a list of peers. This key represents the swarm identifier. Since the only input we have is the URL of the web resource to download, the input key for the DHT query will be the SHA1 hash of the file content URL. This hash will be used also during the handshake step, to start the connection with other peers.

Since the info-hash used by the HTTP2P client for announcing to the DHT is different from the conventional BitTorrent one, the swarms created by the two clients

for the same content file will be different. This also means that the clients using the HTTP2P protocol extension will have to set a specific bit in the handshake extension-bytes to avoid protocol-message conflicts during the download.

BitTorrent handshake message:
<19><"BitTorrent protocol"><conventional_extensions><info_hash><peer_id>

HTTP2P handshake message:
<19><"BitTorrent protocol"><HTTP2P_extension><SHA1(URL)><peer_id>

The same difference between conventional BitTorrent handshakes and HTTP2P handshakes applies also to Merkle hashes extension. However, HTTP2P and Merkle hashes protocol extensions uses different hashes only for the handshake message. The *piece* messages[1] still uses the Merkle tree root hash in both extensions (see *Pollution prevention* chapter). Further studies can lead to a strategy to unify this little "handshake" difference.

3.1.3 Security and pollution prevention

The base block of security is the technique of Merkle hashes. In the absence of a torrent file providing the piece hashes, the hashes will be traded among the peers in the swarm. However, unlike Merkle hashes technique, there is no trusted root hash against which the received *piece* messages can be checked. The lack of a trustworthy root hash is the biggest problem to face to guarantee content integrity. Preventing content pollution, derived by fake block attacks, is not possible without a technique that ensures security over such a network. This thesis proposes the *Pollution prevention algorithm* as the solution to this dilemma. The whole next chapter is dedicated to explain the algorithm in detail.

3.2 Design considerations

3.2.1 Minimal impact

When creating a software system that has to operate over an existing environment, one of the main goals is keeping *backward compatibility*. This means that the new system has to be integrated in the existing environment with minimalistic changes, causing minimal impact and trying to keep the previous system as much unchanged as possible.

To prove the importance of this concept, lets take as an example the *X-Torrent* http-header extension proposal. The reason why such a solution has not been

[1] In reality, Merkle hashes and HTTP2P protocol extensions use the *Tr_hashpiece* message instead of the conventional *piece* message.

adopted yet is because of backward compatibility issues. To allow the integration of this extension, web servers need to be able to detect system overload and send the new header parameter in case of a flash crowd; at the same time, clients unaware of this extension have to be able to ignore the new header. This example shows that the impact for adopting the *X-Torrent* extension is bigger than the benefits it brings.

For this reason, the minimal-impact factor was one of the main concerns when designing HTTP2P. Web servers must stay unchanged; normal browser agents have still to be able to access the web resource in the same way. Avoiding server side support is a main goal, as well as the cause for making an HTTP/BitTorrent hybrid a complex task.

3.2.2 Desired behavior

An ideal solution for our system should not impose any change on the server side and operate transparently for the user. The user does not need to have any knowledge about the system he is using; hybrid download should happen as when normally browsing the Web. At most, the user is required to decide whether to enable the P2P support or not. User friendliness is usually the main target when designing user interfaces. For this reason, the UI has been reduced to the minimum in the design of HTTP2P. *Right-click* on the link to download, *click* on *Hybrid download* from the menu panel, is the only interaction with the system required.

The layer of complexity, involving the way the content is retrieved, should be invisible to the user and to the web server. In the background, the system will start the HTTP download, announce itself to the DHT, join the swarm (if any), and start trading pieces of the content file with other peers; all of these steps must not require the user interaction or server changes.

These requirements lead to the design of two main components: a minimalist browser plug-in, that has the only responsability of grabbing the user selected URL link, and a *Background process*, responsible for turning the URL received from the plug-in into a file on the disk. How these two components cooperate is described in detail in the architecture section.

3.2.3 Procedure steps

Summarizing the considerations explained in the previous sections, we synthesize below what the procedure to perform a hybrid download is:

1. The user, who wants to download a file, clicks the link embedded in the web page.

2. The browser plug-in grabs the click event and sends the URL of the download to the *Background process* (*BG*).

3. The *BG* parses the URL received from the browser plug-in and starts the HTTP download.
4. The *BG* computes the *SHA1* hash of the URL. This value represents the *swarm ID*.
5. The *BG* performs a DHT query with the *swarm ID* as key parameter to search for a swarm distributing the same file retrieved from the web server.
   - If the swarm does not exist, the file will be retrieve entirely from HTTP.
   - At download completion, the *BG* computes the Merkle tree out of the downloaded file and announces to the DHT.
6. The *BG* starts the connection with the peers in the same swarm.
7. The *BG* starts trading pieces retrieved from HTTP with other peers by sending and receiving Merkle *piece* messages.
8. Along with each *piece* message, the hashes of the Merkle tree are received. The Merkle tree is built out of the hashes received from other peers and the hashes computed from the HTTP retrieved content.
9. The integrity of the received content is performed by the *Pollution prevention* module included in the *BG*.
10. The download completes.

Figure 3.1 shows a simplified representation of the procedure steps where 1) the BG process receives the URL from the browser plug-in, 2) the HTTP download starts, 3) the swarm discovery is performed through DHT and 4) the BitTorrent download starts.

3.3 Architecture

3.3.1 Tribler core

The system architecture is based on the Tribler core, an open source BitTorrent client developed by the Parallel and Distributed System group of TU Delft. The Tribler core is based on python and offers a framework able to provide P2P functionalities and many other features (DHT, web seeding, Merkle hashes, gossip protocols, etc.). It provides a simple API through which the developer can easily build P2P based applications. A simple-to-use API interface and extended features allows the Tribler core to be a stable base on which can be developed the HTTP2P system.

As previously described, HTTP2P is divided into two main components: the browser plug-in and the *Background process*. Since the browser plug-in is a minimal

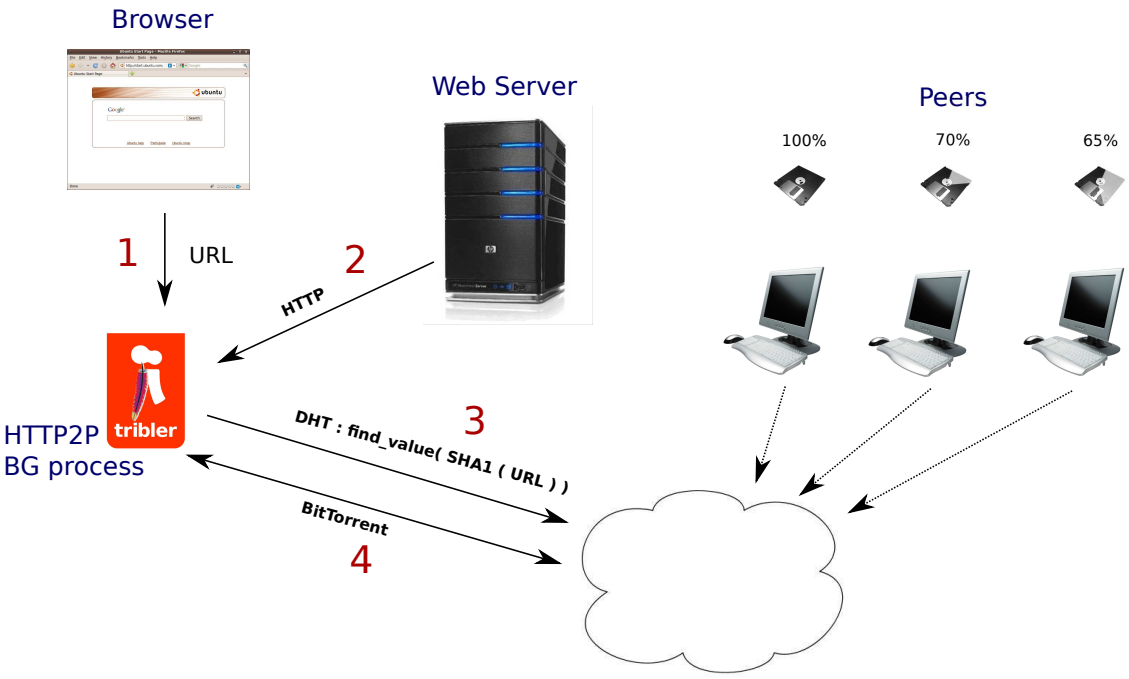


Figure 3.1: *HTTP2P download procedure steps*

component, with the only responsibility of sending the selected URL to the BG, and its implementation should be specific to the browser in which it is embedded, the attention is mainly concentrated on the architecture of the BG.

The BG is a software layer on the top of the Tribler core. Its functionality is to parse the URL received from the browser plug-in and to instruct a Tribler *session* to perform a download. A Tribler session implements all the operations to be performed for retrieving the content from the BitTorrent network.

Since the Tribler core was not designed to perform all the steps required by HTTP2P, it is necessary to include some changes into the core, involving mainly the Pollution prevention algorithm procedures. These changes are described later on in the *In-core flow* section.

3.3.2 Connection interfaces

As described in figure 3.2, HTTP2P has to interact with different actors: the web server, the browser, DHT and the BitTorrent peers. HTTP2P interacts with these actors through specific interfaces. The communication with the browser plug-in is handled by an interface layer in the BG. The browser plug-in initiates a TCP connection with the BG *Browser interface* module and sends the URL request. The communication protocol between these two components is reduced to **GET url**

from plug-in to BG and **OK** or **NOK** from BG to plug-in. This very simple protocol can easily be extended with other messages, like **STATUS x%** to warn the plug-in about the status of the download; however this is out of the scope of this thesis.

The interaction with the DHT is handled in the Tribler core by the *mainlineDHT* module. After a *bootstrapping* phase needed to join the DHT network, the DHT module sends *StoreValue* messages to announce to the DHT and *GetValue* messages to retrieve the list of peers participating to the swarm. Both messages require a parameter, usually the info hash of the torrent; in HTTP2P, the SHA1 hash of the URL string is used for this purpose.

The *Downloader* module handles the connection with BitTorrent peers, while the *HTTPDownloader* is the module responsible for handling the connection with the web server, and for translating BitTorrent messages into HTTP messages. Since the HTTPDownloader interface is the same as the Downloader one, the web server is seen as a regular peer by the core. However, because of the client-server nature of the HTTP protocol, the HTTPDownloader translates only the BitTorrent *request* message into HTTP *GET* message where the *byte-range* header field is set to the first and last byte of the piece request, as previously described in the request translation section. Other BitTorrent messages (*keep-alive*, *unchoke*, *bitfield*, etc.) have no meaning in HTTP.

3.3.3 In-core flow

The previous section described the interaction of the HTTP2P *Background process* with the external world. To complete the description of HTTP2P architecture, we analyze here the internal interaction between the modules involved in a hybrid download.

Figure 3.3 shows the internal relation of the modules involved. The highlighted block is the *PollutionPrevention* module. The *PollutionPrevention* module has been added to the Tribler core to implement a security feature otherwise missing in the hybrid download environment. As it will be described in the next chapter, the *PollutionPrevention* module coordinates the various modules with the scope of guaranteeing content integrity.

Once the *DHT* module retrieves the list of peer addresses in the swarm, the peer addresses are passed to the *Encoder* which initiates the connections with the swarm participants. The connected peers will be sharing a root hash of their Merkle hash tree. As it is possible to observe, the *Encoder* module keeps a list of different root hashes in relation with different sets of peers. This structure is needed in case of attack, where we might be connected to one or more sets of peers sharing fake root hashes.

When receiving pieces from peers, the *PollutionPrevention* module will perform several checks to guarantee content integrity. In case of suspected attack, the *Pollu-*

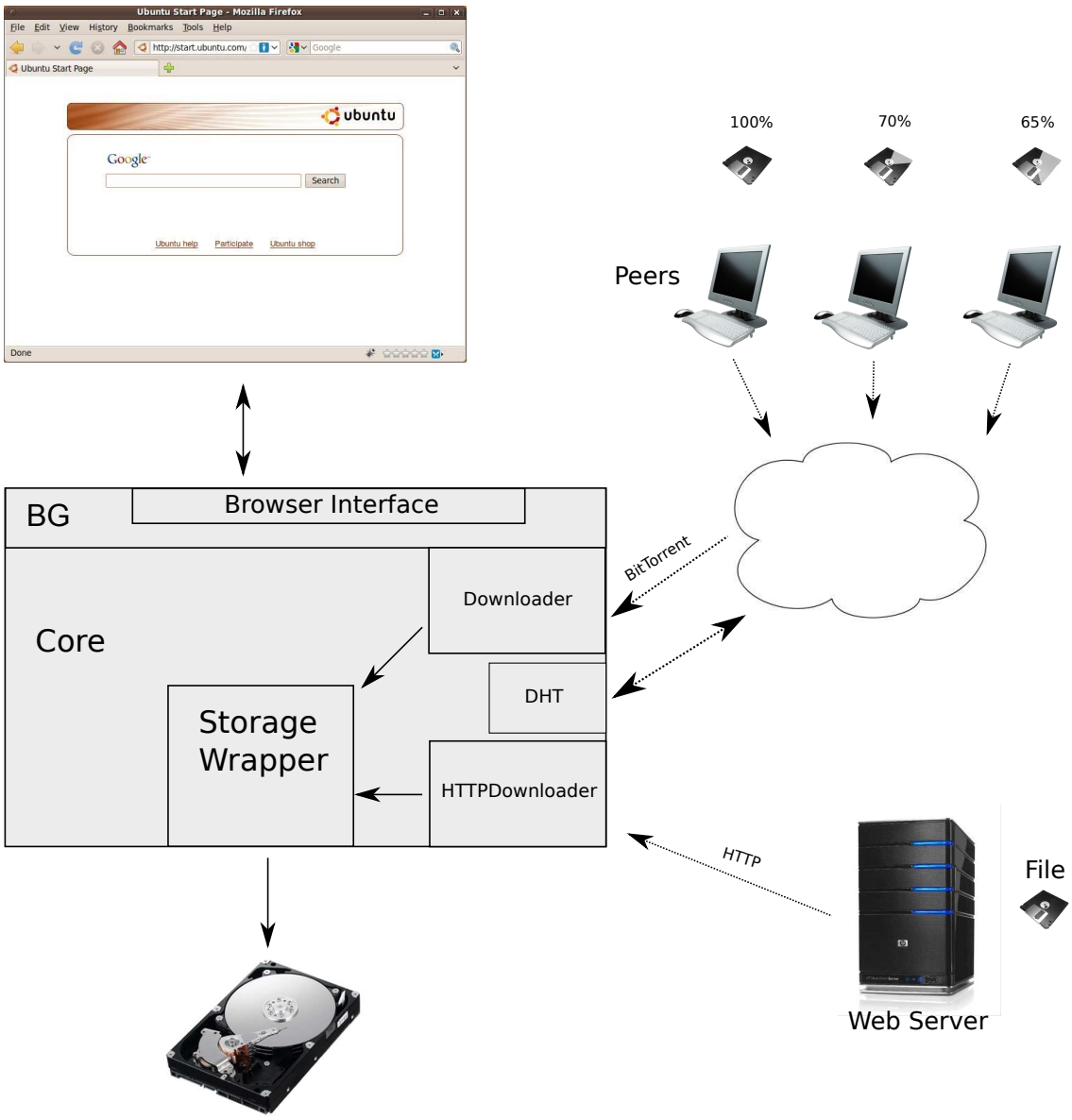


Figure 3.2: *HTTP2P top view architecture*

tionPrevention module will instruct the *PiecePicker* on which pieces to download. A good piece selection in this case will help the *PollutionPrevention* module in finding the attackers. Once the attacker peers are found, their connection will be dropped by the *Encoder*.

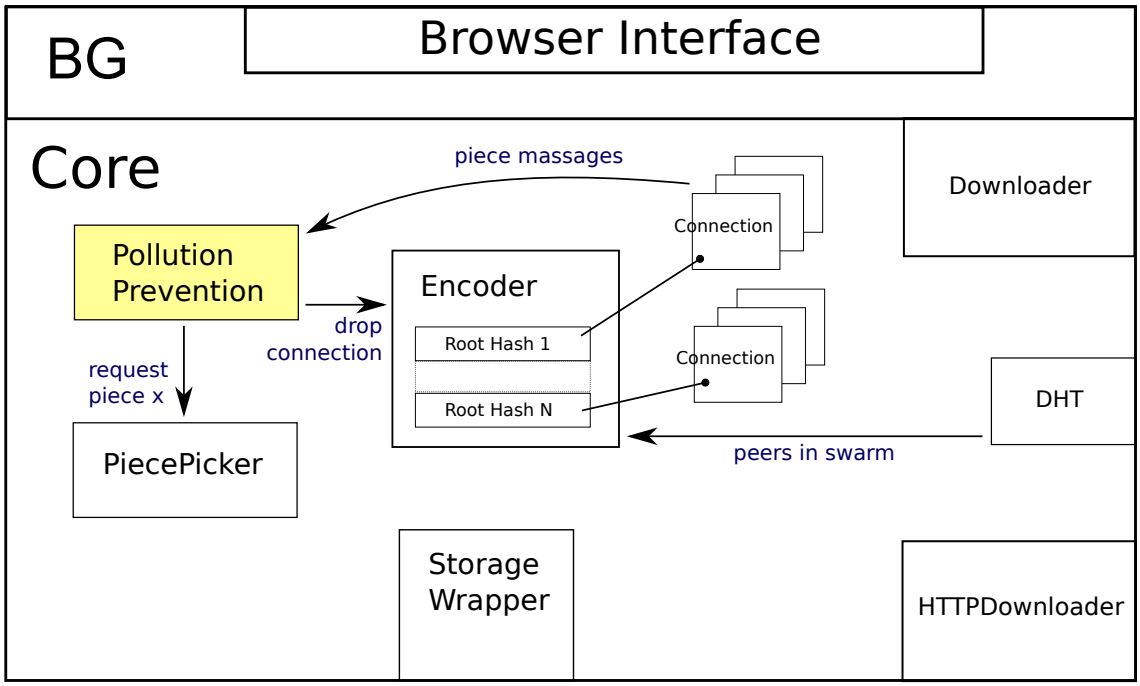


Figure 3.3: *HTTP2P internal architecture view*

3.4 Summary

In this chapter we addressed the main problems to face when creating an HTTP/BitTorrent hybrid: 1) the swarm discovery problem is solved by the use of DHT; 2) the request translation is performed by exploiting the HTTP seeding feature; 3) content integrity is guaranteed by the introduction in the Tribler core of the PollutionPrevention module, implementing the Pollution Prevention algorithm described in the following chapter.

After presenting the main design consideration, we introduced the architecture of *HTTP2P*, a tool capable of HTTP/BitTorrent hybrid download. The system is composed by two main components: a browser plug-in, responsible for the user interaction, and the Background process, a software layer built on the top of the Tribler core, responsible for performing the hybrid download.

In the following chapter will present the details of the tasks performed by the *PollutionPrevention* module.

Chapter 4

Pollution prevention

The key challenge in a torrentless environment is preventing content pollution. Without piece hashes, the integrity of each piece retrieved cannot be verified. If integrity cannot be guaranteed, polluted content may nullify the effort of a download session and, from a broader perspective, all the P2P infrastructure. The lack of a technique that can guarantee security is the main issue to be solved in an attempt to support HTTP with BitTorrent.

Due to the constant attacks in the BitTorrent network, addressing security has been the subject of studies in the recent years and has motivated the creation of different techniques, later include in the main BitTorrent clients. As an example, blacklists of malicious peers addresses have been introduced in clients like uTorrent or Transmission to prevent swarm attacks by the injection of fake content. This is evidence that P2P attacks are a real problem to deal with in BitTorrent environments [20].

This chapter presents an algorithm to guarantee security in an environment where the piece hashes are not known in advance and the only trustworthy authority is the web server. The problem of the fake block attack is presented, followed by a detailed description of the Pollution Prevention algorithm, rotten branch detection and invalidate message techniques.

4.1 Fake block attack

The fake block attack refers to the injection of incorrect content in a swarm [17]. Each piece of a torrent download file is composed of many blocks. Hash checking is performed over a piece; if one of the blocks in a piece is corrupted, the whole piece is discarded and the download of that piece has to start again. Therefore, injecting a small quantity of polluted content causes a big part of the download bandwidth to be wasted.

In a conventional BitTorrent environment, the file download completes correctly despite the attacks. In case of an environment without piece hashes, the download of

a fake piece results in the corruption of the file downloaded, since no piece integrity checking can be performed. This is the reason why the hybrid download technique that this thesis proposes is vulnerable to the fake blocks attacks and why a pollution prevention algorithm is needed.

4.2 Pollution Prevention algorithm

The fake block attack introduces a serious issue in an environment where no piece can be checked against its hash. How can the pollution of the content be prevented in such a scenario? In this section we present an algorithm to guarantee security when the piece hashes are not known because of the lack of a torrent file, and the only trustworthy authority is the web server holding the original content.

The hash checking will be performed through Merkle hashes. When downloading with the Merkle hashes technique, the hash of piece X, the hash of the sibling of X and all the uncles of X up to the root hash get received with the X piece in the same message (see figure 2.1 in chapter 2). Since, in this situation, there is no trusted root hash to be compared with the ones received from the peers, an algorithm that ensures a level of security needs to be devised.

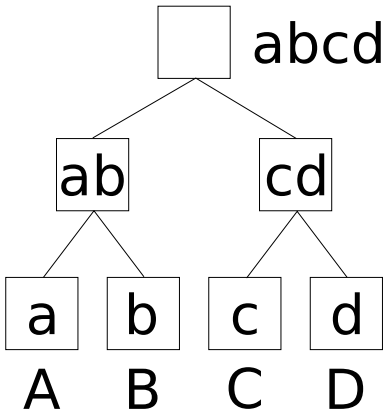


Figure 4.1: *Original Merkle tree. The capital letters denote the piece while the lower case letters denote the hashes. Note that in this case the string 'ab' does not represent the concatenation of 'a' and 'b' but the hash of their concatenation. Therefore 'a' = hash('A') and 'ab' = hash("hash('A')hash('B')"). These assumptions are also valid in the next figures.*

Here is presented how the algorithm works; figure 4.1 will be referred as the correct Merkle tree to take into consideration in the following examples. When a *piece* message is received from a peer, a preliminary check is performed to guarantee the coherence between piece and hashes. Figure 4.2 shows an incoherence between a piece and its hash; in this case the piece is discarded. When receiving the same root hash from different peers (figure 4.4), we are sure that all the rest of

the tree's hashes are equal, assuming that the preliminary hash check is performed. This does **not** mean that the received content is correct but that the content received from different peers is the same. In other words, if the peers are sharing fake content, they are all sharing the same fake pieces. This comes from a property of the Merkle hashes. One root hash identifies one unique Merkle tree. Two different root hashes have at least one leaf hash differing. Therefore, if two peers are sharing some content that is differing by just one byte, they will be sharing two different root hashes (figure 4.5).

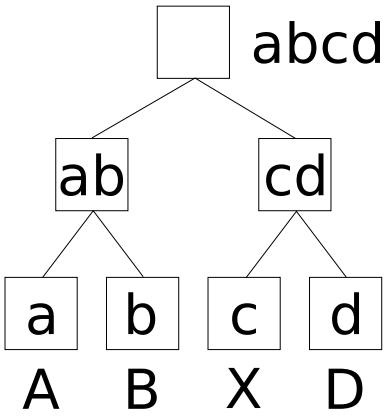


Figure 4.2: *Merkle Tree incoherence. In this case the correct piece should be 'C' and not 'X'. When receiving the **piece** message containing 'X' from a peer, a preliminary check between the piece and its hashes will be performed, revealing the incoherence of the message.*

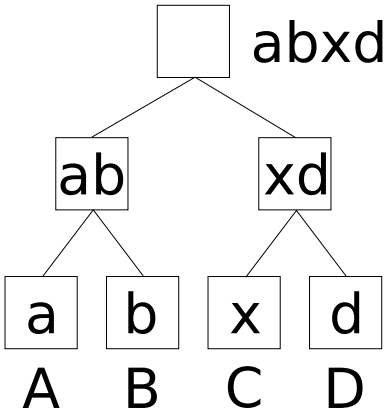


Figure 4.3: *Merkle Tree incoherence. The hash of piece 'C' should be 'c' and not 'x'. When receiving piece 'D' from a peer, the sibling hash of 'D' ('x') is received along with the **piece** message. If piece 'C' is retrieved from the web server, the hash checking will reveal that the merkle tree shared by the peer from which we received piece 'D' is incoherent.*

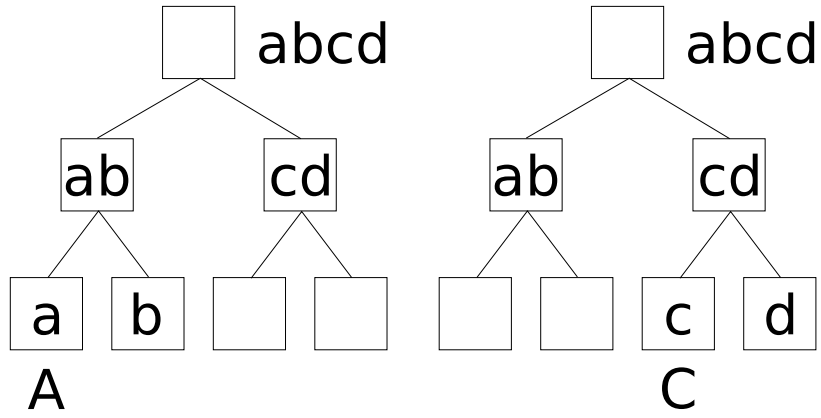


Figure 4.4: *Known part of the Merkle tree of two peers sharing the same content. This figure shows the part of the Merkle tree received along with two different pieces: piece 'A' from a peer (left) and piece 'C' from another peer (right). Since the two peers are sharing the same root hash, we can assume that they are sharing the same content, unless one of the two peers sends an incoherent message like in figure 4.2.*

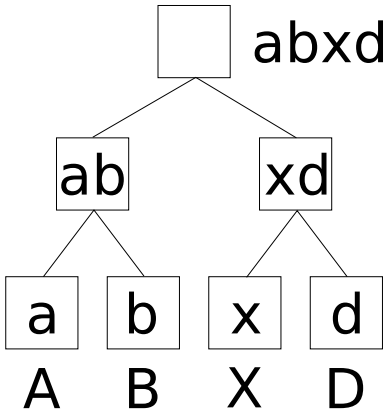


Figure 4.5: *Peer sharing different content. Comparing this Merkle tree with the one in figure 4.1 reveals that the two root hashes are different, therefore the content associated to those Merkle trees is different.*

The only trustworthy entity in this scenario is the web server. Comparing the content received from the web server against the same content received from a peer reveals whether the peer is lying or not about a specific piece (but not necessarily about the rest of the pieces). It is important to note that it is infeasable to compare all the pieces received from a peer against the web server to ensure integrity.

Instead of requesting from the web server the same piece retrieved from a peer, the *sibling piece*[1] will be requested. There are multiple advantages with this ap-

[1]Two sibling pieces are two pieces whose leaf hashes have the same parent. In figure 4.1 *A* and *B* are two

proach. First we avoid downloading duplicate content, once from the web server and once from peers, hence, converging to a faster download completion. Second, we immediately have a hash to perform a coherence check, instead of randomly download pieces from the web server and check if a hash is available or not; this reveals content incoherence (see figure 4.3) between server pieces and peers hashes as soon as possible.

Uppon finding a mismatch between a piece retrieved from HTTP and the same piece from a peer, all connections with peers sharing the same root hash will be dropped. Since these peers are sharing the same root hash, they are also sharing the same fake content, and therefore, they are all attackers.

It has been presented how to detect attacks in case the *piece* message is not coherent (figure 4.2) and when a (sibling) hash received from a peer does not match the hash computed from the same piece retrieved from the web server (figure 4.3). We also presented how to identify peers sharing the same content(figure 4.4). However other scenarios are possible: the case in which a client is connected only to peers sharing the same fake root hash (and no fake piece is found by the comparison between hashes received from peers and pieces received from the web server) and the case in which the client is receiving content from peers sharing different root hashes (that means that there is some attackers). It will be demonstrated that, in the first case, pollution can be prevented by the *invalidate message* and that, in the second case, the attackers can be prevented by applying the *rotten branch detection* technique. These two techniques will be explained in the next sections.

Before proceeding it is necessary to introduce a few concepts:

- **good peer**: the good peers are those sharing the same content of the web server and sending coherent messages (messages that pass the preliminary check). It is never possible to know if a peer is good, unless all the content is received from that peer and checked against the web server's content; however this case conflicts with one of the main goals of this thesis: server load reduction.

- **evil peer**: the evil peers are peers injecting polluted content in the swarm. They can be detected by a unsuccessful preliminary check (figure 4.2); by a sibling hash that does not match the hash computed from the same piece retrieved from the web server (figure 4.3); or when, in the case of the *rotten branch detection*, a piece received from a peer is different from the same piece retrieved from the web server.

- **candidate peer**: the candidate peers are peers connected to the client in a *candidate* state. It is not possible to know whether they are *good* peers, and they have not been detected as *evil* peers yet. Since the case in which a client

sibling pieces

is connected only to *evil* peers is assumed to be rare, the *candidate* peers are most likely *good* peers.

- **good root hash**: the root of a Merkle tree computed from content equal to the web server's content (non polluted content).
- **evil root hash**: the root of a Merkle tree computed from polluted content.
- **candidate root hash**: root hash shared by *candidate* peers.
- **conflict state**: we are in a conflict state when the client is connected to peers sharing different root hashes.
- **candidate state**: we are in a candidate state when we are connected only to peers sharing the same root hash. We cannot know if that root hash is *good* or *evil*. Most likely in a *candidate* state we are receiving data from *good* peers.

It is not possible to know whether the P2P client is connected only to *evil* peers. However this scenario is exceedingly unlikely for large swarms. The most common scenario is the one in which the client is connected to a group of *evil* peers and to a group of *good* peers. This *conflict* state will generate two or more sets of peers associated with different *conflicting* root hashes, in contrast with a *candidate* state where there is only one set of peers sharing the same *candidate* root hash.

While in a *candidate* state we cannot know if the data we are receiving is correct or not, in the *conflict* state we are sure that there is at least a set of *evil* peers. It is important to find the *evil* root hash as soon as possible, avoiding to download polluted data. The bigger the number of pieces differing from the original content in an *evil* root hash, the easier it is to find wrong data since there is a higher probability to download a fake piece; this means that the worst case we can have is when an *evil* peer is sharing some content that differs from the web server's content by just one piece. To help find the differing piece in a *conflict* state, we propose a technique called the *rotten branch detection*. This technique consist in comparing the branches of the Merkle trees of conflicting root hashes.

The rotten branch detection enable us to find the fake piece in at most $\lceil \log_2 n \rceil$ steps with n equal to the number of pieces in the content file, where each step implies the download of the same piece twice, once from the conflicting root hashes, and once from the web server in the last step. Therefore, the amount of redundant data to download, where the redundant data is the data that has to be downloaded twice, once from a peer and once from the web server, is minimal and is proportional to the number of conflicts detected during the download session.

As mentioned before, there is only one case where a fake piece attack is not detected: the case where the P2P client does not connect to any *good* peer but only to *evil* peers all sharing the same root hash. However, this scenario is very unlikely to

happen. To address this eventuality, the *invalidate message* technique has been introduced. The *invalidate message* is a technique to spread *evil* root hashes detections over the swarm. This technique, explained in the following sections, introduces a new concept: distributed security.

Security is therefore distributed among the peers participating in the swarm; each peer performs integrity checks on portions of the downloaded file. When a peer detects an *evil* root hash, it shares its discovery with its neighbors, increasing the **global** security of the swarm.

In the following sections, the *rotten branch detection* technique and the *invalidate message* will be explained in detail. The chapter continues with some considerations about sharing policies and concludes with a few considerations.

4.2.1 Rotten branch detection

The rotten branch detection is the proposed technique to find the *evil* root hash as soon as a conflict happens. A conflict is defined as the state where the P2P client is connected to two or more different sets of peers sharing different root hashes.

As already mentioned, building the Merkle trees out of two files differing by just one byte, ends in two different root hashes. Therefore, if the client is connected to two sets of peers sharing two different root hashes, we can deduce that we are about to download at least one piece different from the web server content and therefore unwanted. The amount of polluted data can vary from just one piece to the whole content. It is important then to find where the fake piece comes from and which root hash is to be invalidated.

In case of conflict, the ideal way to find the fake piece would be to compare all the leaf hashes (with same index) related to the conflicting Merkle trees sequentially, from first to last, until the mismatch is found. When two hashes mismatch, the piece relative to that hash is fake in one of the two hash trees. Figure 4.6 illustrates how to detect a conflict in case all the hashes are known.

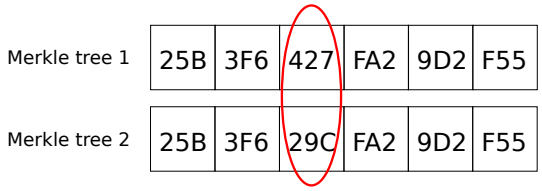


Figure 4.6: *Conflict detection. Each strings represent a single leaf hash. In this case all the piece hashes are known and the detection of mismatching pieces is simply performed by comparing all the hashes until the mismatch is found.*

However, as explained in the Merkle Hashes section, not all the hashes in the tree are known at the beginning but only the parts received along with the piece messages. For this reason it is not possible to compare each hash of a hash tree with

the same hash of the conflicting hash tree, as the most of the leaf hashes may not have received yet.

The hash tree is built during the process of collecting pieces coming from peers using the same root hash. Along with each piece, the piece hash, the sibling hash and the uncle hashes are received. By concatenating the piece hash with the sibling hash and applying the hash function, we compute the parent hash. Repeating the process with the computed parent hash and the uncle hash, we compute the grand parent hash and so on up to the root hash. Therefore, for each downloaded piece we have all the parent hashes up to the root; this creates a logical path from root to leaf. We define the *known paths* as the paths that connect the known leaf hashes to the root hash, as shown in figure 4.7.

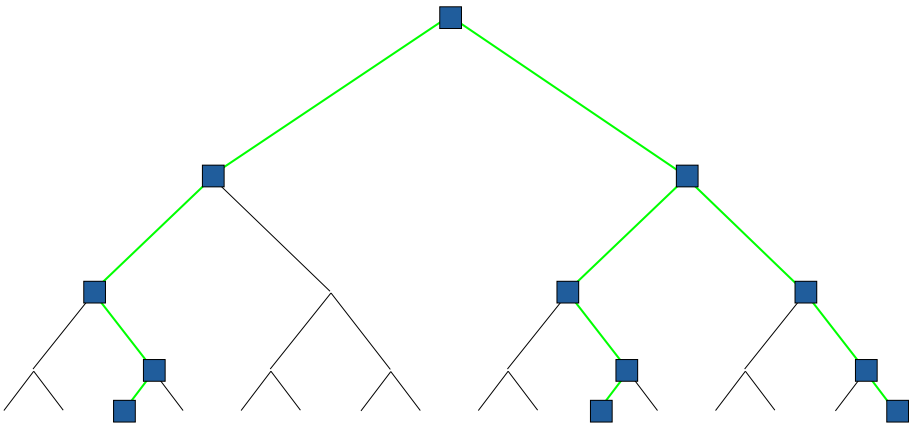


Figure 4.7: *Known branches connecting the received leaf hashes with the root hash.*

The rotten path is the path that starts from one fake piece and propagates up to the root hash. The term *propagate* has been used here for a reason: when concatenating the fake piece hash with the sibling hash and computing the parent hash node, the *pollution* is transmitted from the fake piece hash to its parent and all the parents up to the root, *contaminating* all the branches in the path as shown in figure 4.8. Therefore, by finding the rotten path we find the fake piece. To find the rotten path, we need to detect (at least) one rotten branch at the time. Hence, the name *rotten branch detection*.

The previous definitions are needed to understand how the rotten branch detection technique works. This technique consist of comparing the hashes in the same path of two conflicting Merkle trees, from root to leaves. The path where all the hash nodes mismatch in both Merkle trees is a *rotten path*, and the leaf of that path is the hash of a fake piece. Once the rotten path has been detected we need to identify which of the two Merkle trees contains the fake piece. Downloading the same piece from the web server and comparing it to the pieces received from the peers will reveal which is the fake piece and therefore which of the two root hashes is the

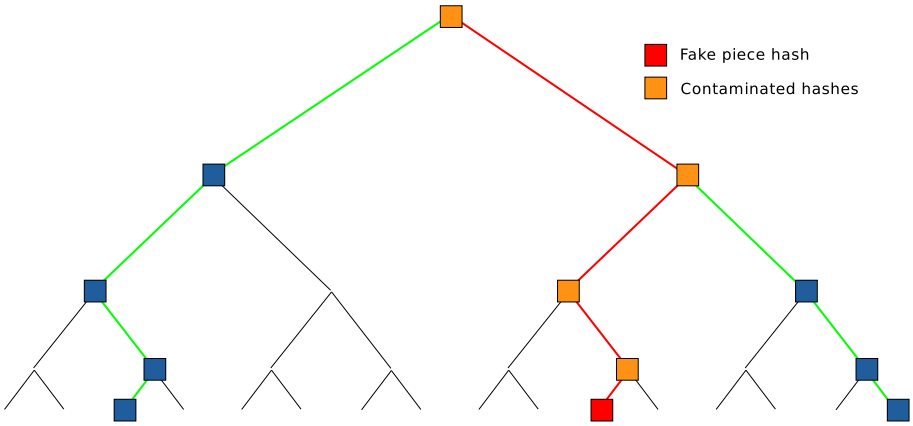


Figure 4.8: *Propagation of the pollution from the fake piece hash up to the root. The red block represents the fake block hash. All the parents of that hash will be contaminated and therefore different from the original Merkle tree. Hence the pollution propagates from the leaf to the root hash.*

evil one.

Not all the paths are known at the beginning. This means that, to find the rotten path, new pieces could be asked to the conflicting peers. Along with the new pieces, also the hashes needed to build a path for the comparison between two conflicting Merkle Trees will be received. It is unproductive to select random paths to compare; this could end up in downloading a big amount of data from the peers, while we want to find the fake piece as soon as possible. The best way to select a path is to analyze the result of the previous comparison. As shown in figure 4.8, the rightmost path has one branch in common with the one on its left. That branch is *rotten*, while the rest of the path is not. Therefore, we could start from that branch to select the next path to compare. This means that in the next comparison we will be one step closer to find the fake piece.

The example shown in figure 4.9 will help in understanding the procedure of each step of the rotten branch detection technique. For simplicity, just one Merkle tree is shown. This tree is the result of overlapping the two conflicting trees. The nodes in red are differing in hashes, the nodes in green are equal in both trees, the red branches are rotten branches connecting two red nodes and the green branches connect two nodes where the lower one is a green node.

- **Step 1**: In the first step we just need to select a path from both Merkle trees to be compared. The best path selection is always a *known path*. In this way we do not need to download any piece from at least one peer (from both if the path is a known path in both Merkle trees); this assumption is valid for all the steps. If a known path does not exist, a random path can be selected. Suppose that, as shown in figure 4.9, in the first step we select the path that connects

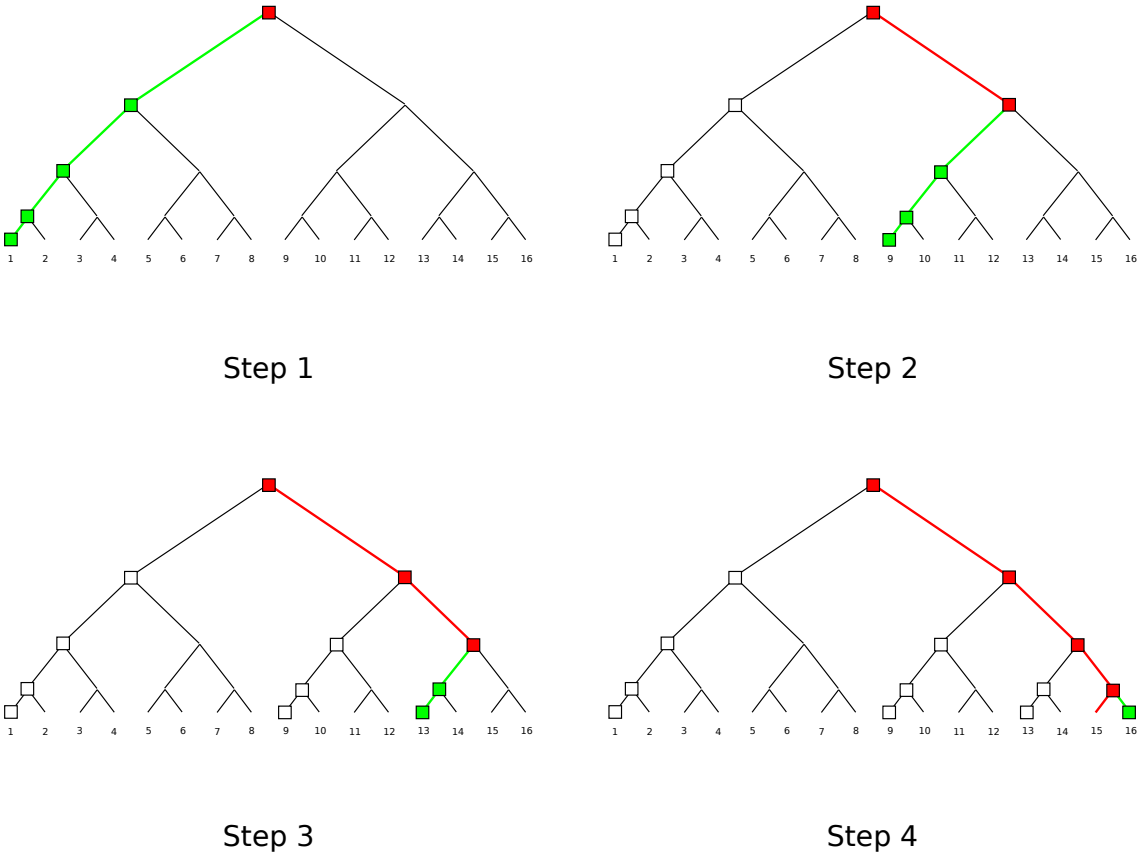


Figure 4.9: *Rotten branch detection example. At each step, one rotten branch is detected by comparing the hash nodes in the same path of two conflicting Merkle trees. When all the hash nodes in a path mismatch, we find a fake piece, pointed by the hash leaf of the rotten path. This example show the worst case scenario: only one mismatching piece in the two conflicting Merkle trees and just one branch detected at each step; if the first path to be compared was path 15, like in step 4, the rotten branch detection would have taken only one step to find the fake piece.*

the root hash to piece 1 (path 1 for simplicity). In this case all the hash nodes in the path are equal. Of course the root hash of the two conflicting Merkle trees is different. Since the left child node of the root hash is equal in both trees, then all the paths descending from this node will be equal in both trees.[2] Indeed, there is no need to compare all the hashes in a path; we can start from the root hash and go down until we do not find the first non mismatching hash node. From that point on, all the hashes will be equal in both trees. Therefore, in this example, all the pieces between piece 1 and piece 8 will be equal in both trees. This means that there is at least one fake piece in the range 9-16.

[2]Recall that one root hash identifies one unique hash tree. This is also valid for subtrees: if the root hashes of two subtrees are equal, then the rest of the hashes in that subtree are equal.

- **Step 2**: In the previous step we found that the pieces in the range 1-8 are equal in the two conflicting Merkle trees. Therefore there is a fake piece in the range 9-16. This means that the right child node of the root hash is a *rotten branch*. At this point we select a random path going from the root hash to a leaf hash in the range 9-16. If a known path is available in that range we select it, otherwise we need to download the same piece from peers sharing the conflicting root hashes. Lets suppose we chose path 9. As shown in figure 4.9, in this path the first branch (from top to bottom) connected to the root hash is a rotten branch, while the rest of the branches are matching. Once again, this means that in the range 9-12 the pieces are equal in both trees, while there is a fake piece in the range 13-16.

- **Step 3**: As it is possible to notice, in each of the previous steps we found one rotten branch, reducing the range in which we can find the fake piece. In this step we select a path starting with the rotten branches just detected. Lets suppose we select path 13. The last two hash nodes in green are equal in both trees. Therefore there is a fake piece in the range 15-16.

- **Step 4**: Path 16 is selected out of the range 15-16. Since the hash leaf 16 is equal in both trees, we can deduct that the mismatching hash is in leaf 15. Therefore the *rotten path* is path 15 and the piece 15 is fake in (at least) one of the two conflicting Merkle trees. At this point we need to request piece 15 from the web server. Comparing piece 15 retrieved from the server with piece 15 retrieved from the peers will reveal which is the *evil* root hash.

In this example, the worst case scenario is taken into consideration; only one piece is differing in the two conflicting Merkle trees and only one rotten branch is detected at each step. If in the first step, path 15 was selected, then the rotten path was found at the first iteration. Since the algorithm detects at least one branch at each step, and the path length is equal to the height of the tree, at most $\lceil \log_2 n \rceil$ steps are needed, where n is the number of pieces in the file. In this case, $\log_2 16 = 4$ steps were needed.

To simplify the understanding of this technique, we assumed to download the same piece from two conflicting peers. In reality, we will download two sibling pieces[3]. The reason for this is that, by downloading two sibling pieces, the same hashes are received along with the *piece* message, but we avoid redundancy and increase the download speed.

Once the rotten branch detection is finished, we pass from the *conflict* state to the *candidate* state. The pieces downloaded from the *evil* peers and matching with the *candidate* Merkle tree, can be kept; in such a way we take advantage from *evil* peers by downloading *candidate* content. The connection with the peers sharing

[3]Two sibling pieces are two pieces that have two sibling leaf hashes: two leaf hashes with the same parent.

the *evil* root hash will be dropped, leaving space for new connections to come in, and the *evil* root hash will be inserted in a *blacklist*. When a new connection is established, the root hash shared by the peer will be checked. If the root hash is already in the *blacklist*, the connection will be dropped.

When new connections come in, we increase the chance of having a conflict. Conflicts are important for two reasons: (a) we can discover that the current *candidate* root hash is indeed an *evil* root hash and (b) with each conflict we increase the **global security** by sending an *invalidate message*.

4.2.2 Invalidate message

The invalidate message is a message sent from one peer detecting an *evil* root hash to the swarm. The message consists of the SHA1 hash of the URL source (swarm identifier), the *evil* root hash and the piece index referring to the polluted content, and can be used to warn other peers about *evil* root hashes in the swarm. This message speeds up the detection of *evil* peers and polluted content and is intended to help the swarm in converging toward the *good* peers.

By its nature, the *invalidate message* protocol is a gossip protocol and operates within an overlay network. The peculiarity of this gossip protocol is that the messages between two peers can be exchanged even if the two peers are directly connected. This is useful in the scenario where one peer is downloading content only from *evil* peers, since it is highly unlikely it can receive an invalidate message from its attackers. It can instead receive an invalidate message from a peer in the overlay network. The details of the overlay network where the *invalidate message* operates are not going to be discussed here. We can just assume that the overlay network for a specific swarm is composed of the peers registered in the DHT as participants of that swarm.

Once the invalidate message has been received by a peer, it will be noted down and its information will be used when the peer establishes a connection with an *evil* peer sharing the invalidated root hash. When this happens, the client peer can bypass the rotten branch detection algorithm step and ask directly for the presumed fake piece. The presumed fake piece will be then compared to the same piece retrieved from the web server. If the piece is fake, the client peer can resend the invalidate message to the swarm.

The word *presumed* is used here to qualify the fake piece since the invalidate message could have been sent by an *evil* peer as an attack. This is also the reason why the invalidate message has to be verified before it is redistributed in the swarm. In such a way the *evil* invalidate message is blocked at the first hop instead of being spread to all the peers participating to the same swarm, generating useless traffic.

The invalidate message is very useful when received in a particular condition: when the receiver is connected only to peers sharing the same *evil* root hash. Previ-

ously in section 4.2 it has been explained that the only case in which the algorithm is vulnerable to attacks is exactly in this situation. There is no way to detect the *evil* root hash if no conflict happens and if the pieces coming from the web server do not reveal the presence of a fake piece. In this case, the invalidate message helps in detecting the polluted content when there is no other way to do so.

To conclude, the invalidate message helps the swarm in converging toward a *good* swarm in a cooperative way; each peer verifies a portion of the downloaded content and shares its *evil* root hash detection with other peers, improving the global security of the swarm. In HTTP2P, we can define this concept as **distributed security**.

4.2.3 Sharing policies

The pollution prevention algorithm is based on Merkle hashes. The Merkle tree can be built in two ways: (a) by downloading all the content from the web server and computing the hash tree when the download is finished, or (b) by receiving the hash tree portions along with each *piece* message. A new connection can be established only if the other party has a part of the Merkle tree (and therefore a root hash) to share. Without a Merkle tree, peers cannot send the *piece* message, since the content integrity cannot be verified.

In the first phase of the swarm creation, no peer will have any part of the Merkle tree. Hence, all the content will be downloaded from the web server. When the first download completes, the first Merkle tree can be computed and the peer holding the content can start establishing connections. The sending peer will distribute parts of the Merkle tree, the receiving peers will have the first portion of the hash tree and can start establishing new connection in turn; the hash tree starts spreading over the swarm as well as pieces. In this second phase, the load of the server is distributed over the P2P network.

Another scenario is also possible: the first Merkle tree has not been computed yet and *evil* peers distribute portions of an *evil* hash tree. In this case, the *distributed security* effect will help in detecting the *evil* root hash; *invalidate messages* will be spread over the swarm, invalidating the *evil* root hash. Once the attackers have been detected, the *evil* hash tree will be discarded. Since no other root hash is present in the swarm yet, the peers will continue the download retrieving the content only from the web server until the first Merkle tree is generated.

As we just examined, the first sharing policy is to share content only if (at least) a part of the Merkle tree has been received. This ensures local content verification and global root hash invalidation. Another simple and effective sharing policy is to redistribute to the network only pieces received form the web server. This policy guarantees to inject in the swarm only non-polluted content. On the other hand, when sharing the content received from the P2P network instead, the risk of

injecting polluted content is higher.

Good sharing policies can increase the global security over the swarm. In this chapter only two policies have been presented. However the pollution prevention algorithm is flexible and open to integrate other sharing policies that can speed up attacks detection.

4.3 Considerations and conclusions

We presented how, in an environment lacking in piece hashes, the content integrity cannot be guaranteed unless a mechanism to resist to fake block attacks is devised. This mechanism has been synthesized in the *Pollution prevention* algorithm. This algorithm makes the use of the Merkle hashes technique to trade hashes between the peers, and is supported by the *Rotten branch detection* and *Invalidate message* techniques to respectively identify *evil* peers and to distribute attack detections over the swarm.

With the *Invalidate message* we also introduced the new concept of *distributed security*, where each peer participating to the swarm is responsible for checking the integrity of a portion of the download and for sharing fake root hash detections among other peers in the swarm.

Fake block attacks are usually coordinated attacks intended to create as much damage as possible. Therefore, these attacks usually happen in swarms sharing popular content, where the number of peers is substantial. From an attacker point of view, it is not profitable to attack swarms of a few dozen peers; it means that the shared content is not popular and that the impact of the damage to the swarm is small.

The whole concept of this thesis is to resist to the phenomenon of *Flash Crowds*. Flash Crowds happen for popular content, where thousands clients try to download the same content. This is where we can profit the most from the P2P support. This is also the situation where the risk of attacks is higher.

The effectiveness of the pollution prevention algorithm is proportional to the number of peers in the swarm; the more the peers, the better the algorithm performance. At the same time, the bigger the swarm, the higher the risk of attacks. Therefore, the pollution prevention algorithm is effective when it is most required. If the number of peers in a swarm is small, the web server can handle the load of distributing all the content and the P2P support is not needed anymore. In such a way the content integrity can be guaranteed when the pollution prevention is weak, by downloading the content only from the web server.

Chapter 5

Implementations and Experiments

This chapter describes the experiments performed with the two main implementations of the HTTP/BitTorrent hybrid. The first one concerns with the web seeding features included in the Tribler Core during the thesis, the second implementation is a simulator of the *pollution prevention* algorithm aiming to test the server-side performance of the BitTorrent/P2P architecture hybrid. The reason for two different implementations is that the web seeding feature has been included in the Tribler Core in the early stages of the thesis, while the Pollution Prevention algorithm has been created in the second phase, and not included in the Tribler Core. However, the Pollution Prevention algorithm should not influence much the download speed/ratio behaviour of the Tribler Core, and the server load variation can be tested with the Pollution Prevention simulator. The results of the experiments performed with the two different implementations show that the performance achieved is the result of merging the best of HTTP and BitTorrent; in particular the low download start-up delay is typical of HTTP while the download load is distributed over several hosts including the web server, a typical characteristic typical of BitTorrent environments. In the following experiments, the *server load* is defined as the amount of content retrieved from the server over the total amount of content retrieved by the peers.

5.1 Tribler Core

The implementation of a browser plug-in rendering video content retrieved from a P2P engine, was developed in the early stages of this thesis [26][8]. The goal was to provide a web page with video content retrieved from the BitTorrent network. It's easy to imagine the advantages of such a tool in the attempt of building a YouTube-like service. The result has been the SwarmPlugin, released in 2009 [12], composed of a modified version of the *VLC browser plug-in* responsible for rendering the video in the page, and by the *Background Process*, based on an existing feature of the Tribler Core that provides video on demand content retrieved from the BitTorrent network.

The project then evolved in 2010 [16][13] in a partnership with Wikipedia in the attempt of including video content in the Wikipedia pages, in such a way that their low-cost infrastructure is able to satisfy a large amount of user requests. The HTTP seeding feature, already included in the Tribler Core in the early stages of this thesis, has been adapted to cope with the project requirements. With the introduction of a policy to balance the bandwidth coming from HTTP and BitTorrent, the developed engine has been able to provide P2P video service able to integrate from HTTP the bandwidth necessary to keep the playback sustainable.

The following sections show the results of the experiments performed with the Background process including the HTTP seeding features. These tests use different torrent in which the *url-list* parameter has been embedded. The *url-list* parameter specifies the URL of the resource shared in the swarm, and enables the HTTP download in the Tribler Core.

5.1.1 Background process interface

The Background process interface is composed by two channels: a TCP socket for receiving requests and sending the status to the connecter[1], and an HTTP channel providing the video content.

```
Escape character is '^]'.
START http://url2torrent.net/torrent/f75686cf82a7bd71e8fdd6d0817898b20cdc2eff.torrent
INFO Prebuffering 0% done (connected to 0 people). Playing in less than a minute.
INFO Prebuffering 0% done (connected to 0 people). Playing in less than a minute.
INFO Prebuffering 32% done (connected to 1 person). Playing in less than 5 minutes.
PLAY http://127.0.0.1:6877/content/0f49090a7a43e3038407ee296098c2602cd866a8/0.733461915126
INFO
```

Figure 5.1: *Telnet session connecting to the BG process and starting a download*

Figure 5.1 shows how the connecter, in this case a telnet session, interacts with the Background process by sending the URL of the torrent to download. Figure 5.2 shows the Backgound process' output where the *START* command is received by the connector, and the activation of the HTTP support, that is the HTTP seeding feature, is triggered by the url-list parameter contained in the torrent metadata. Upon the activation of the HTTP seeding feature, the BG starts receiving content from the web server. When enough content to keep the video playback sustainable has been downloaded, the BG sends back to the connecter the URL of the internal VideoServer from which the video content can be retrieved.

```
bg: Test if already running
Build 17598
bg: Kill-on-idle test enabled
bg: Awaiting commands
bg: Plugin connection_made
bg: Got command: START http://url2torrent.net/torrent/f75686cf82a7bd71e8fdd6d0817898b20cdc2eff.torrent
bg: get_torrent_start_download: Starting new Download
bg: VOD EVENTS ['start', 'pause', 'resume']
main: Starting new Download '\x0fI\t\nzC\xe3\x03\x84\x07\xee)`\x98\xc2`,\xd8f\xa8'
http: Activating HTTP support
http: Content URL: http://upload.wikimedia.org/wikipedia/commons/8/84/Play_fight_of_polar_bears_edit_1.avi.OGG
http: Got 32768 bytes from server
http: Got 32768 bytes from server
http: Got 32768 bytes from server
bg: Telling plugin to start playback of /content/0f49090a7a43e3038407ee296098c2602cd866a8/0.733461915126
Play_fight_of_polar_bears_edit_1.avi.OGG DLSTATUS_DOWNLOADING 27.34% None up     0.00KB/s down   350.47KB/s
Play_fight_of_polar_bears_edit_1.avi.OGG DLSTATUS_DOWNLOADING 27.34% None up     0.00KB/s down   166.90KB/s
videoserv: do_GET: Got request /content/0f49090a7a43e3038407ee296098c2602cd866a8/0.733461915126 bytes=0-
videoserv: do_GET: MIME type is audio/ogg length 5992448 blocksize 32768 Thread-12
http: Got 32768 bytes from server
Play_fight_of_polar_bears_edit_1.avi.OGG DLSTATUS_DOWNLOADING 30.62% None up     0.00KB/s down   122.58KB/s
http: Got 32768 bytes from server
http: Got 32768 bytes from server
Play_fight_of_polar_bears_edit_1.avi.OGG DLSTATUS_DOWNLOADING 39.92% None up     0.00KB/s down   118.93KB/s
```

Figure 5.2: *BG process output. The highlighted lines show the START command received by the connecter, the activation of the HTTP seeding feature, and the availability of the downloaded content on the specified URL (start playback line)*

5.1.2 Bandwidth usage

Figure 5.3 shows the system behaviour of a BitTorrent download with the HTTP seeding feature activated. The file retrieved has a total size of 237 MB, and the swarm, at the moment of the download, was composed of 15 seeders and 240 leechers. After the swarm discovery step, the client could successfully initiate connections with 25 peers. One can notice that the HTTP incoming bandwidth starts immediately and remains constant, like in a regular HTTP download. The BitTorrent bandwidth starts later, increasing the total speed of the download. In this case the BitTorrent download start-up latency is low (around 4-5 seconds), since the swarm selected for the test was properly populated. In the following section we will see that for not well populated swarms, the download start-up latency of the BitTorrent network is higher.

After the Tribler engine starts the HTTP download, new connections are also started with peers found during the swarm discovery step. The first peers start giving bandwidth after 5 seconds. After around 40 seconds from the start of the download, the download speed increases by a factor of two with a peer giving around 1 MB/s of bandwidth. This also means that the server load is reduced by around 2/3 during the download session, since just one third of the data is downloaded from the web server.

In this test, we are not interested in any sort of bandwidth balancing, unlike in the next experiments. The system just tries to get the maximum bandwidth from

[1]Even if the BG has been designed to interact mainly with the VLC plug-in, any software component able to ”speak” the BG protocol can activate a P2P download. this is why we don’t refer to the browser plug-in but to a generic connecter.

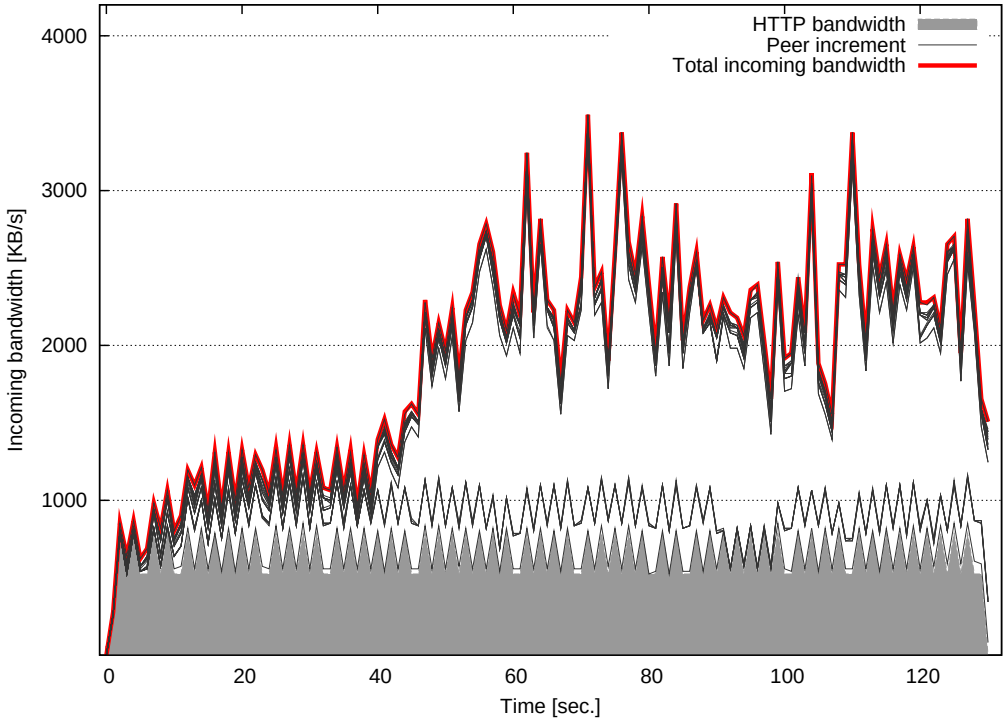


Figure 5.3: *Hybrid download behaviour. The grey filled curve represents the HTTP bandwidth. On top of the HTTP bandwidth each of the 25 peers increases the total incoming bandwidth with its bandwidth contribution. The total download speed is represented by the red line.*

HTTP and from BitTorrent. This experiment is meant to show the best of both worlds: low start-up latency, improved download speed, server load reduction.

5.1.3 Start-up time delay

This experiment measures the start-up time delay for HTTP and BitTorrent. In particular we are interested in the latency for completely receiving the first piece. The reason for selecting this measure is that it both combines the latency for receiving the first bit (start-up latency) and the speed at which the first bytes are received (initial download bandwidth). The start-up latency itself is not much indicative if we consider the case where the first bit is received with low latency but the download bandwidth is very slow.

Figure 5.4 shows the start-up time delay measurements of 10 executions of the experiment. The torrent in question is the popular *Elephants dream* short movie, with a piece size of 512 KB. Combining these results with the ones in figure 5.3 we can understand not only that BitTorrent has a higher start-up latency compared to HTTP, but also a lower initial download bandwidth. These results are particularly

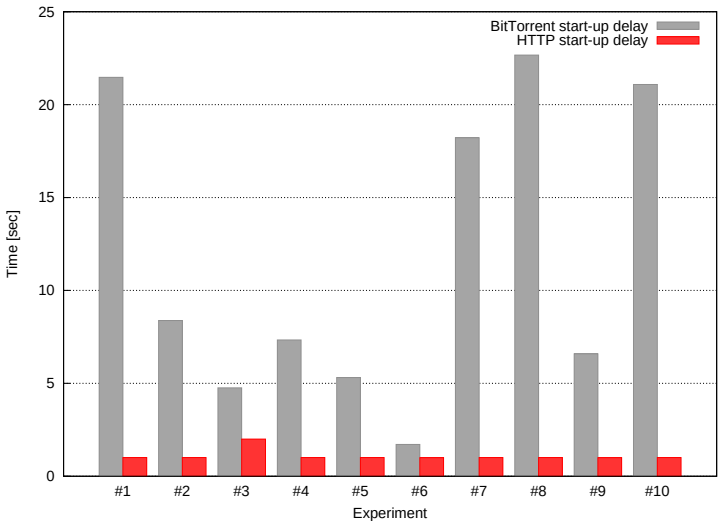


Figure 5.4: *Delay comparison between HTTP and BitTorrent for receiving the first 512 KB of data. The resulting values combine the latency for receiving the first bit and the initial download bandwidth.*

meaningful in the context of P2P based video on demand, where the playback start-up delay is critical.

We can observe that the HTTP values vary between 1 and 2 seconds. In the BitTorrent values we can observe a bigger variation, between 1.7 and 21.4 seconds. This big variation is caused by some factors like the tracker response time, the peers outgoing bandwidth, the peers unchoking delay and swarm characteristics like the swarm size. We can conclude that, in the context of P2P based video on demand, the support of HTTP can greatly improve the start-up delay of the playback.

5.1.4 Flashcrowd simulation

This experiment analyses the global behaviour of the swarm in two different configurations: an HTTP download with the BitTorrent support, and the same download without the BitTorrent support. The web server upload bandwidth has to be split between 10 clients, that start their download with a small delay one from each other. The maximum upload bandwidth of the web server is 250 KB/s, while the download and upload bandwidth of each peer has no limit.

As it is possible to see in figure 5.5, in the first few seconds where peer #1 started the download, its download bandwidth reaches the maximum web server's outgoing bandwidth limit. In a second step, peer #2 starts its download, and the web server's bandwidth has to be split between two clients. When all the 10 peers have started their download, the bandwidth of the web server is equally split between them. When the download of the first peers completes, the download bandwidth of

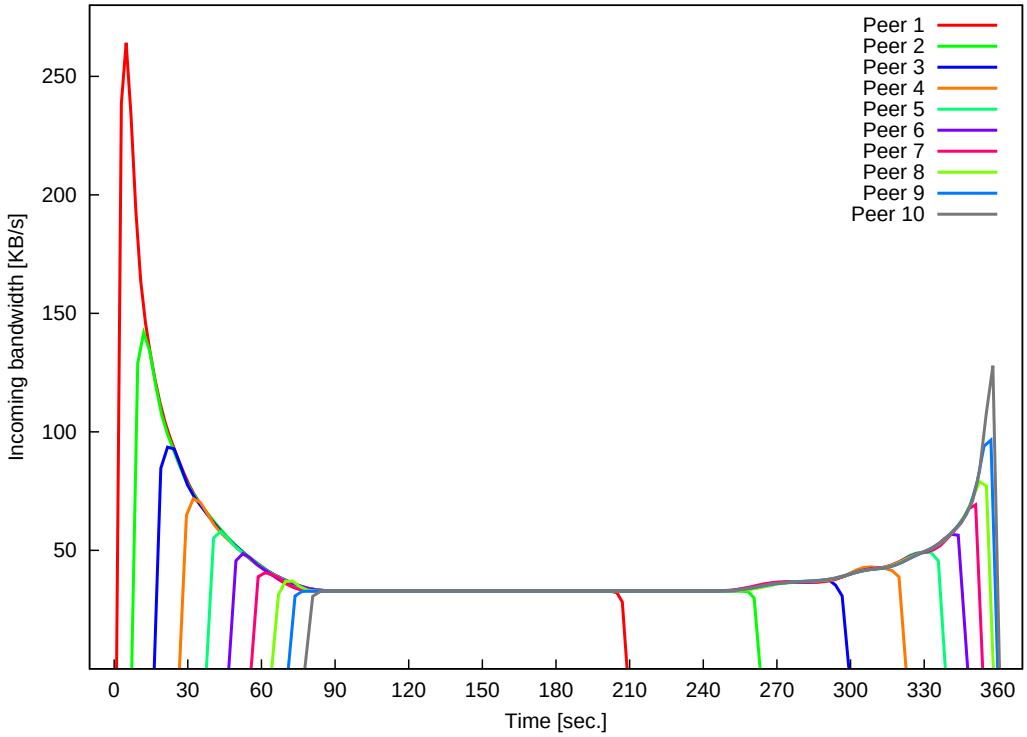


Figure 5.5: *HTTP download. The web server output bandwidth is equally split between all the clients.*

the last peers increases, since the server's bandwidth has to be split between fewer peers.

Figure 5.6 shows the behaviour of the same experiment but with the P2P download support enabled. we can notice the differences on the two graphs of the Y- and X-axis scales, denoting different incoming bandwidths and completion times. For each peer, the HTTP incoming bandwidth and the total incoming bandwidth (HTTP + BitTorrent) is displayed. As in the previous case, the HTTP bandwidth of the web server is split between the peers and the HTTP incoming bandwidth of each peer has the same behaviour of the previous experiment. However, the total incoming bandwidth of each peer behaves differently.

In the first phase, each peer gets the expected HTTP bandwidth plus all the content retrieved by the swarm until that moment. As an example, peer #2 gets the half of the HTTP outgoing server bandwidth (since it has to be split with peer #1), plus all the content retrieved by peer #1 until peer #2 started the download. As the next peers start the download, the initial download curve reaches higher values, because more content has been downloaded by all the previous peers. This explains the curve peaks during the initial phase.

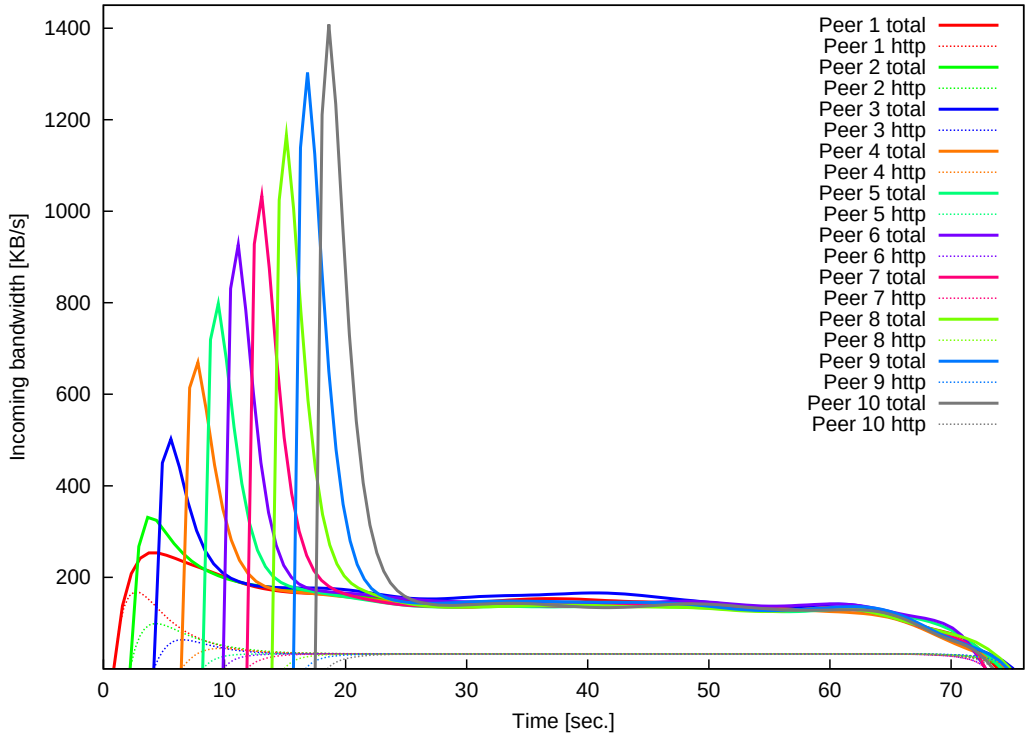


Figure 5.6: *Hybrid HTTP/BitTorrent download. The web server output bandwidth is equally split between all the peers and each peer receives from the others their portion of HTTP downloaded content. Therefore the total bandwidth received by each peer is equal to the web server's output bandwidth. In general, in an hybrid download, independently from the swarm size, each peers receives a bandwidth equal to the maximum web server's output bandwidth.*

After the first initial phase, the download bandwidth of all the peers converge to a value that is slightly smaller than the web server output bandwidth. In other words, the HTTP download bandwidth of each peer is shared between each other and therefore, the total download bandwidth of each peer is equal (or less) to the web server output bandwidth. As an example, if the total output bandwidth of the web server is 10 pieces per second, and 10 peers download simultaneously from the server, they get 1 piece per second; at the same time, the peers will be sharing the piece retrieved from the web server with each other, and therefore each peer gets one piece from the web server and 9 pieces from the other peers in the same second, that is the maximum web server output bandwidth. The reason why the peers converge to a download speed that is slightly lower than the server output bandwidth is that peers can download the same piece at the same time or with a little delay each other, and therefore the piece in question will not be shared, decreasing by a small quantity the download speed of the peers.

With the analysis of this experiment we can conclude that the hybrid download

has two main benefits: a) the web server has to distribute the content only once and b) each peer receives a total bandwidth that is equal to the web server's output bandwidth. These are the same conditions as when each client is the only one to download from the web server, regardless the number of clients.

5.1.5 Intelligent bandwidth balancing policy

As previously stated, a new project was born from TU Delft's collaboration in 2010 with Wikipedia. The goal was to study the possibility of including P2P-based video streaming service in the Wikipedia pages. Since Wikipedia relies on donations to fund their infrastructure, low cost video delivery is a strict requirement. This is a good reason for adopting a P2P based solution instead of a central server based architecture such as content delivery infrastructure.

Since low server load is the main interest and, at the same time, low playback start-up delay has to be guaranteed, a smart bandwidth balancing policy is required. The policy is based on the concept that the web server has to support as less load as possible; therefore, when downloading a video, the bandwidth priority is given to BitTorrent, and HTTP has to be used as a fall-back to guarantee playback stability.

The behaviour of the bandwidth balancing policy is shown in figure 5.7. For clarity, the same results have been plotted by applying some curve smoothing in figure 5.8, making the graph easier to read. We will refer to figure 5.8 in the following analysis. The red line represents the total incoming bandwidth, as a combination of the HTTP and the BitTorrent bandwidth. As one can observe, the total incoming bandwidth waves around the video bit-rate value; the cause for such a behaviour depends on an internal video buffer that, as it gets filled or emptied, requires to the engine more or less bandwidth, increasing and decreasing the download speed.

The blue line, representing the incoming BitTorrent bandwidth, is generated by alternating two peers sharing the video at two different speeds: one peer seeding at 38 KB/s and another peer seeding at 18 KB/s. Around 190-420 seconds range, the incoming BitTorrent bandwidth is slightly higher than the video bit-rate, and therefore almost no HTTP bandwidth, represented by the green line, is required.

When there is no bandwidth coming from the BitTorrent network, HTTP has to supply with enough content to keep the video playback sustainable. As an example, we can notice that during 350-400 seconds range there is no incoming BitTorrent bandwidth, and therefore the missing bandwidth has to be requested to HTTP.

Finally, when the incoming BitTorrent bandwidth is not enough to keep the video playback sustainable, like between second 70 and second 130 or between second 250 and second 340, the missing bandwidth is compensated by HTTP. Figure 5.9 shows an alternative view of the same experiment results, where it is easier to understand what the bandwidth ratios are.

To conclude, this experiment show how different configurations of the band-

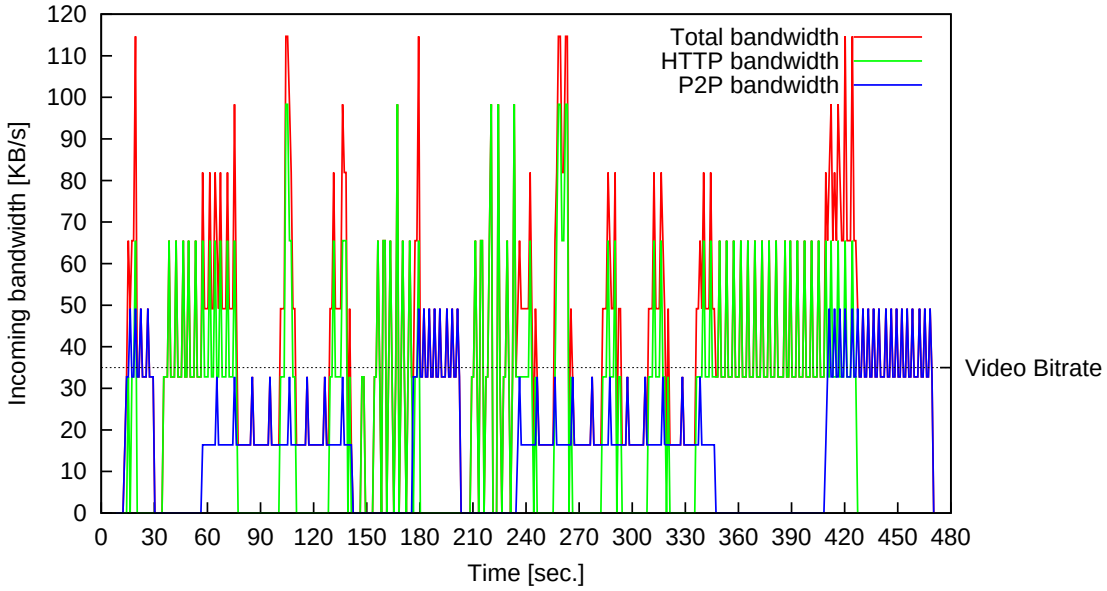


Figure 5.7: *Intelligent bandwidth balancing policy behaviour. The priority is given to the BitTorrent network. The missing bandwidth required to keep the video playback sustainable is integrated with HTTP bandwidth. One of the effects is that the server load is as low as possible to meet the video playback sustainability requirement.*

width balancing easily lead to coping with the environment requirements. In this case the requirement is to support BitTorrent with HTTP in the task of video streaming service, keeping low server load as a priority. In the previous experiments, on the other hand, the BitTorrent bandwidth is used to support the HTTP one. By just tuning a few parameters, the best performance can be achieved by the best qualities of HTTP or BitTorrent.

5.2 Pollution prevention experiments

The *Pollution Prevention* algorithm introduces some restrictions in the bandwidth usage ratio. This is important mainly for the server side results, since more data is required from the server to guarantee content integrity. For this reason a simulator of the *Pollution Prevention* algorithm has been developed with the purpose of measuring the server load in different conditions, depending on the file size, swarm size and quantity of seeders.

Since the results of the experiments where the file size and swarms size parameter variations are not meaningful to the end of analysing the server performance, but they just show that the download ratio between HTTP and BitTorrent keeps constant over the variation of those parameters, they will be not presented. The at-

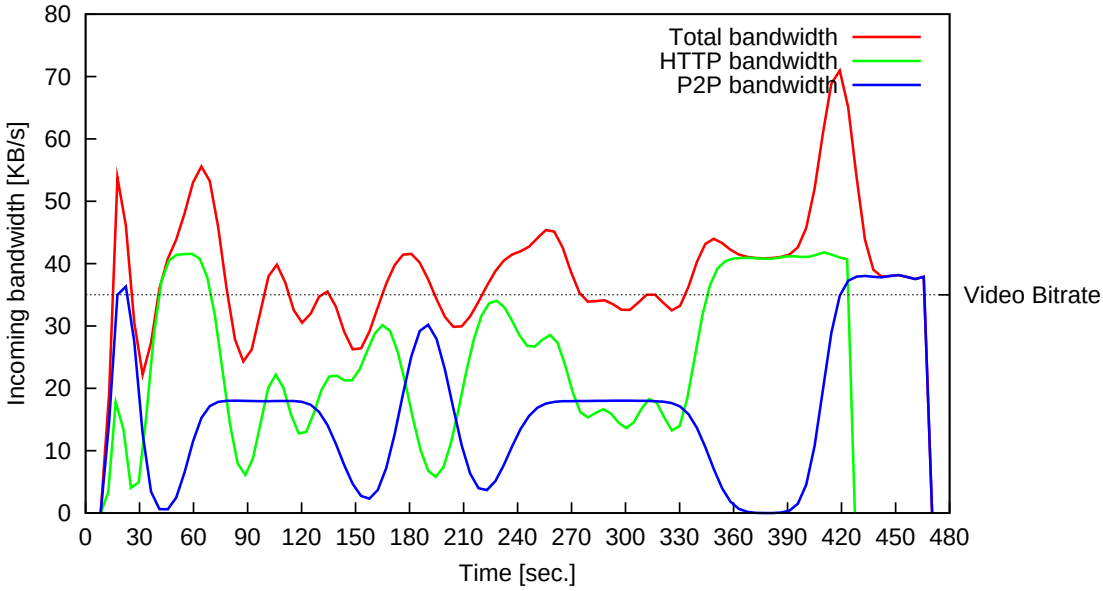


Figure 5.8: *The same results of figure 5.7 have been plotted with curve smoothing to improve the readibility of the graph.*

tention will be concentrated on the experiments where the number of seeders in the swarm variates.

This implementation simulates DHT, web server and peers. Each peer downloads a piece from the web server or from another peer depending on the piece selection instructed by the *Pollution Prevention* module. No network data transfer is involved, and the unchoking algorithm is simplified to the minimum. The simulator just analyses the amount of pieces retrieved from the web server and from the swarm. In particular the server load variates depending on the amount of seeders in the swarm. The reason is that a small amount of seeders allows the root hash to be spread among the peers, enabling the peers to trade pieces instead of being forced to download the content from the web server. Indeed, when a peer has no root hash, either received from another peer or computed after retrieving all the content from the server, it can not share its content since it is not possible to generate a *piece message* containing all the needed sibling and uncle hashes.

These experiments demonstrate that only a minimum amount of data needs to be retrieved from the web server to perform content integrity checks over the data retrieved by other peers. The most important outcome is that a minimum amount of seeders (around 5%) in the swarm reduces the web server load by a factor of more than the 70%.

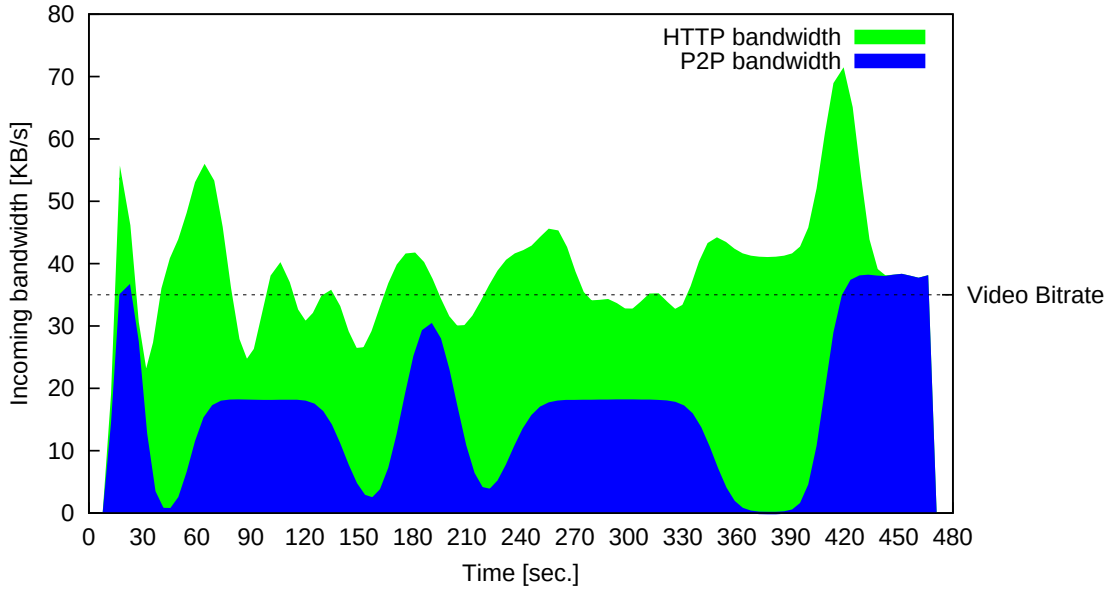


Figure 5.9: *A filled curve representation of the results shown in figure 5.8. Here it is easier to have an idea on what the HTTP and BitTorrent component ratios are.*

5.2.1 Server load on seeders ratio variation

The graph represented in figure 5.10 shows how the server load is influenced by the number of seeders in the swarm. The tests are performed in an environment where 100 peers are simulated, and the file size is 2048 pieces. The *5%*, *20%* and *100% seeders* experiments are performed over the 100 peers plus the 5%, 20% and 100% of seeders more. The *Start-up* experiment represents the initial phase in which all peers are leechers and start the download concurrently. Since there is no seeder in the swarm, there is no root hash and the peers cannot share pieces between themselves.

In the *2 steps* experiment, an initial set of 20 peer are started concurrently while a second set of 80 peers is started when the first set completed the 97% of the download. This gives the first set of peers the opportunity to retrieve more content from the web server and to compute earlier the Merkle tree once the download has completed. Once the first set completes the download, the root hash is spread in the swarm and gives to the second set of peers the opportunity of trading pieces, lowering the web server's load.

An analysis of the results shown in figure 5.10, demonstrates that a small quantity of seeders in the swarm allows a web server load reduction of more than the 90%, where, as a reminder, the server load is defined as the amount of content retrieved from the server over the total amount of content retrieved by all the peers of the swarm.

As an example, if the content size is 2048 pieces, and there are 100 peers downloading and 5 seeders, the total amount downloaded by all the peers is 204800 pieces; where 20480 pieces are retrieved from the server and 184320 pieces are retrieved from the P2P network.

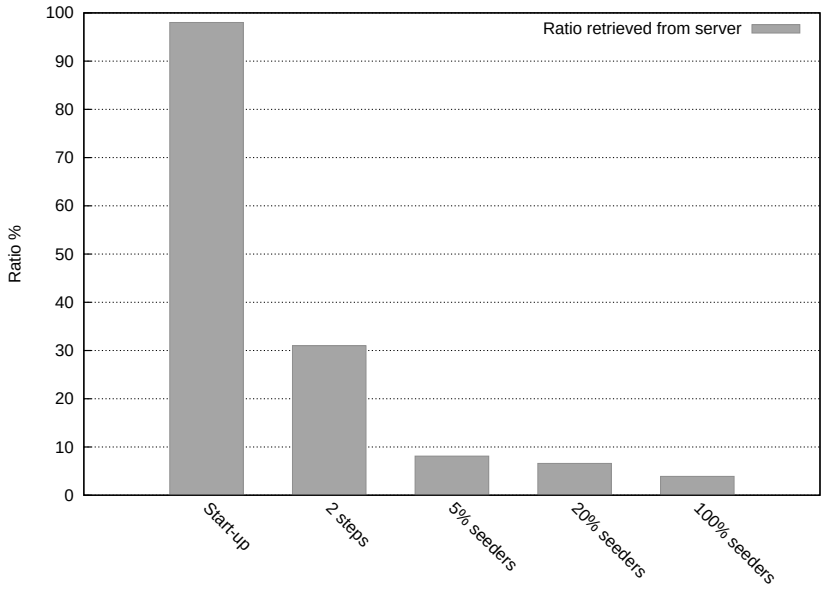


Figure 5.10: *Server load on seeders ratio variation. With the 5% off seeders in the swarm, the around the 90% of the total content is received from the P2P network and the 10% from the web server.*

5.2.2 Swarm download session simulation

This experiment aims to analyse the behaviour of the swarm, the server load and the HTTP/P2P pieces ratio during a whole download session. The difference with the previous experiment is that in this case the swarm changes over the time, while in the previous case the swarm conditions were predefined.

In this experiment, 3 sets of peers are started with different time delays. The total amount of peers started is 500; the size of the downloaded content is 512 pieces. The first set consists of 100 peers, started with a very small delay one from the other. We expect in this case to have all the peers downloading from the web server concurrently, since no root hash is circulating in the swarm. Since the peers are started with a small delay from each other, the first ones to be launched will have a little advantage that will cause them to complete the download earlier and to generate the root hash. The root hash will be distributed over the swarm, allowing the rest of the peers to share pieces and lower the server load.

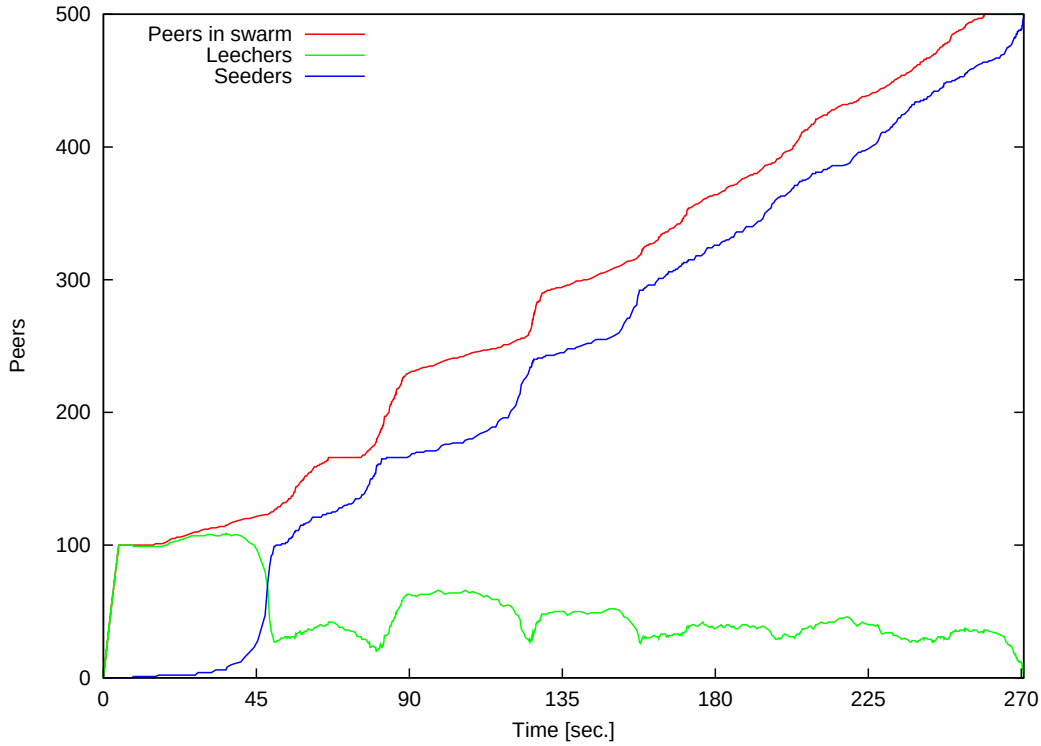


Figure 5.11: *Behaviour of the swarm during a download session.*

The second set consists of 66 peers also started with a small delay from one another. The expected behaviour for this set is not to load the server with many request, since a root hash should be already present in the swarm, allowing those peers to receive pieces from seeders and to trade pieces each other.

Finally, the third set, consisting of 334 peers, starts all the peers concurrently, with no delay. With no P2P support, this set would overload the web server with a large amount of requests. However, we expect the web server to be loaded only with around 10% of the total requests.

The 3 sets simulate three of the previous experiment cases: *Start-up*, *2 steps* and *20% seeders*, but in a scenario where all the peers are started continuously and not in different experiments.

Figure 5.11 displays the results concerning the swarm evolution. The red curve, representing the total amount of peers in the swarm, grows linearly over the time. The number of leechers has its peak in the first phase, when no root hash has been generated yet or entirely spread over the swarm; at the same time the number of seeders slowly grows. In the second phase, the first downloads complete and the root hash is generated and spread over the swarm, the number of leechers decreases and the number of seeders grows fast. After this transition phase, the number of

leechers remains constant during the swarm growth, and the number of seeders grows linearly like the swarm size.

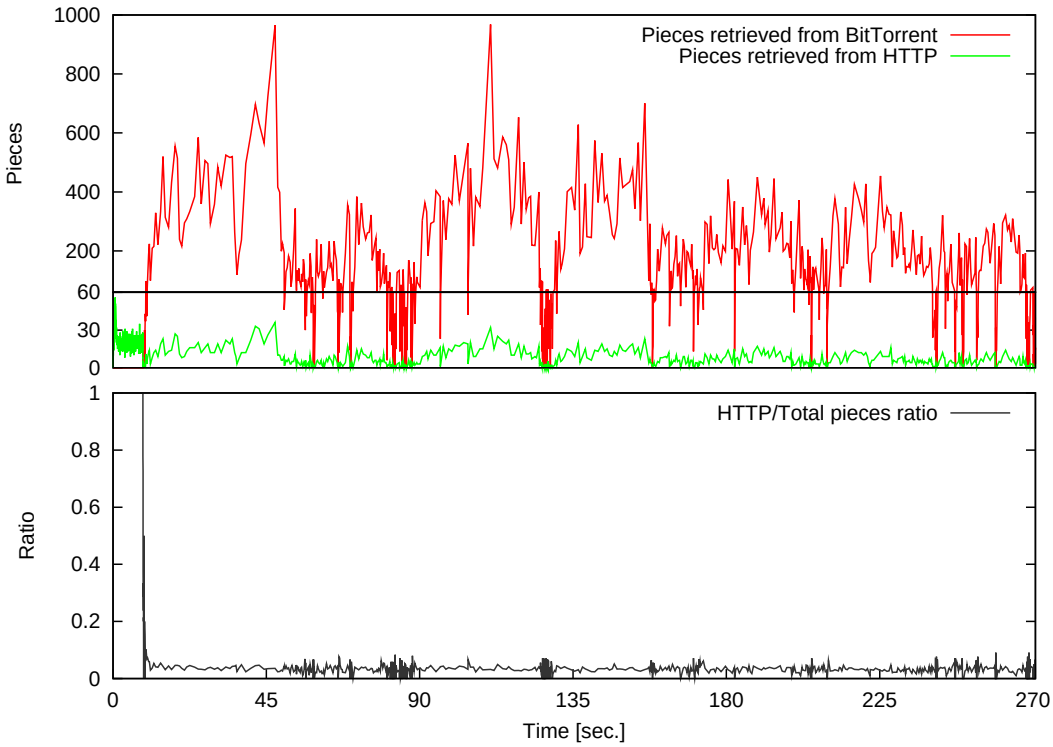


Figure 5.12: *Distribution of the load over HTTP and BitTorrent during the download session. The server load is defined as the ratio between the HTTP downloaded pieces over the total amount of downloaded pieces. The top graph has a scale enlargement in the range [0,60] on the Y axis to better show the behaviour of the downloaded HTTP pieces.*

Figure 5.12 displays the results of the experiment concerning the number of pieces downloaded and its ratio. As expected in the beginning phase, all the content is retrieved from the web server, as displayed by the green line in the upper graph. At the same time, the black line in the bottom graph shows that the ratio of the downloaded pieces is 1, meaning all the downloaded pieces are retrieved from the server. After this initial phase we can observer that the number of BitTorrent and HTTP downloaded pieces is constant, in the same way as the number of leechers remains constant over the same range of time. In this phase, the pieces ratio remains constant to a value under the 10%. This means that the 90% of the load generated by the swarm download is spread over the P2P network and only the 10% over the web server.

To conclude the analysis of this experiment, the results show that the web server load is higher in the first phase of the download session, when no seeder, and therefore no root hash, is present in the swarm yet, and when it's impossible for peer

to upload. However this is also the phase where the total number of peers in the swarm is relatively small. As the graph shows, the number of pieces retrieved from the server is reasonably small over the whole duration of the experiments, both when the HTTP/Total pieces ratio is high and when it is under 0.1.

5.3 Conclusions

The previous experiments show what the performances of hybrid systems are. The findings can be divided in two groups: client side and server side. In the client side we analysed how merging HTTP and BitTorrent bandwidths improves the speed of the download. We also analysed the improvements in the start-up delay, a parameter that is essential in the context of P2P based video on demand. In the *Flashcrowd* experiment we showed how clients and server benefit from the hybrid download: the server distributes the content only once, and the clients receive a bandwidth equal to the web server's output bandwidth. Then we demonstrated that, with an intelligent bandwidth balancing policy, an HTTP/BitTorrent hybrid can easily adapt to different needs, like for example trying to keep the download bandwidth constant.

In the server side results we analysed how the server load, defined as the amount of content retrieved from the server over the total amount of content retrieved by the peers, behaves depending on swarm growth and number of seeders. These results show that the web server is able to support a number of concurrent connections much higher than conventional HTTP. The Pollution Prevention experiments also show that, in an environment where HTTP2P is used, the condition for reducing the server load to the 10% is to have a minimum number of seeders, between 1% and 5% of the total swarm size.

Chapter 6

Conclusions

The web content is changing. HTTP content size and request frequency are growing. The popularity of the content can also affect server performances with phenomena like the *Flashcrowds*, generating service denial. At the same time, BitTorrent is leading the world of file sharing due to its ability to scale with big increases of peer requests. Therefore, the combination of HTTP and BitTorrent is a candidate solution to address this issue. The previous attempts of building a solution that combines HTTP and P2P have not had much success, but the need for a solution is still present.

The problem of supporting the HTTP protocol with BitTorrent in the task of web content retrieval stands mainly in the security aspect. Because of the nature of HTTP, the lack of piece hashes is a limit for guaranteeing content integrity. A few existing techniques allow the process of translating the URL of a web resource into a BitTorrent download; however a technique able to guarantee content integrity in this environment is lacking.

This thesis work proposes the architecture of HTTP2P, a tool able of hybrid download, and the *Pollution Prevention* algorithm, a technique able to guarantee content integrity over the swarm, based on two main concepts: the trusted authority (web server) and the distributed security (through the *invalidate message*).

Our experiments have shown that combining the best of the HTTP and the BitTorrent worlds greatly improves the performance of a download session, both on the server and the client side. The proposed architecture and the *Pollution Prevention* algorithm solve the problem of supporting HTTP with the BitTorrent protocol. However, also the effects of supporting BitTorrent with HTTP, that is the HTTP seeding technique, have been analyzed in this work, as part of an hybrid system.

In a scenario where the web content is growing, where the number of connections is increasing and where the content popularity is a fundamental parameter, the combination of HTTP and BitTorrent brings consistent improvement to a download session. These improvements are experienced in the client side, where the download speed is improved and the download start-up latency is reduced, and in the

server side, where the load of the server is distributed over the swarm and therefore more connections can be supported.

Bibliography

[1] Azureus introduces dht layer. http://www.slyck.com/news.php?story=772.

[2] Bittorrent enhancement proposals for dht. http://www.bittorrent.org/beps/bep_0005.html.

[3] Bittorrent protocol specification document. http://wiki.theory.org/BitTorrentSpecification.

[4] Dht definition. http://en.wikipedia.org/wiki/Distributed_hash_table.

[5] Firecoral, browser-based peer-to-peer content distribution network project. http://www.firecoral.net/.

[6] Getright's style web seeding specification. http://getright.com/seedtorrent.html.

[7] A new http header that might be useful. http://clubtroppo.com.au/2009/01/14/a-new-http-header-that-might-be-useful/.

[8] P2p-next swarmplug-in trial. http://trial.p2p-next.org/.

[9] Slashdot effect definition. http://en.wikipedia.org/wiki/Slashdot_effect.

[10] Tribler p2p client. http://www.tribler.org.

[11] Video in html 5. http://www.html5video.org/.

[12] Torrentfreak article about swarmplug-in. http://torrentfreak.com/bbc-trials-bittorrent-powered-hd-video-streaming-091203/, 2009.

[13] Wikipedia p2p video streaming test. http://wikipedia.p2p-next.org/, 2010.

[14] Ismail Ari, Bo Hong, Ethan L. Miller, Scott A. Brandt, and Darrell D. E. Long. Modeling, analysis and simulation of flash crowds on the internet. In *Technical Report UCSC-CRL-03-15*, 2004.

[15] Arno Bakker. Merkle hash torrent extension. http://www.bittorrent.org/beps/bep_0030.html, 2009.

[16] Arno Bakker, Riccardo Petrocco, Michael Daley, Jan Gerber, Victor Grishchenko, Diego Rabaioli, and Johan Pouwelse. Online video using bittorrent and html5 applied to wikipedia. In *Conference on Peer-to-Peer Computing (P2P), 2010 IEEE Tenth International*, 2010.

[17] Prithula Dhungel, Di Wu, Brad Schonhorst, and Keith W. Ross. A measurement study of attacks on bittorrent leechers. In *7th International Workshop on Peer-to-Peer Systems*, 2008.

[18] R. Fielding, UC Irvine, J. Gettys, J. Mogul, H. Frystyk, L. Masinter, P. Leach, and T. Berners-Lee. Rfc 2616: Hypertext transfer protocol – http/1.1, June 1999.

[19] Michael J. Freedman, Eric Freudenthal, and David Mazieres. Democratizing content publication with coral. http://www.scs.cs.nyu.edu/coral/.

[20] Jerome Harrington, Corey Kuwanoe, and Cliff C. Zou. A bittorrent-driven distributed denial-of-service attack. In *3rd International Conference on Security and Privacy in Communication Networks (SecureComm 2007)*, 2007.

[21] John Hoffman. John hoffman's style web seeding specification. http://bittornado.com/docs/webseed-spec.txt.

[22] S. Iyer, A. Rowstron, and P. Druschel. Squirrel: a decentralized peer-to-peer web cache. In *Proceeding PODC '02 Proceedings of the twenty-first annual symposium on Principles of distributed computing*, 2002.

[23] Jaeyeon Jung, Balachander Krishnamurthy, and Michael Rabinovich. Flash crowds and denial of service attacks: characterization and implications for cdns and web sites. In *WWW '02 Proceedings of the 11th international conference on World Wide Web*, 2002.

[24] Peter Maymounkov and David Mazieres. Kademlia: A peer-to-peer information system based on the xor metric. In *Proceeding IPTPS '01 Revised Papers from the First International Workshop on Peer-to-Peer Systems*, 2002.

[25] J.J.D. Mol, A. Bakker, J.A. Pouwelse, D.H.J. Epema, and H.J. Sips. The design and deployment of a bittorrent live video streaming solution. In *ISM '09 Proceedings of the 2009 11th IEEE International Symposium on Multimedia*, 2009.

[26] Diego Andres Rabaioli. Tribler browser plug-in. `http://www.tribler.org/trac/wiki/BrowserPlugin`, 2009.

[27] D. Serenyi and B. Witten. Rapidupdate: Peer-assisted distribution of security content. In *IPTPS'08 Proceedings of the 7th international conference on Peer-to-peer systems*, 2002.

[28] T. Stading, P. Maniatis, and M. Baker. Peer-to-peer caching schemes to address flash crowds. In *Proceeding IPTPS '01 Revised Papers from the First International Workshop on Peer-to-Peer Systems*, 2002.