RICH METADATA DISSEMINATION IN PEER-TO-PEER OVERLAY NETWORKS

Candidato: Andrea Reale

Relatore: Prof. Dott. Ing. Paolo Bellavista

Correlatori: Prof. Dr. Ir. Johan Pouwelse
Ir. Nitin Chiluka

Anno Accademico 2009/2010 - Sessione II
Don’t panic.
# Contents

**Introduction**  1

1 **Background**  3
   1.1 Peer-to-Peer Systems  3
   1.2 Overlay Networks  4
      1.2.1 Structured Overlay Networks  4
      1.2.2 Unstructured Overlay Networks  6
   1.3 BitTorrent  7
   1.4 Tribler  9

2 **Problem Definition and Related Work**  13
   2.1 Motivations and Vision  13
   2.2 Problem Description  15
      2.2.1 Rich Metadata  16
      2.2.2 The information dissemination problem  16
   2.3 Current solutions  17
      2.3.1 Metadata dissemination in peer-to-peer  17
      2.3.2 Information Dissemination in Mobile Ad-Hoc Networks  20
      2.3.3 Related Work on Metadata  22
   2.4 Discussion  24
      2.4.1 Integrating standards in our system  25
   2.5 Goals  25

3 **Metis: Model and Architecture**  27
   3.1 Requirements, Assumptions and Goals  27
   3.2 Domain Model  29
   3.3 Failure Model  31
   3.4 Problem Statement  31
   3.5 An Architecture for gossip-based dissemination  33
      3.5.1 Gossip protocols  34
      3.5.2 Structure  35
      3.5.3 Interactions  37
4 Design

4.1 Platform description
   4.1.1 Tribler
   4.1.2 BuddyCast
   4.1.3 ChannelCast

4.2 Subtitles Dissemination
   4.2.1 Domain Model

4.3 Design Overview

4.4 Protocol Description
   4.4.1 Subtitles Ingestion
   4.4.2 Subtitles Announcement
   4.4.3 Subtitles Retrieval
   4.4.4 Spam reduction and good metadata diffusion speed up

4.5 System Architecture
   4.5.1 Structure
   4.5.2 Interactions

4.6 Metis possible limitations and extensions
   4.6.1 Direct peer-to-peer exchange of subtitles
   4.6.2 Duplicate Announcements Overhead
   4.6.3 Infection models for Epidemic protocols

5 Implementation Details

5.1 Preliminary Knowledge
   5.1.1 Implementation Guidelines
   5.1.2 Packages Structure
   5.1.3 Tribler API Basics

5.2 Metis Implementation
   5.2.1 Domain Objects and Persistence
   5.2.2 Announcements Gossip
   5.2.3 Subtitles Retrieval
   5.2.4 Service External API

5.3 Testing the System
   5.3.1 Unit Testing
   5.3.2 Integration Testing

6 Metis Experimental Evaluation

6.1 Experiment Methodology
6.2 Experimental Setup
6.3 Experimental Framework
   6.3.1 Emulation Control
   6.3.2 Logs Analyser
6.4 Swarm behaviour description
   6.4.1 Number of Peers
   6.4.2 Churn Rates
Introduction

Production and distribution of contents today has become easier than ever. Hardware devices such as audio and video recorders able to store gigabytes of user produced multimedia, are available for very cheap prices; portable, mobile and even wearable devices capable of creating and serving rich multimedia content are becoming gadgets of everyday use; the capillary diffusion of ubiquitous Internet connectivity, advances in digital distribution technologies, along with the adoption of standard and interoperable compression technologies such as MPEG-2 [33] or MPEG-4 [34] make this content accessible to a continuously growing audience.

Among the consequences of this phenomenon there is the explosion of the quantity of multimedia available to be consumed, both of the “traditional” type (i.e. created and distributed by traditional content providers such as broadcasting companies) or user generated. In such a scenario it becomes vital to have adequate tools to search, browse and catalogue the items; we believe that one of the main enabling factor for a successful development of such services is the availability of quality rich metadata, describing and expressing the meaning of the objects; human or software users could then use these data to discover what is in a set of items without having to actually access the information itself, and leverage this knowledge in order to implement smart search and filtering algorithms exploiting the semantic relationships between the available audiovisual content.

In this Thesis we will especially focus on the technologies necessary to leverage the informative power of metadata in the context of peer-to-peer (P2P) content distribution systems. In this kind of systems the lack of centralised components managing the information and the natural abundance of it makes it even more important to have an efficient infrastructure for content discovery, and we believe that the availability of rich metadata can be a pillar on top of which to build a modern interactive content delivery platform.

Contributions of the Thesis

In this Master’s Thesis we will analyse the problem of audiovisual metadata distribution in the context of peer-to-peer content-oriented systems and design a possible solution to it. Our main contributions are:

- Dissect the constraints, requirements and desirable features for a fully-distri-
Propose a metadata dissemination solution based on epidemic information distribution, integrated with an existing peer-to-peer platform.

Implement a prototype of the proposed solution and evaluate its effectiveness and its limitations through a deep analysis based on a thorough emulation of a real system deployment.

Propose and evaluate an alternative dissemination protocol based on Bloom filters, aiming to reduce bandwidth wastes, and experiment it through a simulation framework.

We developed an important part of this work during a six months collaboration with the Parallel and Distributed Systems group at the Delft University of Technology. They helped us with their guidance and expertise in the context of peer-to-peer systems and we integrate our system in Tribler, the production quality P2P platform they are developing since more then four years as part of the P2P-Next European project [1] for the next Internet TV standard. The solution proposed by this thesis, Metis, makes an extensive use of Tribler’s services, building on its integrated gossip engine, BuddyCast [59], to develop our lightweight epidemic information distribution system. Moreover, we design a very simple and ad-hoc P2P request-response protocol which allows peers to retrieve the actual metadata content and leverages the knowledge acquired through epidemic contacts in order to find possible sources within the overlay to download that content from.

Our final goal is to contribute to the creation of a platform which combines the well known robustness and scalability of peer-to-peer systems with a metadata-based content-centric approach, in order to encourage a shift from simple file-sharing applications to multimedia centres having Users and Contents as their foundations.

Thesis Outline

The thesis is organised as follows. The necessary background knowledge about P2P systems concepts is given in Chapter 1. In Chapter 2 we describe the Metadata Dissemination problem and survey existing solutions in both the context of peer-to-peer and other akin environments. The requirements and the goals of our systems, and an abstract model describing it are discussed in Chapter 3 and a design proposal which instantiates that model is presented in Chapter 4; then, in Chapter 5, some details about its implementation are given. Chapter 6 presents the results of our extensive emulation-based experiments and introduces the alternative Bloom-filter based dissemination variant analysing its performance through the outcome of a series of simulations. In the last Chapter we present our conclusions and propose several future directions of development.
Chapter 1

Background

This chapter will briefly describe the technologies forming the context in which this work has been developed. Their peculiar characteristics will be analysed focusing on their benefits and the challenges they bear, and some basic concepts and terminology that will be used throughout the remainder of the Thesis will be introduced.

In Section 1.1 the common features of peer-to-peer systems are outlined. Section 1.2 briefly describes some of the architectures that emerged to cope with the challenges of peer-to-peer networks. Finally, in Sections 1.3 and 1.4 the BitTorrent and Tribler systems, that were chosen as the deployment target of our work, will be introduced.

1.1 Peer-to-Peer Systems

The term peer-to-peer system, often abbreviated to P2P, indicates a particular type of distributed computing system following a series of common architectural principles, which can be basically summarised in the following:

- Any participant (also referred to as peer) contributes with a portion of its resources to achieve the desired functionality. The contributed resources can be of any kind, for example bandwidth, CPU time or disk storage.

- There is no strong distinction between the roles of nodes in the system: each of them can act both as supplier or consumer of the provided services, without the need of external coordination or synchronisation agents. \(^1\)

- Membership in the system is extremely dynamic and managed by ad-hoc solutions. This part is so critical that usually is what makes a P2P system different from another.

\(^1\)We will see how in some systems this statement does not completely hold, since they present some form of weak distinction between roles.
This significantly deviates from the more traditional *Client-Server* architecture, in which a limited set of nodes (just one in the most simple case) struggle to produce services to be consumed by a large set of clients.

The most famous family of P2P systems that have come to public attention includes file-sharing networks such as Napster [47] or FastTrack (KaZaA) [30], probably mainly for the legal matters that were often linked to them. Nonetheless other types of peer-to-peer system variants exist accomplishing different tasks: examples can be the SETI@home project by the University of Berkeley [4], which uses donated computational power to analyse signals coming from the Universe, or Skype [8] which enables low cost - high quality phone calls between computers and standard telephony, or OceanStore [40] a service for global persistent data storage designed for high availability and scalability.

The contribution in terms of resources by all the members of the systems make peer-to-peer systems inherently scalable with respect to the number of participants on a theoretical level, and the lack of central components together with the possibility of achieving a high degree of resource replication make them extremely robust to single or multiple failures. However, several relevant challenges are to be faced; the absence of information centralisation makes coordination and synchronisation between peers an extremely hard task, and the dynamic nature of the set of participants makes membership management and message routing the most critical parts of a P2P system. A good survey describing characteristics and critical design points of peer-to-peer architectures is provided by Androutsellis-Theotokis et al. [5]

### 1.2 Overlay Networks

Modern peer-to-peer systems build on the concept of *Overlay Network*. An overlay network is, in its general definition, a network built up another pre-existing network. A P2P overlay network is a network abstraction built on top of the IP network and acts for the peers as a communication framework offering a wide range of basic services such as membership management, message routing and delivery, data location, failure tolerance, and sometimes also more advanced ones such as node authentication or anonymity. Lua et al. [49] provides a comprehensive survey of peer-to-peer overlays; he identifies two classes within them: *Structured* and *Unstructured* overlays.

#### 1.2.1 Structured Overlay Networks

The key idea of structured overlays is to logically arrange peers and data in the system following a well defined network topology and try to keep it despite nodes joining and leaving. The properties of this structure are then used to locate nodes or data inside the overlay. Most of the examples of structured overlay networks are based on the abstraction of *Distributed Hash Table (DHT)*. A DHT is a distributed
service for P2P data storage and retrieval, exposing an API that can be summarised in the two basic functionalities shown in Figure 1.1.

**put(key, value)** associates the given value to the key, and stores it in one or more nodes in the overlay.

**get(key) : value** given a key, retrieves the associated value from the peer(s) in the overlay responsible for it, and returns it to the caller.

Given this simple API, different DHT solutions vary from each other in the choice of the logical topology, the choice of the type of keys and the algorithm that assigns the responsibility for keys (and their associated value) to peers in the network. A usual approach is to assign every node an identifier from a given identifier space (e.g. 160bit strings), and to have keys chosen from the same space. A desirable property is to have a uniform distribution of node identifier and keys, therefore a way to assign identifiers is to generate them using a hash function (e.g. SHA-1) on something like the node’s IP address, and to generate the keys hashing the data value they refer to.

Given a mapping of node identifiers and keys, values are assigned to peers in the overlay exploiting some metric which defines the distance between the key and the node id in the identifier space.

For example Chord [68] arranges its 160-bit identifier space in a logical circle, and places peers in the circle according to the numerical value of their identifier. For each possible key $k$ it defines $successor(k)$ as the first peer in the circle whose identifier is equal to or follows $k$: whenever a $put(k, value)$ is issued, the middleware stores value at $successor(k)$, as shown in Figure 1.2 (a).

To perform a $get(key)$ operation, any node is provided with a so called finger table. A finger table contains $m$ entries, where $m$ is the size of the identifier space (e.g. $m = 160$). The $i^{th}$ entry in the table of node $n$ contains a pointer to the node who is $successor(n + 2^{i-1})$, as shown in the example in Figure 1.2 (b) (in the
successor(1) = 1
successor(2) = 3
successor(6) = 0

Figure 1.2: Chord: (a) An identifier circle in a 3-bit space, consisting of nodes 0, 1 and 3. In the example, key 1 is located at node 1, key 2 at node 3, and key 6 at node 0. (b) A different circle arrangement showing the finger table of node 3. Adapted from [68].

In each dashed line represents an entry of the finger table. In the Figure, when node 3 wants to lookup for key 6, it queries the node in its finger table whose identifier is closer to the key (in this case node 5), who in turn knows that the responsible for key 6 is node 7 by looking at his finger table.

The main benefit of DHT-based P2P overlays is that they have strong theoretical foundations, guaranteeing that a key can be found if it exists and giving upper bounds on the number of hops necessary to perform lookup operations. Moreover, the fact that keys and identifiers are uniformly distributed provides the system with natural load balancing capabilities.

Keeping the properties of the topology creates — on the other hand — a great runtime overhead in terms of overlay management cost. It has also been showed [62] that the performance of DHTs rapidly degrades when the network presents fast and continuous membership changes (phenomenon known as churn). Moreover given the agnostic nature of the process of identifier assignment, neighbouring peers in the overlay can be far, in terms of hops, in the underlying IP network, resulting in possible high communication latencies; for the same reason it is not possible to exploit any kind of form of data locality to boost performances for common lookups.

1.2.2 Unstructured Overlay Networks

The family of unstructured peer-to-peer overlay network includes a large variety of systems all sharing the common property of not arranging peers and data following a strict set of rules, unlike structured overlays. The actual topology of the network varies from overlay to overlay, spanning from random graphs to hierarch-
ical structures, and various different techniques have been used to implement data lookup and message routing including gossip-based techniques, random walks or expanding-ring searches.

For example the first versions of the Gnutella platform [38] used a completely flat topology. To join the network a peer contacts one of several well-known peers, and then announce themselves by flooding a join message with a limited TTL. Similarly distributed search for data is performed using a flood-based technique.

A study on the proprietary Skype network by Salman et al. [8] showed instead that its overlay is made out of a two-level hierarchical topology, with some peers having a greater amount of CPU and bandwidth resources — the superpeers — acting as relays for common peers.

The looser organisation allows unstructured networks to have a much cheaper cost for what concerns membership management algorithms; they show themselves to be extremely resilient to high churn dynamics, even when subject to pathological disconnections of nodes from the graph [70]. Moreover, the fact that data is not uniformly placed among peers in the networks favours the lookup performance for popular and highly replicated data items, in a way that is perfectly suitable for a mass market deployment scenario.

Nevertheless — unlike DHTs — unstructured overlays do not have theoretical guarantees about lookup and routing performances and effectiveness, and queries about uncommon values are generally penalised.

In the next two Sections we will introduce in more detail two unstructured overlay network: BitTorrent, a file-centric P2P system with centralised membership management, and Tribler, a P2P platform built around BitTorrent introducing several improvements such as decentralised membership management and search.

1.3 BitTorrent

BitTorrent [15] is a file distribution protocol created by Bram Cohen in 2001. Its main contribution is the introduction of a tit-for-tat mechanism to encourage voluntary contribution in terms of bandwidth: basically, it consists in peers offering their upload capacity to other peers that had given them data before. Due to this mechanism peers with high upload rates will likely have high download rates as well.

BitTorrent is a file-centric protocol, meaning that the unit of distribution is the file: the goal of the protocol is to make replicas of a file as fast as possible at its participants hosts. Any peer who has a complete replica of the contents to share is called a seeder. The other peers trying to get a copy are called leechers. The set of seeders and leechers (i.e. all the peers) for a single file is known as swarm.

The peer who owns the first copy of the file needs to create the swarm for it; first of all the file is logically divided into smaller fixed-size pieces, and an integrity hash for each piece is created: the piece size, and the list of piece hashes are referred to as Info Dictionary. The Info Dictionary is included into a file commonly known
as the .torrent file (from its extension), along with a descriptive name and the total file length; An additional SHA-1 hash of the Info Dictionary is computed: named infohash it is used as a unique identifier for the swarm. Finally, the initial seeder needs to announce itself to a central component called tracker.

In the BitTorrent architecture the tracker keeps track of the peers which are currently in a swarm, and runs a lightweight protocol over HTTP (or more recently also over UDP) used to expose a membership management service. The address of the tracker is also included in the .torrent file.

Any other peer who wants to join the swarm needs a copy of the .torrent file, from which it reads the swarm’s identity and information about the pieces. He then contacts the tracker to obtain a list of peers with whom it can start bartering data. Figure 1.3 provides an overview of the entities involved in the protocol, and an example of a peer joining a swarm.

Once in a swarm a peer can exchange data and messages with each other; however, it is important to remark the peers from different swarm cannot communicate through the BitTorrent overlay. Moreover, a peer’s identity is not kept trough different swarms and not even enforced between different sessions in a single swarm. These two design aspects of the protocol are important limitations because they prevent more complex interactions and policies between peers to be implemented in BitTorrent.

Another relevant problem of the protocol is the presence of a clear single point of failure and scalability bottleneck represented by the tracker: to overcome this problem several solutions have been studied [79, 18, 23, 48].

For instance, the BitTorrent Extension Proposal #5 (BEP 5) [48] defines a pro-
protocol for trackerless peer management, based on the Kademia DHT [53]; the responsibility of tracking the members of different swarms is distributed across peers implementing the protocol, using a Distributed Hash Table in everything similar to the one described in Section 1.2.1: every peer is assigned a 160-bit identifier, and each swarm is assigned to the peer whose id is closer to the swarm’s infohash using the bitwise XOR as metric; the peer will act as a normal tracker for it. To join a trackerless swarm the DHT can be queried with the infohash, and it will return the address of the peer who has tracking responsibilities for it.

Peer Exchange (PEX) [79] can be used complementary to the DHT to discover new peers from within a swarm. After a BitTorrent handshake, peers supporting PEX exchange a list of peers that they know to be in the same swarm from previous encounters.

Albeit those solutions are a promising alternative to the centralised tracker, they are showing themselves not to be really adequate to completely replace it: Crosby et al. [18] show that DHT lookup performance on the currently deployed systems is very poor, because of hash tables not being enough fast to reflect nodes disappearing from the network: hence an important fraction of time is spent contacting dead nodes, resulting in the 50% of lookups requiring more then 50 seconds to be completed. A very similar problem affects PEX: an extensive analysis on the performance of the Peer Exchange implementations conducted by Vliegendhart [75] demonstrates that peers lists tend to contain a relevant number of dead or not connectible peers: for example it is shown that after three hours from the reception of a message, only the 35% of the received peers is still contactable on average.

Finally, it is worth to remark that the BitTorrent protocol does not cope at all with the problem of content discovery: we said before that a peer needs the .torrent file to start a download, but we did not explain how it can retrieve or search for these files. In fact this is completely left out of the original specification; the general approach is for a content publisher to distribute their .torrents via external Web Servers, which also act as content directory service. An extension has been proposed [27] which allows peers to exchange .torrent files via BitTorrent messages: nonetheless this can happen only once a peer is already inside the swarm for which he needs the .torrent, and therefore does not help in the solution of the content discovery problem itself.

1.4 Tribler

Tribler [58] is a peer-to-peer platform based on the BitTorrent protocol that tries to overcome some of its design problems and add other interesting features.

Every Tribler peer has a unique and permanent secure identifier (shortly PermId) based on an Elliptic Curve Cryptography asymmetric scheme [71]: a PermId is a 120-bit ECC public key which can be used to identify a node across different sessions. By including message signatures created using the corresponding private key, it is possible for other peers to verify the authenticity and integrity of the
The Overlay Swarm is made up of all the Tribler Peers, and they use it to exchange messages.

Nodes are arranged in an Overlay Network, which uses BitTorrent as its main transport: the idea is to create a pseudo-swarm which every peer joins at bootstrap, called the Overlay Swarm. The Overlay Swarm is identified by a well-known infohash and it is joined by all the Tribler peers, as shown in Figure 1.4. Through this swarm clients are free to communicate via BitTorrent messages regardless of the possible other swarms they are in.

Following a two-layers architecture, as suggested by Voulgaris et al. [76], Tribler builds a Semantic Overlay Network. Unlike other unstructured overlays where links between peers form a random graph, the links in a semantic overlay network are built trying to capture relationships between peers, thus causing the spontaneous emergence of several interconnected clusters of peers with similar interests. Garcia-Molina et al. [24] show as this property can have a positive impact on search efficiency and effectiveness if compared to “uninformed” flood-base techniques or random graph walks. In fact, in an uniformed, unstructured overlay, whenever a peer issues a query for a document there is no a-priori clue about which node is able to answer it: hence, the common approach, as we have seen for instance in Gnutella-like overlays, is to flood the query to random nodes with a limited TTL and “hope” some of them will know the answer. As a consequence, it is necessary to reach a sufficiently big number of peers to have a good recall in the results; moreover the precision will be most likely low, since as a result of querying many nodes, a lot of not significant hits will be returned. A semantic overlay network, in-
stead, is usually significantly smaller than the underlying overlay which comprises all the peers in the P2P system, and includes only peers correlated by a sufficiently high degree of similarity. Therefore, issuing a query only among the peers in this smaller overlay not only has the benefit of involving a smaller number of peers in the network, thus improving search performances, but also leads to an higher precision in the results, since the probability of reaching peers having the desired answer is higher.

The core of Tribler’s semantic overlay stands in the BuddyCast [59] protocol which, using a gossip-based approach and an embedded recommendation engine, selects for each peer a semantically close neighborhood. To do so, a profile is built for each peer based on the history of content downloads. Profiles are exchanged through gossip messages and are given as input to the recommender which gives as output the set of most similar peers, called Taste Buddies. At each protocol round (which is every 15 seconds in the normal working conditions) a peer is selected to gossip with: the exchanged messages include the sender’s own profile, consisting of the top 50 content preferences, plus the contact information of 10 taste buddies and 10 random peers.

The peer selection algorithm selects a taste buddy with probability $p$ and a random peer with probability $1 - p$, with $p$ currently set to 0.5: avoiding to gossip with the semantic neighborhood helps discovering novel connections and gives the protocol an important variability factor. To prevent BuddyCast from contacting the same peers too often, a peer is contacted at most once every 4 hours.

Information exchanged through BuddyCast has a threefold role:

1. Formation of the Semantic Overlay.
2. Content discovery through content preferences exchange.
3. Peer discovery through peers exchange in gossip messages.

A more comprehensive description of the overlay protocol and the services built over it can be found in Section 4.1.1 or in [7].

**Summary**

The term Peer-to-Peer identifies a category of distributed systems following a series of architectural principles such as lack of central components, and participants sharing resources and cooperating in order to accomplish a common task. Examples of peer-to-peer systems are notorious file-sharing applications such as Napster, but also platforms for scientific computing such as SETI@home.

An overlay network is an abstraction commonly used to build peer-to-peer systems. It consists in a logic network laid over the standard IP network, and usually provides services like message routing and membership management.

Two families of overlay network emerged in literature. **Structured overlays** provide stronger theoretical foundations and guarantees about their services but
they have shown to be weak with respect to high churn rates. Although they have more fuzzy theoretical basis, unstructured overlays showed to be more resilient to highly dynamic environments and to have performances that are comparable to those of structured overlay networks.

BitTorrent is a file distribution peer-to-peer protocol relying on an unstructured peer-to-peer network organised as a random graph. It incentives peers donation of resources by implementing a tit-for-tat mechanism for file sharing. A BitTorrent overlay includes only peer sharing the same file, collectively called swarm.

Tribler is built around of BitTorrent, but tries to overcome some of its limitations, providing inter-swarm message routing, permanent and secure peer identifiers, and distributed content discovery services.
Chapter 2

Problem Definition and Related Work

In the previous chapter we briefly presented the common characteristics of the networking environment we are going to work with; in this chapter we define the problem that this Thesis will try to dissect and solve: rich metadata dissemination in peer-to-peer overlays.

In Section 2.1 our vision and the reasons that pushed us in pursuing this research topic are explained; Section 2.2 defines the features of rich metadata dissemination and tries to discern different aspects of the problem. In Section 2.3 we survey different existing solutions for information dissemination and metadata description both in the context of P2P, and in other akin contexts. Then, in Section 2.4 we discuss the surveyed solutions, trying to emphasise the limitations that we want to overcome. Finally, in Section 2.5 we expose the goals of this work in the form of concise list.

2.1 Motivations and Vision

Peer-to-Peer applications have been traditionally developed following the file sharing paradigm. In this kind of view, systems are built around the abstraction of file, treated as an opaque container of data. The products adopting this paradigm had a remarkable breakthrough because they offered a way to access types of content, such as digital music and video, that were not possible to find via different distribution channels. This has led to a fast wide scale adoption of P2P protocols, so wide that more then 50% of the Internet traffic in Europe is still now accountable to P2P [67].

Nevertheless, as studies on Internet traffic show [67, 41], this growth has now stopped, and an opposite tendency has started: in Figure 2.1 it is shown how the percentage of BitTorrent Traffic on well-known ports changed in the two years period of 2008-2009; even if the absolute values on the figure are to be considered erroneous since most of peer-to-peer data is exchanged via non standard ports, the
declining tendency is clear.

At the same time, other distribution solutions are emerging; Web based services for video on demand (VoD) like YouTube, BBC iPlayer or MegaVideo [65] are now becoming the main force carrying the Internet business. All these Web sites adopt a centralised distribution solution based on a HTTP+Flash combination, sometimes referred to as pseudo-streaming [25], serving videos directly to users’ browsers.

The shift towards this direction is confirmed by another measurement from Labovits et al. [41], shown in Figure 2.2: the graph shows the sudden growth of the percentage of Internet traffic due to Carpathia Hosting, home of the family of "Mega" services (MegaVideo, MegaUpload, ...).

We believe that the main factors that are leading to this change of direction are:

- Ease of use, enabling the user to consume content directly in their browsers without the need to install or use dedicated software for video playback.

- Availability of rich metadata such as subtitles, comments and timed captions tightly integrated with the video delivery experience.

- Quality of content discovery and search services, which go further file name based search and include the ability to look for videos based on their semantic content, their popularity, their publisher and plenty of other features.

However, with centralised solutions come serious disadvantages such as scalability limits, but more importantly a strong dependency on the service provider

Figure 2.1: Peer-to-Peer traffic on well known ports for the 2007-2009 period. Reproduced from [41]
which can lead to phenomena like easy and arbitrary censorship, or privacy issues, especially when dealing with user generated content.

For these reasons, we believe that there is still a need for fully decentralised solutions, and we recognise that there is an urge for them to evolve from the file-sharing paradigm to something able to involve users in a richer experience; we think that the success of Web based VoD services and the declining tendency of P2P are mainly due to the abundance, in the former, of rich metadata for distributed content. This metadata varies in types and representations, ranging from subtitles, timed advertisements and descriptions, to objective and subjective information about the content, thumbnails, or semantic tagging. Once this information is available, search algorithms can leverage it to improve and enrich their effectiveness, and GUI developers can integrate it with content browsing and consumption.

The vision we have is that of user-centric decentralised platforms, enabling users to colloquy with each other in a language that is based no longer on the opaque concept of file, but on the multifaceted concept of digital item, as the conjunction of the content itself and all the possible information surrounding it.

### 2.2 Problem Description

Given our vision, in this thesis we will try to answer the following research issue:

- Design, implement and evaluate a lightweight rich metadata dissemination system for content distribution platforms built on top of a fully decentralised P2P overlay network.
In this Section this statement will be explained and the requirements and challenges of the problem will be outlined.

2.2.1 Rich Metadata

The term metadata has been used in a big variety of contexts in Information Technology literature, and it generally has a very large range of meanings depending on how it is used. The Merriam-Webster dictionary defines metadata as “data that provides information about other data”: this definition is extremely general, and can be source of confusion. Thus, we find appropriate to dedicate some lines to specify the meaning that we want to attribute to it in this Thesis.

Our contribution will focus on metadata for audiovisual objects, identifying with this term the set of data able to describe, or augment with potentially interesting information, multimedia content such as a movie or a song.

We do not restrict a priori the types of metadata that our proposed system will be able to deal with, but we will tend to concentrate more on high-level types that we believe to be closer to what a user needs to improve its daily experience. Examples can be textual information, such as subjective descriptions of content or contextual knowledge about authors, thumbnails or video previews of a long running item, advertising information, intra-item linking, or timed metadata such as scene captions or subtitles. It is out of our interest to consider the description of low level features such as colour histograms for single scenes, or attributes like image patterns description, even if we recognise the importance of having this kind of attributes for a deeper semantic mapping of the items.

It is worth to clarify another possible source of confusion: in BitTorrent the word metadata is often used to indicate the content of the .torrent files, since they contain useful information about the download they describe: we will try to avoid to use this meaning of the world, and where it will not be possible, the unambiguous meaning will be clear from the context.

2.2.2 The information dissemination problem

With the term information dissemination we refer to the problem of pushing information to a set of nodes in an efficient and timely way.

Two main roles are clearly recognisable in a dissemination scenario:

publisher It is the node whose goal is to spread an information through the network. It initiates the dissemination process by providing the data to dissemnate.

destinations They are the nodes that are potentially interested in the information that is being distributed. Their goal is to receive that information as soon as possible. Depending on the scenario, all the nodes in the network, or only a subset can be potential destinations.
The easiest example of a dissemination system is the mailing list, where one publisher wishes to distribute one piece of information (the mail content) to a set of destinations that in this case coincides with the mailboxes of the mailing list subscribers.

The publish/subscribe paradigm is a commonly adopted model used to solve a subset of dissemination problems: information messages are divided in classes or topics, and destinations must explicitly express interest in one or more of those, hence becoming subscribers.

In our specific scenario, any peer can assume the role of publisher, and the set of destinations are all the peers in the Overlay Network. The information to disseminate is metadata about a digital content.

In the next sections we will look at how the problem has been faced in literature in different contexts, with a particular focus on P2P metadata dissemination.

2.3 Current solutions

2.3.1 Metadata dissemination in peer-to-peer

Miro

Miro [57] is an open source “Internet Video Platform” for video content distribution. Its core software integrates a Video Player, a BitTorrent client and an RSS Aggregator.

Miro uses RSS feeds as its main mechanism for content discovery: a user can specify the URL of an RSS document advertising video items, and have Miro periodically check it for updates. Inside the feeds, HTTP URLs pointing to .torrents for the video files are embedded, thus permitting the client to retrieve them and join the corresponding swarms. This combination of RSS + BitTorrent is commonly known as broadcatching [26].

Metadata about the items is distributed exploiting the same mechanism: for this purpose an ad-hoc extension provided by the Yahoo Media RSS Module [63] is used; this module permits to enrich the entries in the feed document with basic metadata such as content title, description, length and bitrate but also richer type such as comments, or URLs pointing to subtitles or thumbnails. The extracted data is used by the client for its search engine, allowing users to search content also on the base of richer semantic descriptions.

No facility is provided to allow publishers to create or host their RSS feeds. Nevertheless “Participatory Culture Foundation” — the no-profit company behind Miro — provides two different, external services to assist in the creation of a personal TV Channel:

- The Miro Guide [56] is a Web based content discovery service which allows its registered users to publish their feeds; the Miro client is automatically

---

1The term subscriber will be used with a different meaning in our proposed solution.
subscribed to the Miro Guide, and therefore any Miro user can browse and search it for content.

- The Miro Community [55] is a Web hosting service. It permits publishers to upload their videos and creates RSS feeds for them. However videos published in this way are distributed in a centralised fashion via HTTP pseudo-streaming [25].

Vuze

Vuze [31] (formerly known as Azureus) is one of the most widely used BitTorrent client. In 2006 the Azureus developers decided to change the name of their software to Vuze, and at the same time they tried to enrich what was a plain BitTorrent client with social features. As today Vuze has become a full fledged media platform, including an integrated video player and a content discovery service.

As for Miro, content discovery is supported through RSS broadcasting. Even though Vuze does not have a central directory similar to the Miro Guide, it subscribes by default to several “famous” legal torrent sites, and uses them as the basic search and browsing sources.

Content publishing and advertising via RSS is supported directly in the client via an installable plug-in: using this feature a user can easily create his own feed for content he wants to share via BitTorrent; the so created RSS document will include links for them in terms of Magnet URIs [50], and they are automatically published via HTTP through an integrated lightweight Web server.

Metadata is also supported via usage of appropriate RSS attributes, but the range of supported types is limited to textual descriptions, information such as content length and bitrate, and links to thumbnails. Comments are supported only for items that are published by some well-known torrent sites such as mininova.org: in these cases when a user clicks on the “Comments” button of an item, the Web site is opened in an external browser on the page referring the selected content.

Tribler

We already introduced Tribler in Section 1.4. From version 5.2 (released in February 2010) the concept of peer-to-peer moderations is supported; through the abstraction of channels any user can publish a set of torrents for other peers to search and browse. Channel owners can add elements to their publications either selecting single .torrent files, or providing external RSS feeds. No utility is currently provided to help creating feeds or .torrents.

Although no rich metadata type is currently supported (not even the metadata annotations available in RSS feeds), channels represents a first step towards this goal, since they provide the platform with basic and completely decentralised publishing tools.
Channels are implemented through the ChannelCast gossip-based protocol [6], which will be described in detail in Chapter 4, since we chose Tribler and channels as the foundation on which we built a prototype of our proposal.

Private Communities

A widespread phenomenon surrounding the file-sharing world is that of private communities. Private communities are basically Web sites offering users a high quality directory service for peer-to-peer downloads. Most of the private communities are not freely joinable: to become a new user is often required an invitation from an existing member. Central concepts around which their success is built are trust and reputation: community members need to reach and keep a certain level of reputation to stay in it, that is usually measured in terms of share ratio or contributions such as releases of new content.

Available items are displayed in dedicated Web pages which are created by the members themselves: high quality and trusted metadata about the content can be found on this page; they usually include textual metadata, thumbnails or subtitle files, and comments from other community members. All this data is retrievable from central servers using HTTP.

Examples of private communities for BitTorrent are TvTorrents.com, LastTorrents.org or Demonoid.com.

DIAS

DIAS [39], acronym for Distributed Information Alert System, is an information dissemination architecture developed in the context of the DIET European Project [52].

DIAS aims to build a lightweight peer-to-peer notification system, able to disseminate user generated events to interested peers. In DIAS peers manifest their interests in events by publishing their profile; a user’s profile consists in a permanent query expressed in an ad-hoc query language\(^2\) that can be matched against particular events specifications. The system guarantees to eventually deliver every peer all the notifications about events matching their profiles.

Figure 2.3 shows DIAS’s network topology; it is based on a two-level hierarchical model, with standard peers and superpeers, in a way similar to the Skype network presented in Section 1.2.2. Each standard peer is connected to one superpeer, which in turn is responsible for a fraction of the peers in the system. A peer sends its profile and the events it produces to its responsible superpeer, which forwards it to the other superpeers.

Whenever a superpeer receives an event notification, it runs an event-profile matching algorithm in order to determine the subset of nodes connected to it that
Figure 2.3: The P2P architecture used by DIA. Events producers $P_1$ and $P_3$ send notifications to superpeers, which in turn propagate them to interested peers. Adapted from [39].

are interested in the processed event, and sends notifications to them.

### 2.3.2 Information Dissemination in Mobile Ad-Hoc Networks

A MANET [14] consists of a set of mobile wireless terminals forming impromptu networks on the basis of nodes’ encounters while they move in a shared physical environment. Applications built on top of MANETs are able to offer a rich set of services that leverage the copresence in space of computing devices such as location based discovery services, or context aware social applications.

Mobile Ad-Hoc Networks and Peer-to-Peer networks share the key concepts of decentralization and self-organization, due to a common nature of their distributed components: P2P networks are made of a highly dynamic set of nodes connected to the Internet, while MANETs consist of moving nodes communicating with each other via multi-hop wireless links.

A work by Schollmeier et al. [66] provides an extensive comparison of the similarities and differences between them. Similarities can be summarised as follows:

- Frequently changing topologies.
- Lack of central coordination infrastructure.
- Low nodes’ reliability.

However strong differences are also recognisable:
Mobile networks can establish direct connections only within a limited physical distance constrained by the wireless range, while P2P Overlay connections have a theoretically unlimited range.

In many cases MANET devices have strict resource constraints; that is not the case in typical P2P Overlay deployments.

Connections and disconnection events in mobile networks are much more frequent due to mobility patterns (i.e. higher churn rate).

Routing in P2P overlays is mainly used for object or node location, while in MANET a routing path through possibly multiple hops needs to be found for every exchanged message.

Most of the outlined differences are originated from the different intrinsic goals of the two abstractions: on the one hand, MANETs are created in situations where users need to interact but there is lack of other forms of connectivity; on the other hand, P2P overlays are created when users need to collaborate to complete a task (such as locating an object), but connectivity does not represent a problem.

Nevertheless a lot of work (e.g. [9, 13, 29]) has been done to find synergies between the two research fields and to find generalised results that can be applied to both.

In the next Section we will survey four dissemination solutions developed to be deployed in MANETs, but that we think can be useful to describe the problem in a more general picture.

Current solution in MANETs

Repantis et al. [61] propose a system for data driven routing through which peers query each other to locate objects in the network. To do so they proactively build partial indexes of object locations through a dissemination protocol in which local views of items position in the network are spread.

Different dissemination policies are proposed and analysed: these policies vary in the choice of the destination nodes, and in the selection of what information is spread to them.

For instance, the most basic policy disseminates synopses about local items only to the peers within the communication range of the sender. More complex policies are based on the local history of the explicit queries exchanged between the users, exploiting the fact that terminals that made a big number of queries about objects published by a particular peer have higher chances to be interested in its disseminated information.

To ensure a minimum overhead that is independent from the nature of the advertised objects, the proposal uses a representation of the disseminated synopses based on Bloom filters [10].
Lee et al. [43] face the problem of information dissemination in the context of Vehicular Ad-Hoc Networks, and they design MobEyes, a middleware for proactive urban monitoring. In their solution the mobility of nodes in an urban environment is opportunistically exploited to perform both a passive and active dissemination of data sensed by the moving vehicles.

With passive diffusion normal vehicles spread data summaries using a k-hop carry and forward protocol, aiming to replicate the sensed data as much as possible. With active harvesting a limited number of mobile agents (a kind of superpeers) strive to collect all the generated summaries with the minimum possible overhead in terms of duplicates collection. In order to achieve this goal mobile agents perform a broadcast based query to their neighbours which includes, as in [61], a Bloom filter representing the set of already collected summaries. The receivers of the query then uses this structure to probabilistically check which subset of their summaries has to be sent to requester.

In [44] the same authors propose an algorithm to coordinate the mobile agents performing the harvesting process, which is based on the analysis of foraging behaviour of some species of animals. It basically mixes random walks with greedy searches to guide the harvesting process in areas where the information density is higher.

Fiore et al. and Costa et al. ([22] and [16]) propose two different information dissemination solutions, both based on the concept of information density. Instead of spreading data in any direction in the network, the key idea is to try to route it along directions that are determined by an esteem of information density, which represents the concentration of information in a given zone of the network.

The two approaches vary in how the density is calculated: in [22], every peer computes it from the view it has of the local neighbourhood based on the exchanged and overheard messages; in [16], the originator of an information must also create and distribute a so called propagation function, which maps space coordinates to numeric values. The function is then evaluated at receiving peers, which — depending on their position — forward the message following the descending gradient of the function.

2.3.3 Related Work on Metadata

When talking about metadata an important subject is interoperability between its representations from different sources. This enables it to be reused in different contexts for different uses, and improves the compatibility between different tools. For example search engines can leverage information carried by metadata to improve the effectiveness of their results: with a commonly accepted representation different algorithms can be integrated and can work seamlessly on the same data.

In the field of metadata — especially related to music and video content — several parallel efforts are being spent in order to define a common standard.
The efforts are twofold: from one side there’s the need to define a fixed model: this model must be able to define a common set of metadata concepts that is able to satisfy the needs of different publishers and for different types of items (e.g. movies, tv series, documentaries). On the other side, once the model is fixed, a common representation for the model is desirable to enable interoperability of tools.

Other complementary efforts are going to find a standard packaging solution for metadata and the items it refers to in the same logical container, thus allowing an easier distribution of rich content.

**Survey of current standards**

The Dublin Core Metadata Initiative [77] defines a content description model encompassing a restricted set of core elements that are able to describe the most common features for any kind of digital content. For more detailed descriptions and types they allow the extension of the core model via instances of a metamodel, known as Dublin Core Application Profiles. Since both the Dublin Core Metadata Element Set and Application Profiles are RDF instances they can be represented in any RDF representation, such for example XML. The Video Development Initiative (ViDe) [74] has developed a Dublin Core Application profile for Digital Video addressing the main needs of metadata for video content [2].

Another initiative is conducted by the Moving Pictures Experts Group (MPEG): through the MPEG-7 [35] standard they define a metadata model especially suited for video content. On the other hand, the MPEG-21 [36] standard deals with a complementary aspect of the problem, defining a packaging convention for digital items and all the related resources, including metadata. MPEG-7 metadata and MPEG-21 package description (called Digital Item Declaration, DID) can be represented using an XML serialisation.

The TV-Anytime Forum also developed a set of specifications regarding video delivery, also considering the metadata associated to it. The TV-Anytime Metadata Specification [72] defines a set of elements not dissimilar to those defined by MPEG-7. An XML schema to represent instances of those elements is provided as well.

A different approach is taken by the URIPlay [73] project. Aim of the project is to provide a service aggregating metadata about multimedia content from different sources. The goal is to build a database of playable URIs enriched with information about their content, their business model, their property rights, and other forms of metadata. To organise this kind of knowledge they define a metadata model as an OWL ontology, thus enabling reasoning based search over their database. Being OWL, this kind of data can be represented in any RDF representation, including XML.

Finally the P2P-Next project also defines a Metadata Working Group, whose effort is to standardise the metadata model used to describe items within Next-Share, the reference platform for interactive P2P video and audio delivery where
the research efforts and results of the project are implemented.

The P2P Rich Metadata specification [20] “provides a minimum set of attributes which are necessary to describe a content item in the Next-Share system”. An important effort in the definition of the specification has been spent to provide full-interoperability with the existing standards. For this reason three different mappings have been defined between this model and the MPEG-7, TV-Anytime, and the URIPlay models. An interesting idea of this specification is the separation between core metadata and optional metadata, as to facilitate its distribution. This idea leads directly to the idea of modular packaging of digital items, where the core data is packed together with the item to deliver, while the optional part are distributed separately and can be retrieved only if necessary. The packaging specification of the P2P-Next Metadata WG is available in [19], and is based on the MPEG-21 standard.

2.4 Discussion

Products like Miro and Vuze show that the peer-to-peer world has already started to move some steps towards the richer user experience we sketched at the beginning of this chapter. Both the platforms allow users to become publishers of their content, and to augment it with more or less simple metadata.

Items in Miro can be enriched with a wide variety of metadata, but its dissemination still relies on centralised distribution mechanism. Moreover, even if a full portfolio of services for user generated content publishing is available, the related process is not really straightforward and well integrated with the platform, and — again — it relies on external centralised architectures.

Vuze shows the same weaknesses of Miro, adopting the same metadata distribution solution; furthermore, it apparently uses only a limited subset of metadata types among those that are made available by the Yahoo! RSS Media Module. However it provides a good integrated plug-in to ease the publishing process, and, by allowing to put RSS documents on a Web-server local to peers, it provides a first contribution towards decentralisation. However, the URLs where feeds are retrievable are not disseminated in the network in any way, so it is up to the publishers to communicate them to their audience.

Another important aspect that neither Vuze nor Miro face is security: one of the problem that file sharing P2P networks have always had to cope with is the high concentration of polluted content and spam. It is reasonable to think that similar problems would arise also for user posted metadata; hence a complete diffusion solution should also be able to guarantee the authenticity of the distributed information, its authorship, and possibly filter low-quality or erroneous metadata.

We showed that although mobile ad-hoc networks share several common properties with peer-to-peer overlays, they pose different requirements and criticalities. Notwithstanding these differences, some of the results achieved in information dissemination in MANETs are worth to be considered and exported to the P2P world,
especially for what concerns the care about protocol overhead: even if wired P2P stations are presumably equipped with broadband connections, we believe that metadata dissemination should happen in “background” without interfering with users’ normal activity, hence not taking a significant amount of bandwidth from the one dedicated to content distribution itself.

2.4.1 Integrating standards in our system

Standards have an essential role to enable easy information exchange and interoperability between different tools and applications; in the P2P scenario, the adoption of an established and widely recognised representation of metadata for digital items could potentially allow to use different information sources to retrieve rich content descriptions and reuse them in different contexts.

Yet, as we have shown in Section 2.3.3 the field of multimedia metadata standards is still not mature. Efforts to date represent separate contributions with too narrow or too specific scope, and there’s been too little progress toward harmonising across them. The result is a metadata standards mixture that contributes more to the growing chaos of multimedia data than to the sense and organisation that metadata standards should promise.

Hence, whilst we firmly believe in the importance of standards and sustain their adoption, we decided not to explicitly adopt any in our work, but to define a very simple and ad-hoc format to represent the data exchanged in our system.

Nonetheless, we have focused our effort on the design of mechanisms and protocols for metadata dissemination as much as possible independent from data representation, and we leave for a possible future work a study focusing on a thorough analysis of the benefits and drawbacks associated to the adoption of a standard representation of metadata, and its integration in our system.

2.5 Goals

Given the state of the art of metadata diffusion in P2P platform, we believe there is still space for improvement. Hence we define the distinctive goals of the metadata dissemination solutions we present in this Thesis in the following points:

- Full decentralisation.
- Tight integration with the content distribution platform.
- Minimum overhead in terms of resources usage (especially consumed bandwidth).
- Robustness against spam and security threats.
Summary

In this Chapter we showed how, according to our vision, the simple file-sharing paradigm commonly used in Peer-to-Peer content distribution applications is no longer sufficient to offer modern users a satisfactory experience. We speculate that this may be the main cause for the declining tendency in the usage of peer-to-peer protocols, and we identify in the availability of high quality rich metadata for audiovisual content one of the crucial reasons for the success of centralised solutions. This suggests that there exists a need to transform file-sharing systems in user-centric systems, distributing no longer simple files but contents enriched with valuable metadata. Everything in a completely decentralised environment.

Existing solutions in peer-to-peer like Miro [57] or Vuze [31] offer only partial support to our vision, and to date, the only rich support for metadata available for peer-to-peer networks is provided by Private Communities.

We therefore identify the subject of this Thesis in the design and evaluation of a fully decentralised Metadata dissemination system to be integrate in a modern user-centric content distribution platform.
Chapter 3

Metis: Model and Architecture

In this Chapter a model for a possible peer-to-peer dissemination system will be outlined. We named this system Metis, as the name of the Titaness Metis in Greek mythology; as she instilled wisdom to Zeus from his belly, our system's role is to instill knowledge throughout the overlay.

The Chapter is organised as follows: first, in Section 3.1 a set of requirements for the system and the assumptions under which it is supposed to work will be listed and explained; Section 3.2 and Section 3.3 respectively describe the domain model showing the central concepts we will use in our work and their reciprocal relations, and the failure model, i.e., the erroneous and faulty situations we consider possible to occur. Then Section 3.4 will show our analysis of the problem that, given the requirements, leads to the architecture that we present in Section 3.5.

3.1 Requirements, Assumptions and Goals

The main goal of the system we designed is to disseminate user generated metadata within a peer-to-peer overlay network in a lightweight, secure and fully decentralised manner.

Given this purpose, we identify the following principal requirements:

**Requirement 1: Provide an API to publish, search and retrieve metadata.** The system should expose a simple interface allowing to use it as a service by peer-to-peer developers; through it they should be able to easily build applications (e.g. user interfaces) using metadata search and dissemination facilities.

**Requirement 2: Push published metadata to peers in the overlay.** As soon as metadata is published by a peer, the system should transparently start the dissemination process, spreading it to the network by following a push model.

**Requirement 3: Guarantee the integrity and authenticity of metadata.** Any peer receiving metadata from our dissemination system should be able to
verify their integrity. It should also be able to check the identity of the publisher.

**Requirement 4: Fulfil all the requirements using no central components.** In our system decentralisation is a must. The overall metadata dissemination system should continue to work despite single and multiple failures of peers in the system. Nonetheless we accept the dissemination of a single metadata information to fail in the case all the involved peers fail together.

The system will be designed to work on top of a pre-existing content distribution platform laid over a peer-to-peer overlay network. We make several assumptions regarding the services offered by the middleware implementing this overlay abstraction. The most important are:

**Assumption 1: Digital items have a unique identifier within the overlay.** The content distributed via the underlying platform is identifiable with a unique name. The uniqueness of the name must be guaranteed only within the overlay. For instance, using BitTorrent as the content distribution protocol, the swarm *infohash* (see Section 1.3) would be a good candidate as an item’s identifier.

**Assumption 2: Peers have unique and secure identifiers, that are persistent across different sessions.** The middleware creating the overlay assigns every peer a unique identifier, which can be used to refer it throughout the network. This identifier is *secure*, i.e., cryptographically difficult for a malicious peer to assume another peer’s identity, and it persists across different sessions.

**Assumption 3: The overlay provides message routing based on peer identifier.**

Peers are able to send messages to other peers just knowing their identifier. The overlay takes care to locate the destination node and to deliver the message to it.

**Assumption 4: It is always possible to send and receive messages to / from an on-line peer.** This is the strongest of our assumptions. It means that every peer has a reachable port and IP address, which is in clear contrast with the reality of P2P systems, where a significant percentage of them are behind NAT devices or firewalls [46]. Nonetheless modern platforms usually implement a set of NAT traversal techniques that can overstep these obstacles: we assume that they are sufficiently effective to offer an abstraction of always connectible peers.

It has been remarked several times that peer-to-peer systems show an extreme dynamism, which makes difficult — if not impossible — to accurately model and predict their behaviours. Therefore it is not of our interests to guarantee the theoretical optimality of our solution; nonetheless we worked on our design trying to achieve best effort performance for the following parameters:
Goal 1: Maximise peer coverage of metadata spread. We want metadata published by one peer to ideally reach any other peer in the network. Moreover, a desirable property would be to incentivize the diffusion of high quality metadata, and do the opposite for low quality or erroneous metadata.

Goal 2: Maximise the speed of the dissemination. Any peer in the network should receive metadata information that is being disseminated within the minimum possible time window from the moment it is published.

Goal 3: Minimise bandwidth usage on single nodes. The process of metadata dissemination should use as less bandwidth as possible, and should not interfere with the bandwidth usage for normal content distribution.

Goal 4: Maximise the availability of metadata contents throughout the overlay. Even if every metadata content is generated by a single peer in the network, it should be replicated and made available in the overlay as much as possible in order to avoid the publisher becoming a single point of failure for its publications. This is even more true for highly popular meta-contents which should be privileged.

3.2 Domain Model

The system we will describe is designed to be a complement of a possible modern peer-to-peer content distribution platform. For this reason we will build on top of some basic abstractions and functionalities that are commonly available in this family of products.

For instance, we will suppose that the platform provides facilities to download content via a peer-to-peer protocol, such as BitTorrent. Moreover, we consider a set of mechanisms available for peers to publish their own items in a channel-like abstraction: in Section 2.3 we showed that most state-of-the-art software in this field already implements these features.

Figure 3.1 depicts the domain model we intend to use for our Rich Metadata dissemination solution.

Publisher It is the role associated to a peer-to-peer user which shares and disseminates content in the overlay network. We make the simplifying assumption that the platform does not support multiple users: there can be at most one publisher for each peer.

Item It represents an audiovisual content being distributed between peers in the peer-to-peer platform.

Channel It is basically a container of Items published (but not necessarily created) by a single Publisher, which is said to own the Channel. In our model a peer can only have a single channel, so that they can be interchangeably identified.
Figure 3.1: Our domain model for Rich Metadata dissemination. A Publisher can have exactly one Channel, which is a container of Items. Rich Metadata instances are associated exactly to one Item and one Channel, which must contain that Item.

**Rich Metadata** Represents a metadata instance, adding valuable and rich information to an Item. Metadata always refers to a single Item in one Channel. In our model only the owner of that Channel can create Rich Metadata for it.

The concepts of publisher and items are pretty straightforward to grasp. Instead we believe that channel and rich metadata types need further explanation. The abstraction of Channels permits peers to create personalised collections of items they want to share with other users in the overlay. By adding entries to their channels, users can create, for instance, selections of content and they can group audiovisual data on the basis of semantic relations. For example, a broadcaster could use its own channel to group and spread episodes of a TV Series in a single place, so that other users know where to find them and where to look for new episodes as they come out.

Channels also give the opportunity to create a coarse grained mechanism of content filtering based on quality: a “good” publisher will tend to insert in his channel only high quality and non-fake items, in order to improve his reputation within the network of peers. This way, “good” channels become an implicit source of “good” content for other peers, while data published in “bad” channels will tend to be ignored.

In the model, an instance of Rich Metadata is uniquely associated to an Item and a Channel with the constraint that the latter must contain the given Item: in other words, a single metadata instance is tightly linked to a particular Item within a single channel. Moreover, we allow only the Publisher owning the Channel to add Rich Metadata instances for it. By preventing peers from posting metadata for arbitrary items or for items in arbitrary channels, we aim to achieve several benefits:

1. The responsibility for the published metadata is tied to a single peer and it is clearly visible by other peers.
2. As a consequence of point 1, we believe that a user is given incentives to publish high-quality data as an implicit way to improve his reputation.

3. We restrict to the scope of a single channel the effects that a counter-productive user willingly publishing low-quality, erroneous, or even spam metadata can have.

### 3.3 Failure Model

Our solution is thought to be used in real peer-to-peer deployment scenarios; hence, we need to assume a realistic failure model.

In this model peers can — and most probably will — fail in every possible way: a peer can crash in any moment of its operation, suddenly stopping. Byzantine failures are admitted, with peers showing arbitrary and unexpected behaviour: this includes also the possibility that some of the nodes perform intentional and malicious adversarial actions.

We assume the network channel to be unreliable: messages can be lost, corrupted or overheard by third parties, and the network connection of one peer can be intermittent or it can die.

Despite all these kinds of failures the overall functionality of the system should be kept for any correct peer.

### 3.4 Problem Statement

In Section 2.2.1 it has been shown that Rich Metadata can assume a large variety of types and representations. Table 3.1 reports some examples of metadata content sizes that were sampled from the private community DDUniverse.

<table>
<thead>
<tr>
<th>Type</th>
<th>Size</th>
</tr>
</thead>
<tbody>
<tr>
<td>Subtitles for a single language</td>
<td>4.78 KB</td>
</tr>
<tr>
<td>Thumbnail</td>
<td>2.70 MB</td>
</tr>
<tr>
<td>Low Quality Preview</td>
<td>26.0 MB</td>
</tr>
<tr>
<td>HD Preview</td>
<td>136 MB</td>
</tr>
<tr>
<td>Textual Metadata$^2$</td>
<td>1.28 KB</td>
</tr>
</tbody>
</table>

Table 3.1: Example of the size of common types of metadata

$^2$The textual metadata size has been calculated using the following approximate estimate: three 100-characters fields respectively for title, director and genre, one 4-characters field for year, and one 1000-characters field for plot description. The characters are assumed to be encoded in UTF-8, using only US-ASCII characters (therefore 1 byte per character).
The table shows as some of them, for instance the video file containing a 1080p movie trailer, can have a significantly large size (in the order of hundreds of megabytes). It is recognisable that pushing this data to all the peers in the network would create an unbearable and unnecessary overhead, since not every peer will be interested in everything.

Therefore we think it is advisable to separate in two distinct parts the diffusion of metadata availability information and the actual retrieval of metadata content: with the first the publisher announces that certain instances of metadata are available, while during the second part the content is pulled by interested peers only when they need it.

Considering also the activities supporting a user in the publication process, three distinct parts are recognisable in our metadata dissemination process:

- **Metadata ingestion.** It identifies the process, and the related mechanisms, that allow a publisher to enrich one digital item with some metadata, and to make these metadata available to other users. During the ingestion process the publisher should indicate to the system the location of the physical files representing metadata, which we assume to be local, and possibly any other necessary descriptive information.

- **Metadata announcement.** It is the process and the related set of techniques used in the system in order to inform users of the existence and availability of metadata for some item. It enables them to perform discovery of metadata. In this stage announcement messages are sent to nodes in the overlay. These messages should be able to deliver to the receiver the necessary information about the nature of the published metadata, such as their type or their size, and should also include sufficient information to allow the retrieval of the actual meta-content.

- **Metadata retrieval.** It is the process, and the related techniques, through which any user can get the physical representation of metadata instances, e.g. the actual files. There could be multiple ways to perform the retrieval: for instance, a peer could use any existing peer-to-peer file protocol to fetch the content from multiple sources in parallel.

It must be noted that the three phases are not required to be strictly sequential; however there is a causal relation in the succession of the three phases in the life cycle of a single metadata content: in fact, announcement cannot start if the content has not been ingested, and of course the content cannot be retrieved if an announcement for it has not been received.

Figure 3.2 shows a schematic description of the three phases of dissemination: reflecting the choices that we made in our design — that will be thoroughly explained in the remainder of this and the following Chapter — it shows a portion of the overlay network in which a user ingests a metadata content at its peer $P$. Curved dashed lines represent metadata announcements that are pushed to other
Figure 3.2: General dissemination scheme. The user publishes some metadata at P and the peer starts to push announcements to other peers; in the example it is helped by peer A₃. Peer A₂ retrieves the actual meta-content.

peers, while the thick double edged line represents the exchange of the actual metadata content that, in the example, is retrieved by A₂ via a direct transfer from P. The reader may also notice that A₃ contributes to the dissemination of the announcement: we will use such a mechanism to speed up the dissemination of high-quality metadata, as it will be described in Chapter 4.

### 3.5 An Architecture for gossip-based dissemination

Given the analysis made in the previous Section, in this one we propose a general architecture for peer-to-peer rich metadata dissemination.

The key idea of our solution is to use a gossip (or epidemic) protocol for Rich Metadata announcement dissemination, and to rely on an external P2P content distribution mechanism to implement the data retrieval phase.

Through the use of gossip, announcement messages are distributed in an efficient way through the Overlay despite peers regularly joining and leaving the network either purposefully or for unexpected failures. Also, the bandwidth needed to disseminate announcement is not taken from the publisher only, but it is spread between peers participating in the gossip protocol.
The following subsection will summarise the common characteristics of epidemic protocols, while in Section 3.5.2 and Section 3.5.3 the design of our dissemination system will be presented.

3.5.1 Gossip protocols

Gossip (or epidemic) protocols are probabilistic protocols for disseminating information. Recently, they have become widely deployed in large scale distributed applications due to their ability to reliably pass data among a large set of interconnected nodes even if it continuously changes, or if the underlying network suffers from slow or broken links.

Nodes perform a simple set of operations periodically and are not aware of the state of the entire system and thus solely act based on local knowledge [17]. However, in a probabilistic sense, the system as a whole manages to converge to a desired global state in relatively short time, typically logarithmic (e.g. in our scenario, the desired state is the majority of peers having received the disseminated information). In many cases gossip protocols might even display global properties that were not thought of in the design phase. By definition, these properties are emergent. Moreover, the system achieves very high levels of robustness to benign failures, thanks to the replication of nodes’ state within their neighbourhood, making gossip-based algorithms very attractive in scenarios characterised by high churn rates and unstable nodes.

**Algorithm 1** Prototypical Gossip: Active Thread

```plaintext
while True do
    wait(Δ)
    p ← selectPeer()
    send(p, my_state)
    state_p ← receive(p)
    my_state ← update_state(state_p)
end while
```

**Algorithm 2** Prototypical Gossip: Passive Thread

```plaintext
while True do
    q, state_q ← receive()
    send(q, my_state)
    my_state ← update_state(state_q)
end while
```

The framework depicted in Algorithms 1 and 2 shows a prototypical gossip algorithm: periodically, a peer selects another peer p using a given peer selection
function, they exchange their states and finally they merge the state they received with their own local state.

The choice of the gossip interval ($\Delta$) and of the peer selection function is the most critical part in the design of a gossip algorithm.

The $\Delta$ parameter determines the frequency peer exchange their state. To choose this value, a designer must consider how fast the nodes’ states change: ideally the dissemination should be sufficiently fast so that every node in the network receives one’s state before it changes again. On the one hand, a small $\Delta$ (high frequency) potentially speeds up the dissemination process, at the expense of a higher and unnecessary bandwidth usage: in fact, it is possible that with high frequencies some peer receives duplicate messages for a state which was already received. On the other hand, long gossip periods tend to slow down the diffusion process, which is not desirable especially for systems where nodes’ states quickly change.

The peer selection functions selects, for each gossip round, what is the destination of a message. In an ideal world, any given node should exchange information with peers that are selected following a uniform random sample of all nodes currently in the system: this way the data would be disseminated homogeneously in the whole network, with the side effect of minimising the probability to send duplicates. In practice this is not feasible for large and highly dynamic networks, since it would need every node to keep a complete and always up to date membership table.

Jelasity et al. [37] propose a general peer sampling service, which is itself based on a gossip paradigm: each node maintains a relatively small membership table that provides a partial view on the network, and refreshes this table periodically through gossips. Using such a service, a peer selection function would randomly select a peer from a so created small candidate list, thus avoiding the need to manage a more complex view of the overlay.

### 3.5.2 Structure

The structural organisation of Metis is shown in Figure 3.3 in terms of its macro components.

At the bottom of our architecture lays the Peer-to-Peer Overlay, which implements the basic communication functionalities and the abstraction we discussed in Section 3.1. We summarise them again here for reader’s convenience:

- **Unique Item identification throughout the Overlay.**
- **Persistent and secure identifiers for peers.**
- **Message routing based on peers identifiers.**
- **Message integrity and authenticity verifiability.**
On top of this communication layer there is the *Gossip Engine* component. This module implements an epidemic protocol similar to the one just described in Section 3.5.1: its role is to implement an appropriate *peer sampling service* and the related *peer selection function*. Furthermore, the component has the role to periodically and actively start the gossip rounds, by choosing a destination and, through the Overlay services, send a message to it. We do not demand to this component the task to specify the actual contents of the messages it sends: every time a gossip round is triggered the gossip engine queries external modules for the information to be exchanged. We will not focus here on the specific gossip parameter used by this module and on its peculiar characteristics: they will be discussed in the next Chapter, where we will explain our decision to use an external service to implement gossip functionalities.

One of the modules queried by the Gossip Engine for data to gossip is the *Rich Metadata Gossip* module, which has the responsibility to create rich metadata announcement messages to be disseminated and to handle incoming announcements.

All the information about Rich Metadata is persisted through the *Rich Metadata Database*, which acts as a wrapper around the persistent storage (e.g. a relational database) and provides methods to save new data and to query the available local knowledge.

The *Rich Metadata Fetcher*, instead, handles the retrieval of the actual metadata content. Given the identifier of a rich metadata instance, which is delivered by gossip announcements, it tries to retrieve the actual contents from any possible source in the Overlay.

The system’s API is exposed by a facade component, the *Rich Metadata User Services*, which provides an interface allowing users to ingest new rich metadata, explore what is available and issue explicit meta-content retrieval actions. Whenever we use the term *user* with reference to our dissemination system we do not necessarily intend to indicate a human directly interacting with it, but also a software client using our services: actually this will likely be the most common
situation; for instance what we call user may be the software component realising
the peer-to-peer client GUI.

All the operations of the system are coordinated by the Rich Metadata Man-
ager, which accepts commands from the User Services, uses the availability data
received from announcement messages to instruct the RMD Fetcher, and starts the
dissemination process for newly ingested content.

### 3.5.3 Interactions

To better clarify the general behaviour of the structure we just presented, we will
describe here the main interactions between those macro components: the attention
will be focused on the use cases realising the main functionalities of the system
under design, which are:

- Rich Metadata ingestion.
- Gossip based announcement dissemination.
- Rich Metadata retrieval.

Figure 3.4 represents an UML sequence diagram showing how the system com-
ponents operate in order to add a new Rich Metadata instance. The process is
started by an external client issuing a command on the RMD User Service module,
which immediately delegates the operation to the RMD Manager. The latter, first
of all, verifies that the meta-content to ingest is valid: the steps involved in the veri-
fication process are highly dependent on the type of rich metadata; for instance, in
the case of subtitles it may check that they are correctly serialised according to a given representation format, such as SubRip [78], or that they have a proper character encoding. Finally, information about the new metadata is stored in the RMD DB, and the announcement process is triggered.

The dissemination process is controlled by the Gossip Engine. According to its internal parameters, such as the gossip round period, time to time it selects a peer to gossip with. At this point, before exchanging the actual data, it delegates to the Rich Metadata Gossip module the task to compose the announcement message. It does so by querying the RMD DB for metadata it has published for its channel; furthermore, announcement message may forward information about metadata published by others, that the sending peer has overheard from other gossips. The possible policies for the selection of which part of other peers’ metadata to forward can be different: for instance, one could forward only recently received data, or only data received from some kind of “trusted” peers. This choice is very relevant, because it significantly influences the diffusion speed of different data in the Overlay. We will not cover this aspect here: in Chapter 4 we will describe how we used this mechanism to promote the diffusion of “high quality” metadata and to slow down the diffusion of fake or bad metadata. Figures 3.5 and 3.6 respectively show how the described interactions are realised for both the self-initiated

Figure 3.5: When the Gossip Engine needs to send an outgoing message, it demands to the RMD Gossip component the task to create the rich metadata announcement.
and externally-initiated announcement management.

Rich Metadata retrieval is started by an explicit user command, that is dispatched to the Rich Metadata Manager, which has the necessary knowledge to possibly start a download. In fact, as shown in Figure 3.7, it verifies that the received request is valid and, on the basis of the announcement messages received by the RMD Gossip module, and depending from the peer-to-peer content download protocol, it may also compute the available sources from which to get the meta-content. A proper fetch instruction, which also depends on the adopted content retrieval method, is then issued to the Fetcher Module, which, in turn, will notify back the Manager once the content is retrieved.
Figure 3.7: The retrieval of actual Metadata contents is performed by the RMD Fetcher component.
Summary

In this Chapter we have laid down the foundations on which we have designed and built Metis, the Peer-To-Peer Rich Metadata dissemination solution developed in this Thesis.

We assumed to work on top of a middleware implementing a basic peer-to-peer Overlay abstraction, providing identification mechanisms for both peers and items and message routing based on peers’ identifiers. Our system shall respect a set of simple requirements, and expose an API to allow users to publish and search for rich metadata instances. Among the functional parameters we would like to optimise, there are the dissemination speed and the bandwidth overhead used to implement it; security and full decentralisation are also central goals to reach.

Considering these requirements, we found convenient to split the problem in three separate phases of ingestion, announcement and retrieval, each of which shows different characteristics and challenges. We sketched a general architecture to solve the problem, based on epidemic diffusion of announcement messages and peer-to-peer retrieval of metadata content.

In the next Chapter this architecture will be further detailed and adapted to a particular peer-to-peer client which provides most of the services and abstractions we have described until now. Also, we will show how this architecture has been implemented to fit to a particular use case, which is subtitles dissemination.
Chapter 4

Design

In the previous Chapter we introduced the general architecture of Metis, providing the basic design principles for a peer-to-peer metadata dissemination solution. In this Chapter we will adapt those ideas to a more concrete use case, and we will specify in much deeper detail the architectural aspect that we presented from an high-level perspective in the previous Chapter.

Our solution will make use of the services implemented by an existing peer-to-peer platform which provides most of the abstractions we assumed to be available. This will permit us to invest our full effort on the design of effective dissemination protocols, since we can devolve many of the Overlay responsibilities upon a thoroughly tested and deployed framework.

In Section 4.1 Tribler [58] — an open source peer-to-peer platform developed by the Delft University of Technology, which we already introduced in Section 1.4 — will be discussed in deeper details especially for what concerns those parts that will directly influence our work. In particular, the ChannelCast protocol will be dissected since it realises the abstraction of Channels we widely use in the system we are designing.

Starting from Section 4.2 and for the remainder of the chapter, we will illustrate how the architecture we described naturally adapts to Tribler with little modifications, and we will present an implementation of that model which realises subtitles dissemination within the Tribler Overlay Network. Specifically, Section 4.4 will detail how the dissemination protocols works for the three phases of ingestion, announcement and retrieval, while Section 4.5 will refine the previously presented architecture adapting it to our specific use case.

4.1 Platform description

In this Section we will briefly introduce the architecture of Tribler in general and of the two gossip protocols BuddyCast and ChannelCast in particular. This is not intended to be a complete description: only the features that are necessary to explain our work will be presented.
4.1.1 Tribler

We already presented some of the main ideas behind the Tribler platform in Section 1.4. Here we will introduce the software architecture that realises those ideas, covering the aspects that are more closely related to the design of the dissemination solution we are discussing in this Thesis.

A simplified view of Tribler’s internals is shown in Figure 4.1: in the diagram blocks represent the main software components and arrows their logical dependencies.

**BitTorrent Socket Layer.** It is responsible of the management of the low level BitTorrent connections to and from other peers both for the Overlay Swarm and for other BitTorrent swarms the node is in.

**Network Thread Manager.** It is responsible of scheduling and executing all the network related tasks, such as reading and writing messages from the sockets in the BitTorrent Socket Layer. All the operations are performed by a single thread, known as the Network Thread.
Secure Overlay. It implements and manages Tribler’s peer-to-peer Overlay. It is responsible to correctly route message to peers in the overlay swarm, and to dispatch the received overlay messages to the other appropriate software components.

Overlay Apps. It is a container and manager of the so called Overlay Applications. An overlay application is a service running in the platform which uses the overlay swarm to interact with other peers and implement some desired functionality. Examples of currently implemented applications are BuddyCast and ChannelCast, which will be soon described, but also RemoteSearch and TorrentCollecting; the former is an extension allowing to delegate search queries to nodes in peers’ neighbourhoods, while the latter provides a service which collects torrents’ metadata (the .torrent files) from encountered peers (for details see [7]). Every overlay application is run by a single thread called Overlay Thread, whose scheduling is demanded to the OverlayThreading Bridge.

MegaCaches They store locally to a peer all the context information received through interactions with other nodes that is relevant to it based on its interests and tastes, such as information about a peer’s taste buddies (see Section 4.1.2), identity and contact data about encountered peers in general, preference lists on which recommendation algorithms are run, and information about Channels (see Section 4.1.3).

BuddyCast is the basic Overlay Application in Tribler. It implements an epidemic protocol [59] used to form a second level semantic overlay which in turn is used for distributed content search (similar to the one described in [76]). Section 4.1.2 will describe its characteristics in more detail.

ChannelCast is a gossip-like algorithm running on top of the peer-selection process performed by BuddyCast. Its purpose is to implement a distributed moderation system through the abstraction of Channels. Since we will build on top of this protocol, we will provide a more accurate description in Section 4.1.3.

4.1.2 BuddyCast

BuddyCast is the name given to the core Tribler Overlay Application and to the epidemic protocol it implements. In Tribler BuddyCast is principally used to support three features:

1. Peer discovery.
2. Content discovery.
3. Formation of a Semantic Overlay [76].
Furthermore, BuddyCast offers to other overlay applications, such as ChannelCast, the services of a Gossip Engine like the one we described in Section 3.5: applications can hook in the BuddyCast core to be notified of outgoing and incoming gossips; this way, they can add their own application data to the exchanged messages and process them when they come from external sources.

For our discussion, we find convenient to consider BuddyCast as if it was logically divided in two different parts, the BuddyCast Gossip Engine and the BuddyCast Data Exchange: the former takes care of all the application independent gossip mechanisms including the actual implementation of the gossip framework itself, while the latter implements peer and content discovery on top of it. In the real implementation the two parts are strictly interrelated and, in fact, there is not an actual distinction between them in the code: that is because — as we will see — the gossip engine realises its peer selection function actively using information produced by the data exchange part, which takes the role of the peer sampling service that has been introduced in Section 3.5.1.

**BuddyCast Data Exchange**

The BuddyCast Data Exchange logical component can be further decomposed in two parts with the two following different functions.

1. Implement a peer sampling service for the Gossip Engine.

2. Provide distributed content discovery.

Figure 4.2 shows the content of the messages exchanged by peers in each BuddyCast rounds. The first information that is shown at the top of the figure is the preference list of the sending peer; this list contains at most 50 infohashes identifying the same number of contents that were recently downloaded. To support peer discovery the contact information of (at most) 10 taste buddies and 10 random peers is

---

**Figure 4.2:** A BuddyCast message is made out of own and taste buddies content preferences, and of the contact information of taste buddies and some random peers.
included; for the taste buddies a short content preferences list of at most 10 entries is sent as well.

Upon the reception of a BuddyCast gossip message, a client processes the data as follows. The preferences information is fed into an ad-hoc recommendation engine which computes the similarity between the receiving peer and every other known peer: the ranked list of the most similar peers is used to keep track of a set of *Taste Buddies*, which currently corresponds to the top N peers in the list.

The peers contact information is used instead for the peer discovery and Semantic Overlay creation processes; addresses of peers and the timestamp of the last moment they were seen online are stored in a local database and TCP connections are kept open with a limited number of taste buddies and random peers: these connections constitute the Semantic Overlay, and they are used, for instance, by the RemoteQuery module to route distributed queries.

**BuddyCast Gossip Engine**

BuddyCast provides also a Gossip Engine regulating all the parameters for the epidemic protocol. As already stated, the gossip service is used by the just described BuddyCast data exchange part, but it can be also used by other Overlay Applications to exchange their information through the Overlay in a scalable and robust way. ChannelCast is one of such applications.

Following the general gossip framework introduced in Section 3.5.1, the operation of Tribler’s gossip engine is periodic and round based. During each round a node actively selects another peer and tries to connect to it to exchange some data through gossip messages. The length of the interval between each round is not fixed and it depends on the local peer’s uptime: for the first 30 minutes since a client becomes online the gossip interval is set to 5 seconds, after 24 hours of uptime it is set to 60 seconds, while between these two intervals it is set to 15 seconds. The rationale behind this choice is that new peers need to “refresh” their information as quick as possible, while elder clients are in a more stable condition and therefore need less frequent updates.

Tribler implements a peer selection function that uses the peer sampling service offered by the BuddyCast Data Exchange layer: from all the peers that are known from exchanged BuddyCast messages, a shortlist of candidates is created: this list comprises five peers whose last_seen timestamp is bigger then everybody’s else (i.e. it includes the most recently seen peers). One peer is selected from it: with a probability $p$ the selected peer will be a taste buddy, and it will be a random one with a chance of $1 - p$. In the current implementation $p$ is set to 0.5. Once a peer is selected it is also inserted into a blacklist where it will stay for 4 hours: this prevents that it will be chosen again too soon.

At this point, the gossip engine starts to build the message to send to it: the first information to be inserted is created by the BuddyCast Data Exchange module, followed by all the other modules that are properly registered on the engine.
4.1.3 ChannelCast

In this Section we will briefly introduce the ChannelCast protocol as it is implemented in Tribler 5.2 and described in the technical specification of the NextShare Platform M24 [6].

ChannelCast is a gossip-like protocol implementing the abstraction of channels in Tribler. As in our domain model (see Section 3.2) every peer has one Channel, using which the user can publish and spread torrents, which are called “moderations” of the channel. Moderations are disseminated to other peers enabling them to know the kind of content that is being spread in the network.

Peers can subscribe to a channel, indicating a positive vote for it; nodes only gossip on moderations from those channels they have subscribed. Users may also disapprove a channel by indicating a negative vote.

ChannelCast uses the BuddyCast Gossip Engine to implement its gossip infrastructure: whenever BuddyCast selects a peer to gossip with, ChannelCast also creates and sends a ChannelCast message to it. A ChannelCast message consists of a bunch of recent and random moderations from only the own and subscribed channels: the current implementation includes a total of 50 moderations; 25 of them are own moderations (15 recent and 10 random), the remainder is composed of moderations sampled from subscribed channels (15 recent and 10 random).

Thanks to this mechanism, highly subscribed channels will tend to spread their moderations more quickly than channels that are not. If no other node subscribes to a channel, then the only way its moderations can spread is through direct contact with other nodes. Nodes that disapprove a channel remove all the associated moderations from their local database and refuse any new moderations from that channel.
Figure 4.3 shows the format of ChannelCast messages. For the complete ChannelCast protocol specification consult Appendix A.

4.2 Subtitles Dissemination

We instantiated the model presented in Chapter 3 as a Metadata Dissemination system integrated in Tribler, refining and implementing the general architecture.

As we have seen Tribler is a peer-to-peer platform offering a complete double-layered unstructured Overlay network, a reusable gossip engine, and the abstraction of items and channels similar to the one we adopted in our domain model. It offers a set of services which are able to satisfy most of the assumptions we made for the design of our dissemination system, such as secure and permanent identifiers and secure message routing. For these reasons, it represents a natural choice for our work.

As a first step we decided to focus our work to support only one simple type of rich metadata: subtitles in different languages for video content. This choice allowed us to concentrate our efforts on the dissemination issues without dealing with the complexity deriving from supporting multiple content types. Moreover this simplification permitted us to achieve a complete working implementation in a limited time, and to accurately evaluate and test it in order to include it in a public release of the Tribler client for further evaluation by actual end-users.

Subtitles were picked among other possible metadata types because we found them to be the prominent resource available in private communities but not in peer-to-peer platforms. Furthermore subtitles allow a video content in a single language to be consumed in different localisations: in file-sharing this is normally achieved by releasing different versions of the file for different languages; in most of the modern peer-to-peer protocols, where download performances are strictly related to the number of available replicas of the content, this fact has a negative impact since the set of all the possible sources are fragmented between the different locales. With the ability to associate multiple subtitles to a single video, the same file can be used and shared by users speaking different languages, hence increasing its number of possible download sources.

4.2.1 Domain Model

The general domain model we presented in Section 3.2 can now be refined reflecting the choice to focus on Subtitles dissemination in Tribler.

Being a BitTorrent based systems, Tribler identifies digital items by means of the infohash value of their swarm. Peers — and therefore Channels — are instead uniquely addressable by their PermId.

For each audiovisual content several subtitles can be published, but no more than one for a single language. Figure 4.4 shows how the domain model presented in
Section 3.2 is adapted to the specific use case of subtitles dissemination. It is immediate to recognise how, being associated to a single item within a given channel, a subtitle instance can be uniquely identified by the triple \((item\'s\ infohash, channel\'s\ PermId, subtitle\ language)\).

### 4.3 Design Overview

Our dissemination system has been designed to exploit the services offered by the Tribler platform: in particular the Gossip Engine provided by BuddyCast is used to control the epidemic announcement dissemination for what concern the peer selection procedure and message delivery, and the ChannelCast protocol is used as it provides the fundamental abstraction of Channels.

Basically, our protocol can be considered as an extension of ChannelCast. Using our subsystem, publishers in Tribler are able to associate subtitles to moderations they have already published in their channels.

We extended ChannelCast in order to disseminate subtitles announcements in the network: every moderation entry in a ChannelCast message is augmented with a new field describing the available subtitles. This way peers gossip about the available subtitles with their semantic neighbourhood, and epidemically spread the announcement information.

The design aims to guarantee full backward compatibility: old clients will be able to understand the new ChannelCast messages with no modifications, and new clients will continue to be able to receive and manage messages in the old format.

Given the size of subtitles, which is usually under the hundreds of kilobytes, we decided to distribute them by need, via direct peer-to-peer download from available sources: we implemented a simple and ad-hoc request-response protocol to transfer
their content via two new overlay messages.

After having retrieved the subtitle content, a peer immediately becomes an additional source for it; every peer informs its neighbourhood about the meta-content it has got through a special field in the announcement messages.

We also leverage ChannelCast subscriptions and spam reporting mechanisms to speed up the diffusion of good metadata and to filter out the bad one, and as a mean to increase the availability of newly inserted subtitles in the network: in fact — in our system — subscribers of a channel automatically fetch metadata published in it as they receive the corresponding announcement messages.

In the next Sections we will detail the designed protocol: we will start from a behavioural description in Section 4.4, and then the specific Architecture will be presented in Section 4.5.

4.4 Protocol Description

In this Section a description of the dissemination and retrieval protocol will be given. A complete and more formal specification can be found in Appendix A.

4.4.1 Subtitles Ingestion

To start the ingestion process a publisher must provide a path pointing to a local copy of the subtitles file and the language of the selected subtitle. Furthermore, he has to choose an item previously added to his channel to whom the subtitles are to be associated.

Then, a copy of the subtitles file is created and it is placed in a specific location on the file system called subtitles collecting dir. In this directory all the subtitles locally and remotely collected are kept. To minimise the chance of file name collisions with other collected subtitles, we perform the SHA-1 digest over the string resulting from the concatenation of the items’ infohash, the channel’s identifier and the 3-characters ISO 639-2 [32] code of the subtitles language; we use the hexadecimal representation of that digest as the file name of the subtitles in the subtitles collecting directory. For instance, a possible subtitles file name could be b96f1a2398cece6f558ced8d259fd979d2b087ab.srt.

Information about the published subtitles is immediately stored into a new appropriate section of the local MegaCaches (see Section 4.1.1), and the dissemination process for the newly added metadata is triggered.

4.4.2 Subtitles Announcement

As said before, our system leverages ChannelCast to disseminate information about availability of subtitles within channels. The protocol has been updated and its messages now include information about subtitles for each moderated torrent.
For each moderation entry in a ChannelCast message (recall Figure 4.3), a new field has been added, called rich metadata, whose internal format is shown in Figure 4.5. The semantics of the old fields has not been changed not to affect the compatibility with old clients. Below you can find a description of the rich metadata field structure:

**Description** is a byte string at most 500 bytes long. It is thought to enable publishers to provide a free form textual description for a moderation, although it is currently unused.

**Timestamp** is an integer value indicating the number of seconds elapsed since the Unix epoch at the moment the rich metadata entry was last modified by its publisher.

**Languages Mask** is a 32-bit string. Each slot in the string corresponds to one of the supported subtitles languages. Each bit set to the value ’1’ means that the moderator has published a subtitle in the language corresponding to that bit. For a list of the supported languages consult Appendix A.

**Sha-1 Checksums** is a list of SHA-1 digests. There is one element in the list for each published subtitle, and it represents the hash of the corresponding file. This is included to avoid third parties spreading fake copies of subtitles.

**Signature** is an ECC signature over the above fields, taken with the publisher’s private key. It can be used by receiving peers to check that the message has
not been altered.

**Have Mask** is a 32-bit string, similar to the language mask. It represents the subset of the published subtitles which are available at the node that is currently sending the message. This permits the receiving peers to build a partial map of the physical locations of subtitles content.

The purpose of the entries in the new field is to provide the receiving peer the necessary data both to understand what is the published metadata, and where to retrieve the associated content. Particular attention has been also payed in order to guarantee the integrity and authenticity of the content being spread: in fact the entry is signed with the publisher’s permanent key, and secure digests of the subtitles files being spread are included, so that a malicious peer cannot spread fake subtitles on behalf of the publisher.

Notice also how using bit masks to inform about subtitles availability guarantees a minimum overhead in terms of message size. Nonetheless it represents a non negligible limitation in terms of extensibility of the protocol and adaptability to different types of metadata.

### 4.4.3 Subtitles Retrieval

Subtitles are retrieved by interested peers via a simple request response protocol. A peer willing to get the content of one or more subtitles for an item within a channel can send request messages to peers who announced to have them through ChannelCast. Receivers may directly respond with a message including the content as payload.

Nodes receiving a request for subtitles may respond only in the case they have at least some of the requested subtitles: if they do not, they should discard the request without further processing it.

Peers keep a soft state about requests they have sent: response messages received from peers that were not asked for content are rejected, and pending requests are discarded after a timeout period $\tau$. This way a certain robustness against peers’ failures is achieved: if a peer fails any moment after receiving a request, the originating node will not wait for the answer for an undefined amount of time.

Requests for subtitles can be sent in parallel to multiple sources, if they are known: when a valid reply is received all the other requests for the same subtitle are removed from the node’s state, so that further responses are ignored. The maximum number of peers queried at the same time is a tunable parameter, currently set to 5. This value was determined from empirical considerations, and it will need to be better validated by further experiments.

Two new Overlay Message types have been introduced to implement the protocol. The formats of the messages are shown in Figure 4.6.

A *GET_SUBS* (Figure 4.6a) is sent by peers as a request message:
Figure 4.6: Format of the GET_SUBS (a) and SUBS (b) overlay messages, used by peers to retrieve subtitle content.

Publisher Id is the PermId identifying the channel where the requested subtitle(s) has (have) been published.

Infohash is the infohash of the item (i.e. the torrent) to whom subtitles are attached.

Request Mask is a 32-bit string. Each '1' in the string represents a requested subtitle in the language corresponding to that bit. In a correct message the Request Mask is a valid subset of the languages mask sent during the announcement phase.

SUBS messages (Figure 4.6b) are sent as responses to GET_SUBS requests, in the case one or more requested subtitles are available at the queried node:

Publisher Id is the PermId identifying the channel where the requested subtitle(s) has (have) been published.

Infohash is the infohash of the item (i.e. the torrent) to whom subtitles are attached.

Answer Mask is a 32-bit string. Each '1' in the string represents a subtitle in a given language included in this response message. It is a subset of the request mask sent along with the GET_SUBS request.

Payload is a variable-size list of the UTF-8 encoded subtitles content. The maximum size of this field in our implementation is set to 1 Megabyte.

Figure 4.7 shows an example of the subtitles dissemination and retrieval process in a portion of the overlay network. The figure represents a sample of the network.
Figure 4.7: An example of the subtitles dissemination and retrieval process.
(a) A publisher (green circle) sends announcements to its neighbourhood.
(b) A subscriber (purple circle) and another peer request subtitles to the publisher.
(c) The subscriber forwards the announcements to its neighbourhood.
(d) The publisher goes down and another peer subscribes to its channel.
showing one publisher (green circle), subscribers of his channel (purple circle),
and other peers. The publisher has added two subtitles for an item in its channel,
one in Dutch, one in Italian. Yellow shadows underneath the peers mean that the
announcement for the subtitles has been received. Vertical and horizontal patterns
inside the circles respectively reflect the availability of one of the actual subtitles
contents.

Different subfigures illustrate different parts of the dissemination:

- **Figure 4.7a:** The moderator gossips with its (logical) neighbourhood about
  the availability of the two subtitles. The languages mask and the have mask
  are shown above the arrows with list notation. In this case they coincide
  because the moderator has local copies of both the subtitles files.

- **Figure 4.7b:** As soon as it receives the announcements, the subscriber sends
  request to the moderator for every subtitle content it has. This is done in
  order to create replicas of subtitles, increasing their availability. Another
  peer also sends request to the moderator, but only for the Italian subtitle.

- **Figure 4.7c:** The subscriber starts to spread information about subtitles via
  ChannelCast. Languages mask and have mask are as well equal in its an-
  nouncements, since it also has copies of both the subtitles.

- **Figure 4.7d:** The moderator goes offline; in the meanwhile, the peer who
  previously requested the Italian subtitles has become a subscriber. Therefore,
  it spreads subtitles information via ChannelCast. This time the have mask
  that is sent only has the Italian bit set, since the peer does not have a local
  copy for the Dutch language.

### 4.4.4 Spam reduction and good metadata diffusion speed up

The lack of central authorities makes it difficult, in decentralised P2P platforms, to
control the quality of content ingested in the system: the proliferation of fake and
spam data is notoriously a relevant problem in common peer-to-peer file-sharing
architectures. Of course, these issues can also affect the overall quality of meta-
data distributed across the overlay: malicious peers can voluntarily publish wrong
or misleading descriptions of digital items and try to spread them to as many peers
as possible. A less dangerous, but still significant issue is the detection of low qual-
ity metadata and, at the same time, the speed up of the diffusion of good one: in
centralised environments, like Web communities, this is typically achieved through
the use of ad-hoc comments or rating systems, used by users to express qualitative
opinions about the published items. While those methods showed to be particu-
larly effective, their implementation in peer-to-peer systems is not trivial, and goes
beyond the goals of this Thesis.

Instead, we decided to use a much simpler mechanism to hinder the diffusion
of spam, while facilitating it for high quality metadata: the ChannelCast protocol

56
includes a mechanism, known as VoteCast, which allows users to express a ternary preference about a channel, i.e., subscribed, marked as spam and no preference; we leverage this pre-existing feature in order to accomplish our goal. In fact, metadata about channels that an user reported as spam are immediately removed from the MegaCache, and are no further spread by the gossip algorithm, and the data originated from that peer will no longer be accepted: if many peers report a channel as spam, the diffusion of its metadata will quickly tend to stop. On the other side, as already said, peers forward metadata announcements about channels they are subscribed to and they don’t for channels for which they have expressed no preference: this will make the diffusion of metadata of channels that have been reported as “good” by many peers much more faster then channels with few or no preferences. Moreover, as we specified in the previous Section, subscribers of a channel automatically retrieve its metadata content as soon as they receive announcements for it: this way an higher degree of replication is achieved for what is generally considered good metadata. As a consequence we believe that the overall system performance is improved, since there will be more sources capable of serving requests for highly requested metadata.

4.5 System Architecture

The Subtitles Exchange Subsystem has been implemented as a new Overlay Application using the services of the platform for what concerns the gossip engine and the communication services.

In this Section we want to describe the software components that realise the behaviour just described, in terms of their individual responsibilities and their interactions.

4.5.1 Structure

Figure 4.8 shows how our architectural model was adapted to fit in the Tribler platform: very few changes had to be performed, most of which have been done in order to adequate to Tribler’s gossip system and threading model.

The Figure depicts the main software components of the system, with emphasis on the logic dependencies between them and pre-existing Tribler modules. Newly designed parts are depicted as black blocks, while old ones are shown in a lighter shade of grey; dashed arrows represent logic usage dependencies, meaning that the source component needs the services provided by the destination components to accomplish its role.

The roles of the Rich Metadata components is practically unchanged from what we described in the previous Chapter. In the following paragraphs we will reconsider them one by one to see how they interact with the surrounding environment.

Rich Metadata Gossip has the responsibility to implement and control the gossip-based dissemination process; it is built on the ChannelCast Core, and its role
Figure 4.8: Software components of the Subtitles Exchange Subsystem, and their logic dependencies. Pre-existing Tribler components are shown in a lighter shade of grey.

is to intercept outgoing and incoming ChannelCast messages, convert them to the new format and deal with the rich metadata entries.

The Rich Metadata Database was added to the set of the MegaCache components: it is a wrapper around the local SQLite database, and it stores and manages all the persistence related issues regarding rich metadata. It provides functionalities to store announcement information and to search and filter available metadata for a given item or channel.

The retrieval of actual rich meta-contents is demanded to the Rich Metadata Fetcher. Given the identifier of a rich metadata instance — which is in the case of subtitles the channel id, item's infohash and subtitles language triple — it tries to remotely retrieve the actual contents. In our implementation, it implements the soft-state subtitles retrieval protocol described in Section 4.4.3 to do so.

Other Overlay Applications or other Tribler components, such as the GUI, may interact with the Subtitles Subsystem using an API which is exposed by the Rich Metadata User Services component. The full API reference and a simple user manual are provided respectively in Appendix B and Appendix ??.

Finally, the Rich Metadata Manager acts as the subsystem controller, dispatching commands on user behalf and co-ordinating and synchronising the operation of the gossip module and the content retrieval process.

Figure 4.9 shows how these high-level components are further decomposed in finer-grained modules that are mapped “one-to-one” to Python modules in our implementation. Their features will be extensively clarified in the following Section, when their principal interactions will be presented.
Figure 4.9: Decomposition of the high-level components of the Subtitles Subsystem with reciprocal dependencies.
4.5.2 Interactions

As we did in the previous Chapter, we will analyse the main interactions between the implemented modules especially focusing on what happens in each of the three phases of the metadata dissemination process.

Figure 4.10 shows how the ingestion of new subtitles is handled; as first, the SubtitlesSupport module checks that the item for whom the subtitle instance was asked to be added is contained in the publisher’s channel, and then demands to the SubtitlesHandler to verify and possibly copy the content in the subtitles collecting dir. Finally the information of the successful publication is stored via the MetadataDBHandler.

Announcement messages are included into ChannelCast messages by a mechanism based on interception of outgoing and incoming messages, as shown respectively in Figure 4.11 and Figure 4.12. Whenever it is time to gossip, the BuddyCast Gossip Engine calls the ChannelCast Core module to make it create a ChannelCast Message. Before returning the control to the Gossip Engine, and after the standard ChannelCast message has been created, the RichMetadataInterceptor module — part of the Rich Metadata Gossip component — intercepts the outgoing message, and adds the message fields described in Section 4.4.2.

Mirroring this procedure, whenever a ChannelCast gossip comes in, the RichMetadata interceptor intervenes to extract the announcement information and process it before passing the cleared message to the ChannelCastCore. At this point, the have mask included in the data is also translated and passed to the PeersHaveManager which uses that information to maintain a partial view of where contents
Figure 4.11: Outgoing ChannelCast message are enriched with subtitles availability information before being sent.

Figure 4.12: Incoming ChannelCast messages are intercepted, and the rich metadata information stored in the Database.
Figure 4.13: The SubtitlesSupport facade demands the task of preparing a subtitles request to the OverlayThread. Possible destinations are given by the PeersHaveManager, and multiple requests are sent to them.

are stored in the overlay.

Finally, the retrieval protocol is implemented by the SubsMsgHandler module and coordinated by both the SubtitlesHandler and PeersHaveManager.

We report here the two scenarios of an outgoing request and an incoming response, depicted as sequence diagrams respectively in Figure 4.13 and Figure 4.14.

A subtitles request (Figure 4.13) is triggered by the invocation of a command on the SubtitlesSupport facade by an external client. A decoupling of the request handling process and the relative message send action is achieved by having two different threads perform the tasks: the Overlay Thread scheduler — i.e. the OverlayThreadingBridge — is responsible for creating and scheduling the task to be executed by the Network Thread. Later, when the action is executed, the PeersHaveManager is queried to return a list of possible peers having the desired subtitles: at most \( n \) of them (currently \( n = 5 \)) are chosen as the destinations for a request, which is built by the cooperating SubtitlesHandler and SubtitlesMsgHandler. Finally, the actual send operation is invoked on the OverlayBridge, which schedules it to be executed by the NetworkThread. Notice that an external client requesting subtitles can specify, at the moment it invokes the action on the SubtitlesSupport module, a callback method to be eventually called when the results are received: a reference to this function is saved by the SubtitlesHandler to be used later.

When a request for subtitles is remotely received it is immediately passed by the OverlayApps module to all the handlers registered for the appropriate message type. In the case of a GET_SUBS, the message is dispatched to the SubsMsgHandler which verifies its formal correctness and decodes its content. The request is then delivered to the SubtitlesHandler which tries — if possible — to satisfy the request: this happens if any of the requested subtitles is locally available. Only
in such a case, a response is built and sent back to the requester encapsulated in a SUBS message, which is scheduled and sent by the NetworkThread. Finally, when a response containing UTF-8 encoded subtitle content is received, the process shown in Figure 4.14 is executed. Again, the message is passed to the SubtitlesMsgHandler, which decodes it and checks whether it is formally correct. The decoded information is then elaborated by the SubtitlesHandler, which performs another control on the validity of the content, saves it on the disk in the subtitles collecting dir, and stores the availability data in the rich metadata database. Finally, the callbacks registered by interested listeners are executed.

4.6 Metis possible limitations and extensions

Before moving to an experimental evaluation of the designed system, we believe it is worth to point out some of its limitations that we recognised with an a-priori analysis of our protocol and architecture for peer-to-peer subtitles dissemination. We will illustrate them here trying to clarify that, in our specific use-case, they constitute a minor problem; however, we will also propose possible solutions to be validated and implemented in the context of possible future works.

4.6.1 Direct peer-to-peer exchange of subtitles

The first source of possible weaknesses is the simple request-response peer to peer protocol we decided to adopt to implement content retrieval. With this approach the entire metadata content is retrieved via single Overlay message from a single
source. This is very different with respect to what happens in most of the modern peer-to-peer file sharing systems, including BitTorrent: in fact usually the file to download is split in multiple chunks and every single chunk is downloaded independently and possibly from multiple sources. This mechanisms has at least four main benefits:

1. The download can be faster since the upload bandwidth of several peers can be exploited.
2. There is no strong dependency on a single peer: if a node disappears during a transfer, the download can be continued from other peers.
3. If a chunk is corrupted, only that chunk needs to be re-downloaded.
4. To become new possible download sources, peers need not to have the complete file but only a complete single chunk.

With our approach no multi-part download scheme is exploited. This means, for instance, that if a received meta-content does not pass the integrity check it needs to be entirely retrieved. However, the contents in our use cases are subtitles for digital video, whose size is within the order of tens of Kilobytes, which is a very small size even if compared with the chunks size of most of the modern P2P multipart-based file-sharing protocols: for example, BitTorrent divides the file in pieces that are commonly 256 or 512 Kbytes long, and each of these pieces is further split in blocks of no more then 123 KBytes (usually 32) to enable finer grained transfers. Clearly the big majority of the payloads of our protocol messages will be smaller then single BitTorrent blocks, thus justifying our choice.

Another implicit benefit of BitTorrent-like protocols is that they include a sophisticated and automatic mechanism to manage new sources as they get copies of the files they download. This is not true in our case, where only subscribers share the subtitles meta-content. Moreover, we do not have, unlike BitTorrent, any centralised management of possible sources: as a consequence only a partial view can be built by peers having the announcement messages they receive as the only source of information.

However, complex peer-to-peer file sharing protocols suffer from big latencies when dealing with small files: they are due to the fact that the overhead necessary to set up the protocol (such as finding available sources) is usually several orders of magnitudes bigger then the time required to perform the actual transfer.

For further extensions of our work, which may include the support of more complex metadata types, it will be desirable to study and use a more refined retrieval protocol especially for what concerns big sized meta-contents. A hybrid solution could also be considered where content whose size is under a given threshold is downloaded via a simpler protocol like the one we proposed here, while longer data are fetched via a BitTorrent-like protocol. Another possible solution could be to pack all the small sized meta-content relative to a given item in a single archive, and then distribute it via multipart transfer: this way the latency cost due to the
protocol management time would be alleviated. Pre-fetching approaches could also be evaluated in order to mitigate the problem of latency times: a peer could automatically start to fetch meta-content as soon as announcements are received, and possibly delete it later in case it is not needed.

4.6.2 Duplicate Announcements Overhead

An intrinsic problem of gossip based protocols is that of duplicate information being spread in the network: as for real gossips it is possible that a peer receives the same information two or more times from different sources, or even from the same (depending on the peer selection function).

In BuddyCast every node keeps a blacklist of peers which have been recently selected for epidemic information exchange: in this list entries are kept for a four hours interval, avoiding them to be selected again too soon. A possible problem is that after this period they will be selected again with high probability, especially if they are part of the Semantic neighbourhood of the peer.

In our system this fact entails that nodes may receive many times the same announcement for some rich metadata entry, causing a useless overhead. Anyhow, we designed our gossip messages to be very small sized: our bitmask based system, albeit limiting the possible number of supported languages, minimises the information to be transferred, causing the bandwidth overhead for duplicates announcements to be limited.

However, an expansion in our dissemination system, especially if we want to support several metadata types, could likely make the size of announcements grow considerably. In such a case the duplicates overhead could become not acceptable any more. A possible solution, that has also been used in other similar contexts, could be to introduce another step of indirection in the protocol: similarly to what is done by the MobEyes middleware [43] we presented in Section 2.3.2, our system could make use of Bloom Filters to reduce the size of the announcement messages. The idea could be to have the gossip messages of a peer consist of a Bloom filter representing the rich-metadata announcement information it already has; the receiving peer — then — would compare this set with the set of metadata he knows about and only send the missing information to the gossip initiator; we have explored this possibility, and we will show the results in the end of Chapter 6.

4.6.3 Infection models for Epidemic protocols

In Section 3.5.1, where gossip protocols have been introduced, we emphasised that two of the most important parameters regulating them are the gossip interval and the peer selection function. In that context we did not mention another feature that yet is fundamental to understand what are the dynamics of the information dissemination: we call it the information retention time $\gamma$. This parameter controls for how long an information received via a gossip message is considered valid and
kept in the system: long retention times entail a better spread of the information in the system, but may cause nodes to deal with old and stale data.

Both Buddycast and ChannelCast protocols are designed to use an infinite information retention time, meaning that, once data is received through gossip, they consider it valid forever. We have not found any practical reason for this choice, therefore we recognise other information retention models worth to be explored.

Studies on infectious diseases have produced a relevant ground knowledge that can easily be exported in the study of alternative models for epidemic protocols. In medical literature [28], mathematical models of infections’ behaviour have been widely studied: predicting the characteristics of epidemics for what concerns the way they spread and affect populations can be of relevant value for the organisation of possible vaccination plans or containment actions.

These models, as predictable, have largely inspired the study of gossip techniques in information dissemination: for instance, the gossip interval is the equivalent of contact rates in epidemiology and equations used to forecast the diffusion of diseases within population have been used to predict the effectiveness of gossip dissemination techniques.

**Epidemic models**

Different models have been proposed to describe epidemics and deterministically try to characterise their dynamics. A lot of parameters are involved in their specification, most of which are not interesting for our discussion: anyway, a main classification is based on what is the life cycle of an infected individual.

The most used and simplest model is the SIR model. The letters in the name are an acronym for **Susceptible-Infected-Recovered**, reflecting the phases gone through by an individual affected by the infection; the basic variant assumes a closed population where a fraction $S_0$ is initially susceptible and another fraction $I_0$ of infected elements is introduced. Equation 4.1 shows the dynamics of that model: $S$, $I$ and $R$ represent respectively the number of susceptible, infected and recovered individuals in time, $N$ is the population size, $\beta$ is the contact rate — i.e. the average number of adequate contacts (contacts sufficient for transmission) of a two people per unit time — and $\frac{1}{\gamma}$ is the average duration of an infectious period.

\[
\begin{align*}
\frac{dS}{dt} &= -\frac{\beta IS}{N}, & S(0) &= S_0 > 0 \\
\frac{dI}{dt} &= \frac{\beta IS}{N} - \gamma I, & I(0) &= I_0 > 0 \\
\frac{dR}{dt} &= \gamma I, & R(0) &= R_0 > 0 
\end{align*}
\]

(4.1)

This SIR model does not take into account births or deaths in the populations, (there are variations that do that) and also assumes that, once recovered, an individual is no longer susceptible to the same infecting agent.
The SIS model, instead, varies from the SIR in that an individual goes from the infected state back to the susceptible state: this means that, once healed from the disease, the individual can contract the infection again if he has another contact with an infected person. The dynamics of the model are simplified with respect to the SIR model since there is a smaller set of states to consider (Equations 4.2).

\[
\begin{align*}
\frac{dS}{dt} &= -\beta \frac{IS}{N} + \gamma I, & S(0) &= S_0 > 0 \\
\frac{dI}{dt} &= \beta \frac{IS}{N} - \gamma I, & I(0) &= I_0 > 0
\end{align*}
\]  

Another similar variation is the SIRS model, which considers recovered individuals to become again susceptible to the infection after an average amount of time \( \frac{1}{\phi} \), as shown in the group of Equations 4.3.

\[
\begin{align*}
\frac{dS}{dt} &= -\beta \frac{IS}{N} + \phi R, & S(0) &= S_0 > 0 \\
\frac{dI}{dt} &= \beta \frac{IS}{N} - \gamma I, & I(0) &= I_0 > 0 \\
\frac{dR}{dt} &= \gamma I - \phi R, & R(0) &= R_0 > 0
\end{align*}
\]  

The last model we want to introduce is also known as “infected-forever”: according to it a person never gets healed from an infection (nor it dies), following an SI life cycle.

\[
\begin{align*}
\frac{dS}{dt} &= -\beta \frac{IS}{N}, & S(0) &= S_0 > 0 \\
\frac{dI}{dt} &= \beta \frac{IS}{N}, & I(0) &= I_0 > 0
\end{align*}
\]  

The SI model is the simplest of the presented, since there are only two states and only a possible transition from susceptible to infected. Furthermore, from the Equations 4.4 it can be easily noticed that after a finite amount of time, which depends on \( \beta \) and \( I_0 \), the system will reach a stability condition where the infection has taken over all the population.

**Epidemic models in Gossip protocols**

The different epidemic models we presented in the previous section are able to accurately represent the dynamics of epidemic information dissemination. An information being distributed can be thought as a single infection that is spreading in the population; peers that never received the information are susceptible peers, while peers that have included it in their state are considered to be infected.
The information retention time determines what happens to a peer after he has entered in the infected state. If the information is kept permanently ($\frac{1}{\gamma} \to \infty$), the gossip protocol follows an infected forever (SI) model. Otherwise, if the information is deleted after a time $\frac{1}{\gamma}$, the peer is considered healed from the infection. Then, depending on what happens, the gossip protocols follow either an SIR, SIS, or SIRS model:

- The information, or e.g. its identifier, can be kept in a list of cured infections; whenever a peer receives again a gossip whose identifier is in the list, it discards it. In such a case the protocol follows a SIR model where one cannot be infected again by the same infection. We call these protocols SIR gossip protocols.

- After the information is removed from the state nothing about it is remembered. A peer is susceptible again to the same infection, i.e., when it receives again an information it has previously removed, that information is treated as new and it is stored again in the node’s state. Protocols like this follow an SIS epidemic model, thus we call them SIS gossip protocols.

- The last possible and most complex case is that of a peer moving in a recovered state only for a limited amount of time and then becoming susceptible again as prescribed by the SIRS model. In such a case a list of the identifiers of cured “infections” is kept but entries are left in it only for a finite amount of time equal to $\frac{1}{\phi}$. Akin to the previous cases, we give to this category the name of SIRS gossip protocols.

SI gossip protocols have the benefit to entail a very simple implementation and are very effective in case the information being spread has an unlimited validity in time; nevertheless, they do not adapt very well in the case that the gossiped data get quickly stale, especially if the cost of stale information is high application-wise. For instance, in a vehicular accident information dissemination system, it is crucial that the data being propagated reflects the updated state of a possible accident.

SIR protocols have the advantage of avoiding the proliferation of stale data across the network, at the expense of a greater management cost for what concerns keeping a list of “cured infections”. Selecting an appropriate value for the retention time $\frac{1}{\gamma}$ becomes a crucial and application dependent choice which has to be considered with great attention: once removed from the state, data is lost forever.

Unlike SIR and SI protocols, an SIS gossip approach does not consider the received information as valid forever, nor it gives a final expiration time to the received data. The advantage is that information is kept only if it is not refreshed for a long time: this approach gives greater flexibility since a source of information can individually decide when its spread data are no longer valid, and simply stop gossipping about that to stop the infection.

SIRS protocols, in our opinion, do not add very much value to SIS models at the expense of an increased complexity in their management: they have the same
benefit of allowing a refresh of old information while permitting to remove stale data, but we honestly do not see the advantage of being “immune” to the infection for some amount of time.

**Epidemic models in Tribler**

The BuddyCast Gossip Engine does not impose — by itself — any of the presented epidemic models to applications using it. However, as we remarked in the opening of this Section, both the part we have called “BuddyCast Data Exchange” and the ChannelCast protocol adopt an *infected forever* (SI) model.

We do not see any reason, except the increased simplicity, for those protocols to use that model instead of, for instance, an SIS model: in fact, due to the fast dynamics of overlay networks information about peers cannot be considered sufficiently stable to be kept forever. The same can be said for data about channels: in such a highly variable scenario it is very likely that some peers create channels and then they leave the network forever without updating the channel anymore: with a SI model the peers who received announcements about these channels will keep that data forever.

Being heavily based on ChannelCast, our system uses the same infection model: this could cause the effect that long lived peers will tend to accumulate announcements for old metadata that possibly is no longer available in the network. A study on different infection models is very desirable for the future.

Furthermore, it could be interesting to study the applicability of an epidemic model which considers also deaths and births in the population, accounting for peers that join and leave the network.

**Summary**

In this Chapter we presented our research efforts towards the design of a rich-metadata dissemination protocol and architecture focused on a particular use case that we find significantly important: the diffusion of subtitles for video items shared within a peer-to-peer overlay. Subtitles are a very valuable rich metadata type since it permits to consume a digital video in localisations others then the content’s original one; they are one of the principal types of metadata which can be found in private communities Web sites, but existing state-of-art peer-to-peer products still lack of a satisfactory support for an integrated distribution of them.

Our architecture is built on top of the services offered by Tribler, a peer-to-peer platform developed by the Delft University of Technology. Tribler provides an interesting Overlay abstraction thought for optimised content search, and offers a rich set of side services such as a Gossip Engine based on BuddyCast (its Overlay Management and Content Discovery protocol), or a basic abstraction of the concept of channels built around another epidemic protocol known as ChannelCast.
The announcement phase of our dissemination process is based on a very lightweight extension of ChannelCast. A new field in the protocol messages announces metadata using a 32-bit string representing the subtitles available within a limited set of supported languages. Subtitles content is fetched using a simple request-response protocol implemented through two new Overlay Messages: direct download has been preferred over BitTorrent-like transfers because the small size of subtitles files would cause long protocol bootstrap times, several orders of magnitudes bigger then the actual transfer time. To increase the availability of published subtitles we leverage the subscription mechanism already implemented in ChannelCast: subscribers of a channel automatically download subtitles as soon are they are available, thus becoming an additional source for them. From received gossips, peers build a partial map of possible available content sources which is then used in the retrieval phase.

We recognise several limitations in our design: the direct download approach, for instance, implies a non optimal usage of all the available bandwidth since the content is retrieved from a single peer even if it is located in more then a single location; nevertheless we preferred to go for simplicity since in our use case this does not represent an important problem considered the small size of subtitles. Another issue is determined by the gossip approach adopted in Tribler, which can likely cause duplicate announcements to be received by a single peer even though the resulting overhead is kept limited thanks to our lightweight approach.

Finally, we remarked that the epidemic infection model adopted by BuddyCast and ChannelCast, which is the infected forever (or SI) model, can have as a possible negative consequence the retention of old and stale information by peers in the system. The exploration of an alternative model such for example the SIS could be an interesting research idea.
Chapter 5

Implementation Details

Metis was implemented and tested as a series of Python modules integrated in Tribler. Our efforts will be included and distributed in the next release of Tribler’s Core API (version 5.3) and in the P2P-Next reference platform NextShare M24 [1].

In this Chapter we will analyse some implementation details that we believe significant in order to acquire a deep understanding of our system’s internals. However, readers who are not interested in how very low level interactions are modelled and implemented may skip directly to Chapter 6, since the concepts being explained here are not strictly necessary to understand the remainder of the Thesis.

The Chapter is organised as follows: in Section 5.1 we will explain some basic concepts that we believe essential to understand the rationale behind our implementation choices in Metis. Then, Section 5.2 will delve into the implementation of the architecture described in Chapter 4, explaining how the designed components realise their intended behaviour. Finally, Section 5.3 will give an outline of our approach to testing, and will cover the unit testing and integration testing phases of our development process.

5.1 Preliminary Knowledge

This first part of the Chapter presents a set of concepts that we believe indispensable to put our implementation work in a context; in Section 5.1.1 the main general principles that guided us in the design of the code are presented and discussed. Following, in Section 5.1.2, the general structure of Tribler’s codebase is introduced and its rationale is explained, while in Section 5.1.3 we present the parts of Tribler’s internal API that are more relevant to our work.

5.1.1 Implementation Guidelines

The implementation of the objects realising the architecture presented in Chapter 4 creates several challenges and open issues to cope with. We tried to face them case by case, choosing each time the solution that we recognised to be the best: anyhow
— whenever we had to take decisions – we always kept in mind a series of practical guideline principles. We find it worth to report here the most important ones:

**Single responsibility.** Each class should absolve a single functionality, implemented in the simpler possible way. The interface it exposes to external clients should be as well as simple as possible: for every single task it should expose no more then one method; publishing a small set of methods enables easier unit testing and better code readability.

**Design for reuse.** Whenever possible, a component should try to abstract from the exact implementation scenario (i.e. subtitles dissemination), so that it can be exported and easily adapted to different use cases. Of course, some of the components will necessarily be very specific: however, other objects should not depend directly on them, but on some higher-level abstraction, in a way that allows their easy replacement.

**Design for flexibility.** The behaviour of the objects in the system should be, to some extent, customisable, thus permitting to adapt it to the different needs of their users and their clients. Where possible, this is achieved through personalisation at objects’ construction time: two of the techniques we often used are the addition of constructor flag parameters, which somehow specify the desired behaviour, or the use of dependency injection patterns which alters an object’s functionality by passing to it different dependencies at initialisation time.

**Design for testing.** The structure of an object must be thought carefully considering the possibility to test its code through isolation and integration testing. Most of the times, a pure Test Driven Development (TDD) approach was used: unit tests for a class were written before coding the actual implementation; this usually had as direct consequence on the achievement of better modularity, along with improved simplicity. The use of dependency injection has also an important impact on testability: at test time, mock objects can be passed instead of real dependencies, improving the isolation of the code under test.

**Complexity hiding.** Where possible, complex algorithms and low level interactions should be hidden in single objects, and exposed via simple interfaces, allowing to improve the understandability and simplicity of the code of higher-level services.

### 5.1.2 Packages Structure

Tribler’s codebase has grown over more then four years of continuous development, with contributions from Master and PhD students, researchers and professional workers. It currently counts 440 python modules and more than 120000
Table 5.1: Tribler’s main packages structure. Only the first level of the hierarchy is shown, except for the packages which are relevant for our discussion.

<table>
<thead>
<tr>
<th>Package</th>
<th>Content</th>
</tr>
</thead>
<tbody>
<tr>
<td>Tribler/Main</td>
<td>Application main module and GUI classes</td>
</tr>
<tr>
<td>Tribler/Video</td>
<td>Integrated video player widgets and VLC wrapper</td>
</tr>
<tr>
<td>Tribler/Tools</td>
<td>Stand alone utilities and helper applications</td>
</tr>
<tr>
<td>Tribler/Lang</td>
<td>Internationalisation and localisation related modules</td>
</tr>
<tr>
<td>Tribler/Debug</td>
<td>Tools and macros for debugging</td>
</tr>
<tr>
<td>Tribler/Test</td>
<td>Unit and integration tests code</td>
</tr>
<tr>
<td>Tribler/Core</td>
<td>Tribler’s core modules: overlay management and overlay applications</td>
</tr>
<tr>
<td>Tribler/Core/APIImplementation</td>
<td>Classes implementing the Tribler API exposed to modules outside the Core</td>
</tr>
<tr>
<td>Tribler/Core/Overlay</td>
<td>Implementation of Tribler’s Overlay Network: Network management code, and Overlay’s API</td>
</tr>
<tr>
<td>Tribler/Core/BuddyCast</td>
<td>Epidemic protocols implementation</td>
</tr>
<tr>
<td>Tribler/Core/CacheDB</td>
<td>Persistence related modules</td>
</tr>
</tbody>
</table>

lines of code. Hence, a consistent organisation of the overall structure is essential in order to allow its maintainability.

Table 5.1 shows the organisation of the system in terms of packages; for each of them a brief description is also reported. Only the first level of packages hierarchy is represented, except for the Tribler/Core package which — as we will see later — has a central role in our discussion.1 In fact, it contains all the modules which realise the actual peer-to-peer platform, including the messaging API, the BuddyCast and ChannelCast epidemic protocols, and every other Overlay Application (including Metis).

5.1.3 Tribler API Basics

Within Tribler’s Core, six object types are very significant in order to understand how Metis works: the Session object, the OverlayThreadingBridge, the SecureOverlay and OverlayApps classes, and BuddyCastCore and

1Some packages are omitted on purpose in order to keep the table size small.
Figure 5.1: Class diagram showing some of the Tribler Core objects. All the classes implement the Singleton pattern, and their are built during the construction of the Session object.

ChannelCastCore. Each of these classes is instantiated as a singleton, meaning that there exists only one instance of it for one running Tribler client. Figure 5.1 shows an UML class diagram, which depicts their structure and shows their relations (the diagram simplifies from all the details that are unnecessary for our discussion).

The Session object basically represents a running instance of the Tribler Core and it is the Core’s central class: it is created during the bootstrap process, once a client is started. Modules outside the Core package access Tribler’s API through the Session object. Session’s __init__ method (i.e. its constructor) has the role to initialise every other component in the Core, and to configure them according to the user’s preferences which are normally stored in a file. Since it has to be accessed by multiple clients, both inside and outside the Core, and probably by more than one thread, every access to the Session’s state is strictly synchronised and methods executions are serialised in sequential order.

The SecureOverlay is an abstraction of the Peer-to-Peer Overlay middleware used by Tribler. It keeps track of active connections and exposes a very simple API for message routing across peers in the overlay: for example, through calls to the send() method, an arbitrary message can be forwarded to another peer simply knowing its permanent identifier (PermId). Network requests coming from different threads are queued and executed according to a simple FIFO priority
Figure 5.2: The OverlayThreadingBridge's add_task method adds task in a FIFO queue. The tasks are executed by the OverlayThread.

Figure 5.3: Overlay Applications may call the send method on the OverlayThreadingBridge which, in turns, demands the scheduling of a network transfer to the SecureOverlay.

scheme, and they are all run by a single thread, namely the NetworkThread.

A second fundamental active control flow in Tribler is represented by the OverlayThread, which runs all the code belonging to Overlay Applications. As we saw in Section 4.1, the scheduling of the tasks for this thread is performed by the OverlayThreadingBridge, which also bridges the NetworkThread and the OverlayThread; its interface includes an add_task() method which simply enqueues a new function to be called by the Overlay Thread (again in FIFO order), but also methods that strictly resemble the SecureOverlay's interface: through those methods, in fact, Overlay Applications can use Overlay services in a synchronous, safe way. If synchronisation is required, methods in the Overlay-ThreadingBridge accept an optional callback function which is called once
Table 5.2: Additional packages introduced to Tribler’s codebase. The path T/C/ is an abbreviation for Tribler/Core/ and T/C/S stands for Tribler/Core/Subtitles.

- **T/C/Subtitles**
  - Subtitles dissemination
  - Overlay Application modules

- **T/C/S/MetadataDomainObjects**
  - Domain Objects implementation

- **T/C/S/SubtitleHandler**
  - Modules for the low level handling of the request response protocol for subtitle transfers

- **T/C/ChacheDB/MetadataDB**
  - Database wrapper for rich metadata information persistence.

The requested operation is completed. Figures 5.2 and 5.3 show a schematic view of how task and message scheduling work.

The `OverlayApps` object, instead, acts as message dispatcher for the `Overlay Applications`: every time a message is received by the `SecureOverlay`, it is forwarded through a short chain, which goes from the `OverlayThreadingBridge` to the `OverlayApps` instance. The latter keeps in its state a table where it associates message types to their appropriate message handler functions: any `Overlay Application` can register a message handler for one or more particular message types on the `OverlayApps` and have that function called in response to the delivery of a new message of the specified types.

`BuddyCastCore` and `ChannelCastCore` are the objects embodying the two Overlay Applications which implement the BuddyCast and ChannelCast epidemic protocols (see Section 4.1.2 and Section 4.1.3). As explained in Section 4.5.2, their `createAndSend[ChannelCast|BuddyCast]Msg` are called by the Gossip engine whenever an epidemic exchange with another peer is in progress. In the same way the `got[ChannelCast|Buddycast]Msg` is called when an incoming gossip is received: reflecting the general architecture for epidemic protocols we introduced in Section 3.5.1, the two classes also have a method — respectively `handleBuddyCastMsg` and `updateChannels` — which is used to merge the received state with the peer’s own state.

### 5.2 Metis Implementation

Exploiting the basic services discussed in the previous Section, we implemented a set of Python modules and classes which realise Metis as an Overlay Application for the Tribler platform.

The packages structure presented in Table 5.1 has been extended as reflected in Table 5.2: In the following Sections we will analyse some implementation details of our code by separately focusing on different aspects of our system, i.e.:
Figure 5.4: Subtitles and Rich Metadata are represented through instances of MetadataDTO and SubtitleInfo classes.

- Domain Objects and Persistence
- Announcement Gossip
- Subtitles Retrieval
- Service API

5.2.1 Domain Objects and Persistence

The way our domain entities are modelled as python objects and the mechanisms associated to their persistence are strictly related: this is the reason why we decided to examine both in the same Section.

The Tribler/Core/Subtitles/MetadataDomainObjects package implements our domain model: its content is shown in Figure 5.4. All the rich metadata information about an item in a given channel is represented, in memory, as an instance of the class MetadataDTO which, as the name suggests, acts as data transfer object for that information. An instance of MetadataDTO can be identified through its properties channel and infohash, both stored as binary strings. Among the interesting operations a MetadataDTO object can perform, very worth to analyse are those related to the serialisation and the signature of the data. A MetadataDTO, in fact, permits through the serialize method, to dump all its information into a python tuple, which can be transmitted through the Overlay services. Related to this method are the sign and verifySignature methods, which respectively allow a peer to sign the rich metadata objects it adds with its permanent identifier, and allows other peers to verify that the signature is
def verifySignature(self):
    
    """
    Verifies the signature field of this instance.
    
    The signature is verified against the packed version of this instance. See serialize
    """
    assert self.signature is not None
    toVerify = self.serialize()
    binaryPermId = self.channel
    return Tribler.Core.Utilities.permid.verify_data(toVerify,
                                                       binaryPermId, self.signature)

Listing 5.1: Signature verification process of a MetadataDTO

Figure 5.5: Interface of the class MetadataDBHandler, acting as a wrapper around the sqlite embedded RDBMS.

valid: the signature is taken over the serialised form of the MetadataDTO. Listing 5.1 shows an excerpt with the central part of the code used for the verification process.

To access the list of subtitles associated to an instance, the method getAllSubtitles can be invoked: it returns a dictionary whose keys are ISO 639-2 three character strings, and whose values are instances of the SubtitleInfo class. This class keeps all the information relative to a single subtitle, and its properties allow to retrieve its language code, its SHA-1 integrity checksum, and its file system path (in the case it is locally available). Moreover methods exist which allow to compute or verify a subtitle’s checksum, given its local path.

Finally, a final module exists in the MetadataDomainObjects package, called Languages. It contains utility methods, mainly used to convert lists of ISO 639-2 language codes to the binary bit masks that are remotely sent as described in Section 4.4, and back again. An excerpt of the code used for the conversion is shown in Listing 5.2.

The persistence of the information enclosed in the domain objects is taken care by a new module added to the Tribler/Core/CacheDB package, which we called MetadataDBHandler: Figure 5.5 shows the main methods of its interface. The task of this component is to abstract from the persistent represent-
def _initMappings(self):
    """ Assigns bitmasks to languages. """
    sortedKeys = sorted(self.supportedLanguages.keys())
    for i, code in enumerate(sortedKeys):
        self.langMappings[code] = 1 << i

def maskToLangCodes(self, mask):
    """ Given a int bitmask returns the list of languages it represents. """
    assert mask < 2**32 , "Mask mast be a 32 bit value"
    assert mask >=0 , "Mask must be positive"
    codeslist = []
    for code, cur_mask in self.langMappings.iteritems():
        if mask & cur_mask != 0 :
            codeslist.append(code)
    return sorted(codeslist)

Listing 5.2: The simple algorithm converting a bit mask into a list of codes. The "private" function _initMappings creates an hashtable in which individual bit codes are indexed by their ISO 639-2 three charcters code.

tation of data, and provide convenient methods to store and retrieve our domain objects. Like all the existing Tribler’s MegaCaches, it delegates the persistence concerns to an sqlite embedded relational database engine. In order to store data, the MetadataDBHandler converts the objects to be permanently saved according to the relational model of the underlying database (check Listing 5.3 for the full model) and, conversely, it transforms the relational data back into python objects when the database is queried and results are returned. A noteworthy particular is the way binary data, such as peers’ permanent identifiers or items’ infohashes, is serialised to the disk; they are stored in the database encoded as Base64 ASCII strings, but they are used in their decoded form throughout the application: all these conversions are hidden from the outside by our database wrapper.

5.2.2 Announcements Gossip

The general interactions that make it possible to augment ChannelCast messages with new subtitles announcement information have been already described in Section 4.5.2. Here we will briefly discuss how this was realised in the code.

Whenever a ChannelCast message has to be prepared, the ChannelCastCore method’s createChannelCastMsg is invoked: in order to keep at minimum the impact of our extension to the pre-existing code, we changed it only by adding
CREATE TABLE Metadata (
    metadata_id integer PRIMARY KEY ASC AUTOINCREMENT NOT NULL,
    publisher_id text NOT NULL,
    infohash text NOT NULL,
    timestamp integer NOT NULL,
    description text,
    UNIQUE (publisher_id, infohash),
    FOREIGN KEY (publisher_id, infohash)
    REFERENCES ChannelCast(publisher_id, infohash)
    ON DELETE CASCADE
);

CREATE INDEX infohash_md_idx
on Metadata(infohash);

CREATE INDEX pub_md_idx
on Metadata(publisher_id);

CREATE TABLE Subtitles (
    metadata_id_fk integer,
    subtitle_lang text NOT NULL,
    subtitle_location text,
    checksum text NOT NULL,
    UNIQUE (metadata_id_fk,subtitle_lang),
    FOREIGN KEY (metadata_id_fk)
    REFERENCES Metadata(metadata_id)
    ON DELETE CASCADE,
    CONSTRAINT lang_code_length
    CHECK ( length(subtitle_lang) == 3 )
);

CREATE INDEX metadata_sub_idx
on Subtitles(metadata_id_fk);

CREATE TABLE SubtitlesHave (
    metadata_id_fk integer,
    peer_id text NOT NULL,
    have_mask integer NOT NULL,
    received_ts integer NOT NULL, --timestamp indicating when the mask was received
    have_mask_length
    FOREIGN KEY (metadata_id_fk)
    REFERENCES Metadata(metadata_id)
    ON DELETE CASCADE, -- the fk constraint is not enforced by sqlite
    CONSTRAINT have_mask_length
    CHECK (have_mask >= 0 AND have_mask < 4294967296)
);

CREATE INDEX subtitles_have_idx
on SubtitlesHave(metadata_id_fk);

-- this index can boost queries
-- ordered by timestamp on the SubtitlesHave DB
CREATE INDEX subtitles_have_ts
on SubtitlesHave(received_ts);

Listing 5.3: SQL query creating the relational structure used to store our domain objects. Several indexes are created to speed up common queries.
def createChannelCastMessage(self, selversion, dest_permid=None):
    message = {}
    # The ChannelCast message is built
    ...
    if selversion >= OLPROTO_VER_FOURTEENTH:
        d = self.richMetadataInterceptor.addRichMetadataContent(m, dest_permid)
    return d

Listing 5.4: At the end of the creation of a ChannelCast message, the RichMetadataInterceptor component intervenes to enrich it with announcement data. The constant OLPROTO_VER_FOURTEENTH identifies the updated Overlay Protocol version, supporting subtitle dissemination.

def updateChannelcastDB(self, source, selversion, message):
    # Processing rich metadata part.
    if selversion >= OLPROTO_VER_FOURTEENTH:
        self.richMetadataInterceptor.handleRMetadata(source, message)
    ...
    # normal processing continues
    ...

Listing 5.5: Incoming ChannelCast message are intercepted as they come in, and the subtitles announcement information is extracted and processed.

two lines at the end of the method: they are shown in Listing 5.4. Similarly, whenever an incoming ChannelCast gossip is delivered to the ChannelCast-Core, before its content is used to update the channels data, it gets intercepted and its subtitles announcement information is extracted and processed, as shown in Listing 5.5. These two tasks are performed by a component defined by the class RichMetadataInterceptor, whose extremely simple interface is shown in Figure 5.6.

A last important component, that we would like to discuss, is the PeersHaveManager which manages the have masks (see Section 4.4) received inside announcements messages. As shown in Figure 5.6, the homonym class exposes the newHaveReceived method which is used to store new masks and update the vision the peer has about the position of subtitles across the overlay. To retrieve possible available sources for a subtitle, the PeersHaveManager also has a getPeersHaving method, returning a list of peers’ permanent identifiers which can be queried for content. The returned list should be ordered so that peers that have much higher probability to be online and to have the required data are in front of the list; several criteria could have been employed in order to achieve that: we
5.2.3 Subtitles Retrieval

The subtitles retrieval protocol is realised through the interaction of several python modules, which include the PeersHaveManager we just presented, the SubtitlesHandler module in Tribler/Core/Subtitles, and all the modules in the SubtitleHandlerSupport subpackage. An UML Class Diagram of these parts is reported in Figure 5.7.

The SubtitlesHandler and SubsMsgHandler classes cover two complementary roles. The former implements the high level logic of our subtitles retrieval protocol, while the latter is notified by the former about the havemasks which are received within announcement messages.

Figure 5.7: The SubtitlesHandler class manages the high level logic of the request-response protocol, while the SubsMsgHandler deals with the low level message construction and dispatching.

Figure 5.6: RichMetadataInterceptor and PeersHaveManager. The latter is notified by the former about the havemasks which are received within announcement messages.

chose a simple heuristic that selects first peers from which a have mask has been received more recently.
def _checkingUploadQueue(self):
    if not self._tokenBucket.upload_rate > 0:
        return

    while len(self._uploadQueue) > 0:
        responseData = self._uploadQueue.pop()
        encodedMsg = self._createSingleResponseMessage(responseData)
        msgSize = len(encodedMsg) / 1024.0  # in kilobytes

        if msgSize > self._tokenBucket.capacity:
            continue  # check other messages in the queue

        if self._tokenBucket.consume(msgSize):
            self._doSendSubtitles(responseData['permid'], encodedMsg,
                                   responseData['selversion'])
        else:
            # tokens are insufficient wait the necessary time
            self._uploadQueue.push(responseData)
            neededCapacity = max(0, msgSize - self._tokenBucket.tokens)
            delay = (neededCapacity / self._tokenBucket.upload_rate)
            self._nextUploadTime += delay
            self.overlay_bridge.add_task(self._checkingUploadQueue,
                                          delay)
    return

Listing 5.6: The SubsMsgHandler periodically checks its internal upload queue for SUBS messages, and sends them according to a Token Bucket based algorithm.

retrieval protocol: it processes the details of locally generated requests, handles the subtitles arriving from other peers as response to remote queries, and checks whether the peer can answer remotely generated requests. The latter, instead, deals with low level messaging details: it copes with the creation of the GET_SUBS and SUBS messages in their proper format (see Appendix A for further details), verifies the formal validity of incoming messages, and discards response messages that are not answers to previously generated queries; since SUBS request can be relatively large (a single SUBS message can be up to 1 MB), the SubsMsgHandler adopts a simple upload rate limiting technique based on a Token Bucket and implemented through the use of a FIFO upload queue, which is periodically checked and emptied: the upload algorithm is shown in Listing 5.6.

5.2.4 Service External API

To use all the services offered by Metis, external clients will need the reference to a single object. This choice greatly simplifies the use of our system, thus encouraging its adoption. This object acts as a facade for the entire subsystem, and is an instance of the SubtitlesSupport class. A reference to it is retrievable through the Session object whose interface has been extended with the addition
of a method called get_subtitles_support_facade.

The SubtitleSupport object provides method to publish, search and retrieve for available subtitles in an easy and intuitive way, and tries to hide all the complexity of the underlying mechanisms. A complete description of how to use its services can be found in the Metis User Guide, reported in Appendix B.

5.3 Testing the System

A non negligible part of time of our development process has been spent writing test code. We performed three different types of testing on the developed system: unit testing, integration testing and system testing. In the following two sub-sections we will discuss how we performed the first two, and what benefits we achieved through them. System testing, instead, has been performed as a “side effect” of our evaluation method and, therefore, we will write about it in Chapter 6.

5.3.1 Unit Testing

Unit Testing is a method according to which individual units of code are tested in isolation to verify whether they correctly absolve the functionalities they were designed for. In our case, as in most object oriented system, the units of code are identified with the individual classes making up the system. The diffusion of this practise and the development of several tools which assist the programmer in writing and running tests, has made unit testing a fundamental development technique in modern software development. The Test Driven Development (TDD) model claims that unit tests should be the main driving force in the design of a software system: the system should be decomposed in its basic functionalities and, for each of them, a set of unit tests should be written before the actual code implementing it. Several benefits are achievable by following this development process:

- Improvement of the quality of the code, by having automated routines which constantly check the correctness of components from the first moments they are written, through all the development cycle.
- Improvement of the design, by having the developers focus on functionalities and the practical behaviour needed to implement them, since they test it before writing the code.
- Improvement on the modularity of the system, by driving the attention on the interfaces of the components rather then their implementation; unit test should verify only the external behaviour that a unit of code exposes: as a matter of fact, a test is the component’s first client and, indeed, a very demanding one.
- Improvement of the isolation of components and of their interchangeability, by encouraging the adoption of particular testing patterns, such as the
use of mock objects combined with dependency injection. As we will see, mock objects help to verify a unit’s behaviour isolating it from its external dependencies that may interfere with it.

In the design of Metis, we tried to adhere to these practices and before writing any module, we first designed a set of thorough test cases for it. This helped us to achieve a good degree of confidence about the correctness of the code, and, through the joint use of a version control system (we had a dedicated branch on Tribler’s subversion server), it became easier to perform even substantial refactoring sessions, since we had automated routines that checked that the functionalities were preserved.

In order to write the test code, we used the PyUnit, the built-in testing framework for python inspired by Java’s JUnit. All the classes were tested in complete isolation: this was immensely helped by python’s dynamic types system, and by what is commonly referred to as duck typing. In fact, the dependencies passed in objects’ constructors (or injection methods) did not have to explicitly comply to a given interface, but it was sufficient for them to implement only the methods required by the class under test: therefore, we could easily design helpful mock objects without external support libraries. We report in Listing 5.7 a test method extracted from a Test Case for the class SubsMsgHandler to illustrate the use of PyUnit and mock objects. That code, like all the test code, is stored in the Tribler/Test package.

5.3.2 Integration Testing

The original Tribler’s codebase did not include unit tests as we defined them in the previous Section. Instead, the pre-existing modules were tested through integration tests which verify whether the objects, which realise a given functionality, are able to work correctly together and produce the expected output. Even if the components are thoroughly verified through unit testing, including an integration testing plan in the development process has several important benefits:

- Reveal semantic inconsistencies between the interfaces of different components, which are not detected by isolated unit tests.
- Verify the correctness of communication and messaging patterns, especially between distributed components.
- Check the validity of the interactions between new and legacy components, and verify if the new objects are able to integrate seamlessly with their surrounding environment.

To conduct our tests we used a simple ad-hoc testing framework included in Tribler’s sources and called TestAsServer. Basically, it permits to instantiate and configure a complete Session, and select which services should be run by it. It
class TestSubtitlesMsgHandlerIsolation(unittest.TestCase):
    def setUp(self):
        self.ol_bridge = MockOverlayBridge()
        self.tokenBucket = MockTokenBucket()
        self.underTest = SubsMessageHandler(self.ol_bridge, self.tokenBucket)

    def testSendSubtitlesRequestConnected(self):
        langUtil = LanguagesProvider.getLanguagesInstance()
        request = {
            'channel_id': testChannelId,
            'infohash': testInfohash,
            'languages': ['kor']
        }
        self.underTest.sendSubtitleRequest(testDestPermId, request, None, None, OLPROTO_VER_FOURTEENTH)
        # selversion was 1, no connect should have been called
        self.assertEqual(0, self.ol_bridge.connect_count)
        # send must have been called one time
        self.assertEqual(1, self.ol_bridge.send_count)

        binaryBitmask = uintToBinaryString(langUtil.langCodesToMask(['kor']))
        expectedMsg = GET_SUBS + bencode((testChannelId, testInfohash, binaryBitmask))
        passedParameters = self.ol_bridge.sendParamsHistory[0]
        self.assertEqual(testDestPermId, passedParameters[0])
        self.assertEqual(expectedMsg, passedParameters[1])

Listing 5.7: An Unit Test example, showing the use of the PyUnit framework and mock objects
class TestSubtitleMessages(TestAsServer):
...

def subtest_receptionOfSUBS(self):
    ol_conn = OLConnection(self.my_keypair, 'localhost', self.hisport)
    bitmask = langUtil.langCodesToMask(['nld'])
    binmask = utilities.uintToBinaryString(bitmask, length=4)
    request = GET_SUBS + \
        bencode((self.anotherpermid, self.testInfohash, binmask))

    subshandler = SubtitlesHandler()
    subshandler.register(ol_conn, self.richMetadata_db, self.session)

    ol_conn.send(request)
    subs_data = ol_conn.recv()

    data = bdecode(subs_data[1:])
    self.assertEquals(SUBS, subs_data[0])
    self.assertTrue(isinstance(data, list))
    self.assertEquals(4, len(data))
    # for fields
    self.assertEquals(self.mdto.channel, data[0])
    self.assertEquals(self.mdto.infohash, data[1])
    self.assertEquals(binmask, data[2])
    self.assertTrue(isinstance(data[3], list))
    self.assertEquals(1, len(data[3]))
    with codecs.open(self.sub1, "rb", "utf-8") as sub:
        expectedContents = sub.read()
    self.assertEquals(expectedContents, data[3][0])

    ol_conn.close()
...

Listing 5.8: An Integration Test example, showing how to use the TestAsServer framework

also instantiates a loopback overlay, meaning that all the messages that are sent through it are delivered to the sender itself. Using this framework, an overlay application developer can test its own code inside a nearly-complete Tribler instance, and verify that it produces the desired communication behaviour and output.

We developed separate integration test modules for the announcement protocol and the subtitles retrieval protocol: the first extends the test module which was previously developed to test ChannelCast functionalities, while the second was created from scratch. Listing 5.8 shows an example integration test for the request-response subtitles retrieval protocol.
Summary

We implemented a prototype of the architecture presented in Chapter 4 as an Overlay Application for the Tribler Platform. Our efforts are going to be released in the next Tribler’s official release. We used the Python programming language to code the different components of the system, mainly for consistency with the Tribler’s Core API.

Our code design method has been largely inspired by the Test Driven Development model, which pushed us in considering tests writing as a central activity in the development process. Following the advises of the model, we largely used design patterns such as dependency injection and mock objects in order to keep components as independent as possible from each other and, hence, easily interchangeable. Our main goal has been to develop an object oriented system designed for flexibility and ease of reuse, whose components’ functionality is testable through automated processes and whose interface hides from external clients the complexity of their implementation.
Chapter 6

Metis Experimental Evaluation

In this Chapter we will show the results of a thorough evaluation we performed on Metis in order to understand to what extent our architecture is able to meet the requirements and the goals we set out in Section 3.1, but also to uncover possible and unexpected limitations or weaknesses that may leave space for future improvements.

The Chapter will start, in Section 6.1 with a series of considerations regarding the possible techniques that can be used to test and evaluate a peer-to-peer application. Then, in Section 6.2 we will describe the methodology we followed to conduct our experiments and in Section 6.3 the framework we designed and used to run them and to analyse resulting data will be briefly introduced; in Section 6.4 we will start showing our results by performing a detailed analysis of the singular and collective behaviour of peers in our experiments. The actual results will be presented in Section 6.5 where we will also try to understand and explain what kind of feedback they give us about our prototype. Finally, in Section 4.6.2 we will explore an alternative protocol for announcement dissemination based on Bloom filters, which aims to reduce the waste of bandwidth we discovered from the previous emulations: we will briefly outline the concepts behind this new idea, and we will present the results of several simulation based experiments we designed and run.

6.1 Experiment Methodology

Peer-to-Peer applications show an intrinsic difficulty in their testing and evaluation due to the fact that, for their nature, they are thought to work in a very large scale distributed network. Studying their behaviour in a local or very small scale environment represents, in fact, a remarkable restriction: the fast dynamics and high concurrency of events in an Overlay Network may cause synchronisation issues that are very difficult to reproduce in a controlled environment, thus inducing the emergence of possible unexpected results. Therefore, in the evaluation of a peer-to-peer system, it is of crucial importance to adopt a methodology which is able to
stress the software in conditions which reproduce — as much close as possible —
an overlay network made of a highly dynamic set of peers.

We will analyse three different approaches that can be used to achieve this goal:

- Simulation
- Emulation
- Crawling of real usage data

Through simulation, the obstacle of reproducing a complex overlay network is
moved around: instead of putting under test the actual distributed system, a model
of it is created; this model summarises and mimics the main dynamics of the real
system but it simplifies from all the unnecessary complexities and, most of the
times, it is run in a completely local environment. Simulations are particularly
useful when the analysis needs to be focused only on some particular aspects of
the system and it is acceptable — for the evaluation purposes — to abstract from
other irrelevant details. One of the main benefits of simulations is that it is very
simple to control them since they are run on just one or a few local machines;
moreover, being based on a simplified model and not on actual running instances
of the software, they can be run at speeds that are several orders of magnitude faster
compared to real time scenarios, allowing to collect a bigger volume of data in a
fixed amount of time. Another main advantage is that it is very easy to output data
in a desired format that can be used to produce results with little or no processing;
furthermore, these data is very easy to collect and summarise since none or very
little distribution is involved in its creation. The principal drawback of simulation
is that the obtained results are valid only as long as the model used for simulation is
valid: if the model does not abstract correctly and to a sufficient extent the scenario
being analysed, then its output can be completely wrong or significantly different
if compared with the behaviour of the system in a real deployment. Hence, it
is very important to accurately create the simulation model, studying the set of
assumptions under which it is able to imitate reality, and carefully choosing what
are the parameters to evaluate and the data to collect and analyse.

The emulation approach, instead, aims to reproduce the exact behaviour of the
system under study, at the same time working in a controlled environment. Instead
of dealing with a simplified model of the system, a slightly modified variant of
the actual implementation is used in order to preserve all the characteristics and
complexities that may affect its behaviour. The changes are often made to allow an
easier retrieval of the measured data and facilitate the on-line monitoring of the run-
ning experiments. For a peer-to-peer system an emulation-based experiment can
be the reproduction on a smaller scale of the overlay dynamics, including also the
aspects linked to the distribution of nodes: a possible approach is to run a certain
number of modified peers on several distributed machines and make them interact
while recording the behaviour of the system. The collection and processing of data
is slightly more difficult compared to simulation: the data to collect consists often
in logs generated by each emulated peer which record the events happened during the experiment time; these logs are likely located into different physical nodes and need to be harvested in a single place before being processed to produce summarised views. Again, it is critical to appropriately choose what are the parameters to monitor: when dealing with very complex systems the choice can be extremely hard since there may be many candidates to choose from. Also, as for simulations, it is very important to read the results considering that they may still deviate from the actual behaviour in a large scale deployment: the main reason for that is the restricted size of the experiments which can lead, for instance, to different bandwidth load conditions at single peers or to a different frequency of interactions between the members of a larger set of nodes. Anyhow, emulation is a very good method when there is the need to have a concrete feedback — not only from a theoretical point of view — from the actual implementation: through it it is possible to put under test the conjunction of all the details that a simulation may abstract, and reveal problematic interactions between system components that might not have been sufficiently explored during the design phase. Similarly, emulation can be useful, to a certain extent, also as a System Testing method and reveal possible bugs remained hidden during Unit and Integration Testing.

Crawling of real usage data, instead, aims to evaluate a peer-to-peer system after it has been already deployed to a large number of distributed nodes. In order to do that, the system is altered so that every application instance, during its runtime, locally records usage data. Periodically one or more crawlers are run to collect these data and store them in a central location: on a theoretical level the crawlers should contact every running peer that, in turn, should accept their connection and make them harvest their logs. Data retrieved this way are able to give the most accurate description of the system characteristics, being taken from a real deployment and usage scenario. Nevertheless it is not always possible — nor straightforward — to adopt this approach. First, the software must be widely distributed among users before being able to collect a sufficient amount of information: anyhow it is easy to imagine that a developer would prefer to have some results on the performances and effectiveness of a peer-to-peer application before passing to its wide scale distribution. Secondly, the process of crawling highly distributed data on a extremely dynamic environment in an efficient, thorough and secure way presents several issues: we have already remarked that an important fraction of the peers participating in an Overlay Network are not directly connectable since they are behind NATs or firewall devices; this implies that a crawling agent has either to have an effective NAT puncturing mechanism, or that the harvested data will represent only a subset of the real network, which will include only the peers accepting incoming connections. Privacy issues are also not to be underestimated: users must allow their data to be collected, and it has to be guaranteed that only authorised parties are allowed to harvest the logs from the network, for instance providing every peer with a whitelist containing public keys of trusted agents.
Table 6.1: Hardware and software configuration of the machines used in the emulation experiments.

<table>
<thead>
<tr>
<th>CPU</th>
<th>2 x QuadCore Intel Xeon E5345 @ 2.33 Ghz</th>
</tr>
</thead>
<tbody>
<tr>
<td>Main Memory</td>
<td>16 GB</td>
</tr>
<tr>
<td>OS</td>
<td>Fedora Core 8 w/ Linux 2.6.23</td>
</tr>
<tr>
<td>Internal Link Speed</td>
<td>1 Gbit</td>
</tr>
<tr>
<td>External Link Speed</td>
<td>100 Mbit</td>
</tr>
</tbody>
</table>

6.2 Experimental Setup

We chose to evaluate our implemented subtitles dissemination system through experiments based on a restricted scale emulation of peers in the Tribler overlay implementing the subtitle exchange features.

The preference of emulation over simulation is motivated primarily because we needed a realistic response from the implemented software, that could also reveal possible problems in a real deployment: as said, our system is intended to be soon integrated and merged in the P2P Next - Next Share platform, and the experiments we performed were able to give us a certain degree of confidence about its dependability.

For the emulation we used 7 machines from the Master’s Students Lab of the Software Technology Department at the Delft University of Technology. The computers were equally equipped: their configuration is reported in Table 6.1. It is worth to notice that the characteristics of both CPU speed and connectivity capacity are very high-end especially if compared to “standard” consumer hardware where the peer-to-peer platform under evaluation is likely to be deployed and used.

On each node we launch 20 instances of Tribler, and at each run their MegaCache (i.e. their state) is reset. The experiment code has been slightly modified with respect to the production code to allow remote control and event logging. In order to make our emulations as much close to reality as possible, we took particular care to make our experimental environment accurately mimic the swarm dynamics of real clients, especially for what concerns the churn rate and the uptime length of single instances: to that purpose, we decided to control the 140+ instances in our experiments following real Tribler usage traces that were collected by the Tribler Development Group during previous researches.

The traces we used describe the behaviour of 5000 users, and include the timestamps of the start and end time of their sessions, their connectivity status (i.e. whether they are offline or online and whether they are accepting external connections or not) and their download actions. For each experiment run we randomly select 140 traces, and we associate each of them to one peer in the emulation. That peer is then commanded to connect and disconnect from the overlay following the
actions of his corresponding real user.

The announcement and retrieval phase of our subtitles diffusion solution are studied by introducing in the emulated environment one moderator and publishing on its channel a single subtitle for one of its items. Studying the behaviour of the system when just one publisher is involved facilitates the tracing and the measuring of the parameters we are interested in, and the collected results are likely to be still valid — with the appropriate adaptations – even when more moderators will be introduced. The moderating instance does not follow a real peer trace, but it is artificially controlled to go up and down depending on the single experiment. Also depending on the experiment, we introduced one or two peers which subscribe to the publisher’s channel and we used them to estimate the effects of their presence on the diffusion process.

Two different settings were explored:

1. The emulated Tribler instances are run inside the official Tribler Overlay. Therefore they are able to gossip and communicate with “vanilla” versions of the client spread through the Internet.

2. The emulated Tribler instances are run inside an ad-hoc and isolated Overlay, we called Overlay #42. Inside this overlay they cannot see other peers, and they can only interact with each other.

A first set of the experiments were performed running emulations of the Overlay network for 24 hours: since our results were not significantly affected from the running time after the first hours, we decided to reduce their duration to 12 hours for the second part.

6.3 Experimental Framework

We designed a simple framework to assist us in the evaluation of our system. The framework consists in two separate parts: the first, named Emulation Control, is intended to be used to easily control the emulation experiments from a central location, while the second part, the Logs Analyser, is used to analyse the logs collected from the experiments and to produce simple graphs which describe them.

6.3.1 Emulation Control

Emulation Control is made out of a set of bash and Python scripts meant to allow a centralised control of the emulation runs. The purposes and functionalities of this subsystem are summarised in the following points:

- Start an emulation experiment. This step also includes randomly choosing traces and associate one of them to each emulated peer.
- Control every peer in the emulation, making it follow the behaviour of its associated trace.
Figure 6.1: The Emulation Control Architecture is made of a central component, the Control Centre which gives command to the Peer Agents distributed over the machines involved in the emulation.

- Start and stop the peer having the role of publisher and its possible subscribers accordingly to the needs of single experiments.
- Stop the experiment quitting every Tribler instance on every node involved.
- Collect the events logs from every machine which participated in the emulation and move and archive them into a central location.

Figure 6.1 shows how the main components of emulation control are organised and distributed. For every peer participating in the emulation a Peer Agent is run for all the duration of the experiment. A peer agent will likely, but not necessarily, reside local to the machine where the Tribler instance of the peer will be executed. Its principal role is to launch and shut down the peer-to-peer client initialising it with the correct parameters, and make it go online and offline according to what is reported in the peer trace associated to it. Figure 6.2 shows how the peer agent interacts with a Tribler instance; as indicated before, Tribler was slightly modified in two main directions:

- The modules composing the Subtitles dissemination subsystem, Channel-Cast and BuddyCast have been augmented with logging facilities, provided by the python logging module. The module has been configured to log the main events of a peer’s life (as reported in Table 6.2), and save them to a local position on the disk.
- A small subset of the Tribler’s API has been exposed to external (and possibly remote) clients through the addition of an XML-RPC [81] based server,
implemented using the python xmlrpc module. The published methods allow, for example, to cleanly shut down an instance or to subscribe a channel given its identifier.

The Emulation Control Centre, instead, is localised on a single machine and it acts as a control panel for the user. It is composed of a bunch of bash scripts which allow to start (or stop) an experiment, and to harvest and store in a central repository all the logs after an experiment.

6.3.2 Logs Analyser

The Logs Analyser is a small but extensible subsystem whose purpose is to create meaningful graphs from the log files collected from an emulation experiment. It is composed by three main functional parts:

- Parsers
- Filters
- Plotter

A schematic representation of the process logically followed by the analyser is shown in Figure 6.3. The logs — stored in a known central position by the Evaluation Control system — are first read and parsed by the appropriate Parsers which produce an in-memory representation of the entirety of the events registered during an experiment. This object can then be analysed by a set of filters: the goal of a single filter is to produce a bi-dimensional array of real numbers; depending on the filter the semantics of this array varies, but basically it represents the variation of a meaningful parameter describing the performance of the system. For instance, a particular filter — called AnnouncementRecvCDFFilter — produces an array which describes the variation in time of the percentage of peers in the emulation.
Figure 6.3: Logs are parsed and then processed by Filters which produce bi-dimensional representation of important system parameters. These values are plotted into 2-D graphs.
who received an announcement message. A plotter component can finally be used to create a graph reproducing the values given as output by one or more filters: this component internally uses the `gnuplot.py` module, which is a wrapper around the Gnuplot plotting utility.

Figure 6.4 shows a more detailed description of the structure of those modules. Basically, any `Parser` needs to implement a simple interface which consists of two methods: one returns a regular expression which singles out the set of file names of the logs which the parser is able to understand; the other is used to parse the actual content of a log, producing a list of `LoggedEvents`. An `Experiment` instance is created for every emulation experiment: using the available parsers it is populated by all the `LoggedEvents` occurred during the emulation. A `LoggedEvent` represents an action performed by a peer during an experiment, and includes all the parameters which describe it. A list of possible event types is represented in Table 6.2. A `Plotter` can plot particular aspects of an `Experiment` using objects implementing the `Filter` interface to process the events in the emulation. The interactions involved in the plotting process are shown in Figure 6.5.

### 6.4 Swarm behaviour description

Before showing the actual results obtained from the experiments, it is of crucial importance to understand the characteristics of the sample of Peer-to-Peer Overlay Network reproduced through our emulation. To do that, we analysed several parameters that we believe are able to meticulously describe the topology and general

<table>
<thead>
<tr>
<th>EventType</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>U</td>
<td>A peer has gone online.</td>
</tr>
<tr>
<td>D</td>
<td>A peer has gone offline.</td>
</tr>
<tr>
<td>C</td>
<td>A peer has sent a ChannelCast message.</td>
</tr>
<tr>
<td>RC</td>
<td>A peer has received a ChannelCast message.</td>
</tr>
<tr>
<td>S</td>
<td>A peer has sent a subtitles announcement.</td>
</tr>
<tr>
<td>R</td>
<td>A peer has received a subtitles announcement.</td>
</tr>
<tr>
<td>H</td>
<td>A peer has processed a new have mask.</td>
</tr>
<tr>
<td>SG</td>
<td>A peer has sent a GET_SUBS request.</td>
</tr>
<tr>
<td>RG</td>
<td>A peer has received a GET_SUBS request.</td>
</tr>
<tr>
<td>SS</td>
<td>A peer has sent a SUBS response.</td>
</tr>
<tr>
<td>RS</td>
<td>A peer has received a SUBS response.</td>
</tr>
<tr>
<td>C</td>
<td>A peer has consumed a subtitle content.</td>
</tr>
</tbody>
</table>

Table 6.2: Event types and corresponding description, as currently parsed by the Log Analyser
Figure 6.4: A peer agent is always running during an experiment and is responsible to control the life-cycle of a single peer, making it follow the behaviour of a trace.

Figure 6.5: A *Plotter* uses the set of filters registered on it to analyse the events in an Experiment and to produce the data to be reported in a graph.
behaviour of the Overlay Swarm. These parameters are:

- The number of peers connected to the swarm at each instant of the experiment.
- The frequency of churn events in a run (i.e. the frequency of events that modify the topology of the swarm).
- How the length of online sessions is distributed among peers. (A session of a peer identifies a time window during which that peer is continuously online)

Since the traces that guide the peers’ sessions are randomly chosen and different from an experiment to the other, it is predictable that a certain variability of these parameters will be observable between a run and another.

For each of the above measures we will report and discuss a boxplot graph — showing the statistical variability within an experiment and between a selection of 10 experiments — and a graph showing in greater detail its dynamics in one significant example.

### 6.4.1 Number of Peers

Figure 6.6 shows, trough a boxplot graph, a representation of the variability of the number of emulated peers connected to the Overlay during an experiment, in a selection of 10 experiments. It can be easily seen that this parameter presents a marked variability both during a single emulation and between different runs: the median values (represented by the black thick in the middle of each “candlestick”) vary from a minimum of 59 contemporary connected peers to 82. This variability is however reduced if only the first and third quartiles of single experiments are considered (quartiles are represented by the wider box in the candlesticks): they vary in a quite limited range, around 20 peers wide. Figure 6.7 shows the detailed variation in time of the number of peers connected to the Overlay in the case of the first of the experiments reported in the boxplot. From the graph it can be noticed how the set of the online peers is never stable, but it changes very frequently in the 24 hours of experiment.

### 6.4.2 Churn Rates

In an Overlay Network the Churn Rate is the frequency of events which modify the topology of the network, i.e. modify the set of peers which are members of the overlay. It is a critical parameter since it represents the rate at which peers join and leave the network, thus giving a measure of the stability of the network.

We measured the churn rate in our experiment runs by recording the intervals between a connection (or disconnection) event and the following one: the more
Figure 6.6: Number of peers connected to the Overlay in time in a sample of 10 emulations. Every candlestick represents a single experiment, and it shows the minimum, the maximum, and the first, second and third quartiles of the number of online peers.
Figure 6.7: The variation of the membership of the Overlay Swarm during a 24 hours experiment.
Figure 6.8: Statistical distribution of intervals between *churn events* in 10 of the conducted experiments.
these intervals are small, the higher the churn rate. From the boxplot in Figure 6.8 it can be seen that in all the reported experiments the churn rate is particularly high: the median values are under 60 seconds in all the experiments, and the third quartiles are all under 240 minutes. There is also no big difference between the various experiments. In Figure 6.9, we show how the intervals between consecutive churn events are distributed within a single emulation. We do that by reporting their cumulative distribution function (CDF): the distribution is highly concentrated around the origin of the axis: this means that the interval between two consecutive churn events tends to be very small; in the example 50% of churn events happens within 60 seconds from each other.

6.4.3 Session Lengths

A peer’s session identifies the time window during which a peer is connected to the Overlay: in different words, it is the time which elapses from the moment a peer goes online to the moment it goes offline. The session lengths distribution is able to capture how long peers remain in the system each time they appear.

Because our observations are limited in time (at most 24 hours) we must take particular care in measuring session in order to avoid biased results. In fact, indic-
Figure 6.10: Distribution of peers’ session length in a sample of 10 experiments.

ating with $\tau$ the experiment time, if we measure the length of all the session that start and end within $\tau$ we would have a bias towards shorter session: for instance, we would record a session of length $\tau$ only in the case it starts at the beginning of the observations and finishes at the end of it; on the other hand we would have $\tau$ opportunities to measure sessions of length 1. Of course, we would not be able to measure sessions longer then the observation window. To solve this dilemma we adopted the method already used by [64] and [69], which they call create based method: the measurement window is divided in two halves of length $\frac{\tau}{2}$, and we only consider sessions which began in the observation interval $[0, \frac{\tau}{2}]$; this way we can make unbiased measures of session long less then $\frac{\tau}{2}$. Nonetheless this method would still give us biases for sessions that are longer then that threshold: to avoid this, we will either measure the session length or just report that it is longer then $\frac{\tau}{2}$.

The boxplot graph in Figure 6.10 shows that the variability of peers session length across different experiments is particularly relevant, varying from very short sessions to sessions longer then half a day. Notice that for some experiment $\frac{\tau}{2}$ is more then 43200 (half a day), since not all the experiment did have the same exact duration. A better understanding of the distribution of session lengths can be
Figure 6.11: CDF of peers’ session length within a single experiment.
achieved by looking at the graph reported in Figure 6.11, which plots the cumulative distribution function of session lengths within a single experiment. For a lot of peers the session are extremely short (less then 90 minutes for the 25% of the peers), while for the upper 25% they are longer then 12 hours.

6.5 Results analysis

In this Section we will present the results we achieved during our emulation-based experiments. They are intended to show the properties of our system with respect to the optimisation of the set of parameters mirroring the goals we posed down in Section 3.1. These parameters are:

1. Cost of the announcement phase of the subtitles dissemination, in terms of the amount of bandwidth used for it by the publisher and its subscribers.

2. Diffusion effectiveness, measured as the fraction of emulated peers which successfully received announcement messages at different moments in time during an experiment.

3. Diffusion speed, in terms of the time elapsed from the moment a peer connects to the Overlay to the moment it receives the announcement message for a published subtitle.

4. Overhead of the request-response subtitles retrieval protocol, measured as the amount of bandwidth used to actually transfer subtitles content, and the effect that the introduction of subscribers has on it.

6.5.1 Announcement Overhead

We measured the impact that our announcement solution has on Tribler as the fraction of necessary bandwidth, in time, to transmit the additional information extending the previous versions of ChannelCast messages. In other words, we measured what is the cumulative size of the rich metadata fields that are sent along ChannelCast message for the announcement of a single subtitle for one item in the publisher’s channel (see Appendix A for the ChannelCast message format).

Given the format of the announcement fields, it is easy to statically compute the overhead on a single gossip message\(^1\):

- 2 bytes for the bencoding of the (empty) description string
- 12 byte string for the timestamp expressed as a bencoded integer

\(^1\)Gossip messages are sent in Tribler as BitTorrent messages: like every BitTorrent message the content of the message is sent in their bencoded format. Bencode is a mixed binary-text based encoding for loosely structured data, introduced by the BitTorrent protocol specification. For further details please consult [80].
• 6 bytes for the bencoded 32bit languages mask
• 25 bytes for the bencoded list of SHA-1 digests (containing only 1 entry for 1 subtitle)
• 67 bytes for the ECC signature
• 6 bytes for the have mask
• 2 bytes for the enclosing list

Summing these values, a 120 bytes overhead per message is obtained. Moreover the growth factor of this value with respect to the addition of other subtitles can be easily calculated: the only thing whose size changes is the list of SHA-1 digests which grows with the rate of 23 bytes (the size of 1 bencoded 20bytes string) per published subtitle. Hence it comes straightforward to see that the maximum announcement overhead for a single rich metadata field is 833 bytes, in the extreme case of an item for whom 32 subtitles in 32 different languages are published. Furthermore, considering that ChannelCast sends at most 50 moderation per gossip we have an upper bound on the overhead per message of 41650 bytes (i.e. 41 KBytes) in the very unlikely case that 50 items are advertised, each of them having 32 attached subtitles.

Therefore, it is easy to understand that the real problem about the announcement overhead can come from a wrong choice of the Gossip policy, which may result in a non optimal choice of the peer selection function or gossip period, rather than from the message format.

Figure 6.12 shows the cumulative bandwidth consumed in 12 hours by a publisher for the announcement phase in the simple case of 1 published subtitle for 1 Item. Although the experiment was conducted in the real Tribler Overlay, which includes thousands of real peers, only the emulated peers were sent the rich metadata information: this happens because a protocol version number is exchanged whenever a connection is made with a peer, enabling the sender to understand what information the receiver is able to process.

In the first 5 minutes (300 seconds) there is a rapid growth due to the fact that the gossip period for this interval is set to 5 seconds (see Section 4.1.2), after which the overhead follows a linear-like growth as connections with the experimental peers are established and gossip with them takes place: the total consumed bandwidth amounts to only 25 Kilobytes, confirming that the goal of minimising overhead is met.

Nevertheless, 25 Kilobytes is much more then the minimum required bandwidth to send announcements to all the 140 peers in the simulations: that bandwidth would amount to 140 * 120 bytes = 16 Kilobytes. Therefore, there must be a waste of bandwidth in sending duplicates: this is confirmed in Figure 6.13.
Figure 6.12: Cumulative bandwidth used to send announcement data.
Figure 6.13: Network traffic generated by unique announcement only (dashed line) compared to the part generated by duplicates only (solid line)
The dashed trajectory represents the amount of bandwidth used to send non-duplicate announcements, while the solid one represents the growth of the traffic wasted in duplicate messages. It is very significant to see that almost the 50% of the announcements sent are duplicates.

Also, it is important to notice that duplicates start to grow only after 4 hours; this amount of time corresponds to the duration of the entries in the gossip blacklist (see again Section 4.1.2). This results show that this countermeasure is helpless in preventing duplicates: in fact, after 4 hours, they start to grow significantly faster than useful announcements.

Finally, notice a periodicity of the growth of both the lines of around 15 minutes, corresponding to the gossip period of BuddyCast.

6.5.2 Announcement Coverage

We measured the announcement coverage as the fraction of peers that appeared online at least once and has been reached by one announcement message. We would like this parameter to grow fast as the publisher goes online and to be stable to values as much close to 100% as possible. That would mean ideally that, as soon as a rich metadata item is published, all the peers in the Overlay are informed within a short interval and also that new peers that come online quickly receive the same information.

Figure 6.14 shows the evolution of coverage in time for three different experimental settings. The three of them were run in the Overlay #42, but similar results were also achieved when running the experiment in the official Tribler Overlay.

- E1 represents the base case. In that scenario we run a publisher continuously for 12 hours.
- E2 differs from E1 in that the moderator goes up and down every 4 hours.
- E3 is the same of E2 except for the fact that a subscriber is introduced and run for 12 hours.

A zoom on the last 20% of the vertical axis of the same plot is reproduced in Figure 6.15.

An important result is that in every case, after around 1500 seconds from the beginning of the emulation (i.e. 25 minutes) more then the 80% of the appeared peers has been reached by announcements, and after around 5000 seconds (i.e. 83 minutes) in both E1 and E3, the coverage is stable between 90% and 100%. On the contrary, in E2 the 4 hours absence of the publisher keeping peers fed with its announcements, makes the coverage drop down to 80% as new peers connect to the overlay; this value soon goes up again as the publisher reappears. It is important to notice that this phenomenon does not happen in E3, where the presence of a subscriber manages to compensate the absence of the moderator as it goes down.
Figure 6.14: Variation in time of the announcement coverage of peers in Tribler’s official Overlay, in three different scenarios.
Figure 6.15: Zoom of the first of the upper 20th percentile of the graph in Figure 6.14.
6.5.3 Announcement Diffusion Speed

A central requirement for our metadata diffusion solution is the timely reception of announcements by peers. To assess whether it is met or not, we measured the amount of time elapsed from the moment a new peer connects to the Overlay to the moment it receives the first announcement message: this time represents the window in which a peer searching for an item will not receive results for the corresponding subtitles; we want that window to be as small as possible.

That measure is complementary to the quickness of the diffusion coverage we measured in the previous paragraph: in a way, they both measure the same aspect, but while the former focuses on a global point of view, this one takes the point of view of single peers connecting to the Overlay.

We measured the distribution of the recorded waiting times in every experiment; Figure 6.16 shows the results as the cumulative distribution function of them among peers in one emulation.

It can be seen that more than half of the peers receive the announcement within 10 minutes from their first appearance in the overlay, while 90% of them receive it within 20 minutes. These values represent a very important result, since they are remarkably small if compared to the distribution of peers’ session lengths, whose
Finally, we wanted to measure the impact in terms of bandwidth usage of subtitles exchange through overlay messages. This parameter is highly influenced by users’ real behaviour since subtitle downloads are always initiated on demand, exception made for subscribers. Therefore, to get a more realistic estimate of its value, we decided to try to simulate users’ behaviour by modifying every emulated Tribler instance and make them autonomously decide whether to download a subtitle or not: every time an announcement for a subtitle is received for the first time, the instance decides to download it with a 33% chance. Moreover, since an human would not issue a download instantly as the announcement is received, we delay the request in time, for an amount which is randomly picked accordingly to a uniform distribution in the window (0, 3600] seconds.

Figure 6.17 and Figure 6.18 show the results for an emulation including one publisher and one subscriber: the first illustrates the total bandwidth used for subtitles exchange, while the seconds shows how this is split between the publisher and the

interquartile range is included in [1.4, 12+] hours.

### 6.5.4 Subtitles Communication Overhead

Figure 6.17: Total network traffic used to exchange subtitles content in time.
Figure 6.18: Network traffic used for subtitles exchanged split between the publisher (red line) and a subscriber (green line).
subscriber.

The size of the subtitle content which is exchanged is approximately 60 Kilobytes, covering a film 1 hour and 30 minutes long. The overall consumed bandwidth in 12 hours emulation amounts to around 3500 Kilobytes. A slightly faster growth is recognisable in the first part of the emulation, in which the majority of peers’ first appearances are concentrated. From Figure 6.18, notice that in the middle part of the emulation the line corresponding to the overhead on the publisher stops to grow since it goes down for a 4 hours period.

It is important to remark again that an accurate measurement would require a more accurate model of users’ behaviour: in fact, the random approach we adopted is unsatisfactory for several reasons. First, the assumption of having an uniform chance among all the peers in the Overlay of downloading a subtitles file is very unrealistic: more likely the distribution of requests for a given meta-content will show different concentrations in different zones of the Overlay, probably showing a correlation among semantically close peers. Second, we fail to model the fact that the amount of request for a given subtitle file depend on parameters like the video item’s popularity, the channel popularity, or the metadata freshness. Finally, the time window we picked to select the moment a metadata content is requested is completely arbitrary, and probably is not able to reflect all the complexities that guide the choice of a human rational mind.

Nonetheless, our results give us an important indication about the effectiveness of the subscription mechanism as a mean to increase content’s availability and to help the publisher distribute the content itself: in fact, as it can be seen from Figure 6.18, a considerable fraction of the total traffic used to distribute subtitles is undertaken by the subscriber, relieving part of the responsibility from the publishing peer.

6.6 The duplicates problem: a possible solution

As predicted in Chapter 4 (see Section 4.6.2) and verified during our emulation based evaluation in Section 6.5.1, a relevant fraction of the bandwidth used by Metis to spread metadata announcement creates an unnecessary overhead by disseminating duplicate messages: we remarked more then once that this fact does not represent a serious limitation in our prototype since the total amount of network traffic that gets wasted in one day arrives up to 24 kilobytes per peer (see again Figure 6.13), a value that is negligible if compared to the average bandwidth commonly available with cheap broadband Internet connectivity.

However, this phenomenon seriously restricts the possibility to enhance the protocol in order to support a larger and extensible set of metadata types other then subtitles. In fact, if we assume that, in a general scenario, we would represent rich announcement messages with XML documents describing a large set of metadata instances of different types, assuming also the average size of this kind of documents to be 10 kilobytes (instead of the 120 bytes of the current announcement
message), we can conclude — with some simple math — that the total traffic for duplicates in a day would amount to 2 megabytes per peer per published instance: it is clear that approaches other than direct announcement dissemination are necessary to be considered in order to face this problem.

In this Section we will propose two different alternatives and we will compare them and with the current protocol through an analysis based on simulations.

Figure 6.19 illustrates the steps involved in a gossip round between two peers: basically, whenever a peers receives the set of the announcements the sender has (which we will call set message from now on for convenience), it computes the set difference between its own and the received one, and picks the announcements to send from the result. In the second step of the protocol the answerer forwards the so created announcements list and includes along with it its own set message,
in order to reduce the number of exchanged messages.

We will present two variants of this simple protocol; they differ from each other in the way the announcement set is represented: the first relies on an exact representation based on UUIDs [42], while the second leverages a compressed and probabilistic data structure built using Bloom filters [10].

### 6.6.1 UUID based set

UUIDs [42] have been developed to provide unique and persistent Uniform Resource Names (URNs) which can be adopted without the need of central authorities to administer them: anyone can create a UUID and use it to identify something with reasonable confidence that it will never be unintentionally used for anything else. RFC 4122 defines five variants of UUIDs: each of them is a fixed-size 128 bit string, and they differ in the way the content of the string is determined: version 1 UUIDs are generated concatenating the MAC address of the originating node and a value which depends on the local clock; version 2 are very similar, except for the fact that part of the clock dependent value is replaced by the user’s POSIX UID (or GID); version 3 and version 5 are respectively based on the MD5 and SHA-1 hashes of the resources to represent, and version 4 is generated exploiting pseudo-random data sources.

The idea is to identify every announcement about metadata published in Metis with a UUID: for our discussion, we do not care which version is used as long as they are unique; identifiers are generated by a publisher at the moment it ingests new metadata in the system without need for coordination. A set message will coincide with a list of 16 bytes strings, corresponding to the identifiers of the locally available announcements. Upon delivery of a set message, the receiving peer will create an answer message by checking that the identifiers of the items it includes in it are not already in the received set. Using this method perfect difference sets can be computed.

### 6.6.2 Bloom filters based set

A Bloom filter [10] is a simple space-efficient data structure for representing a set. When answering membership queries a Bloom filter can respond with false positives (i.e., it is possible that it considers in the set an element which is actually not in it), but with no false negatives, i.e., it never makes mistakes in saying that an element is not a member of the set. Interestingly, space saving benefits outweigh the cost of false positives in many application scenarios; more interestingly, the probability of making mistakes has an optimal upper bound which is a tunable parameter which only depends on the maximum number of elements a filter can hold and on the ratio between the filter size and the number of elements in it.

A Bloom filter is an array of $m$ bits which is used in conjunction with $k$ distinct pseudo-random hash functions $h_0, \ldots, h_{k-1}$, such that for all the elements $x$ in the
elements’ universe $U$ it holds that:

$$h_i(x) = j \quad 0 \leq j < m \quad j \in \mathbb{N}$$

At initialisation, all the bits in the array are set to zeroes. To represent a set of elements $S$ of size $n$, one has to take each $x_s \in S$ and give it as input to all the $h_i$ hash functions: the output will be $k$ numbers between 0 and $m$, which identify the same number of indexes in the array; the bits corresponding to those indexes are to be set to 1.

To check whether any element $y \in U$ is in the set, $y$ is hashed with the same $k$ hash functions and the values corresponding to the generated array indexes are checked. Intuitively, if any of those bits is still 0, that means that the $y$ is not in the set since otherwise the bit would be a 1. On the contrary, if all the bits are 1s it is possible that the element is a member of the set: however, it is clear that this cannot be guaranteed since there are chances that those bits had been set to one as a consequence of the insertion of distinct elements, none of them equal to $y$.

It can be shown [12] that the false positive probability $p$ is a function of the parameters $k$ and $m/n$, the first being the number of hash functions, the second the number of bits used to represent each element in the set. More precisely these values are linked by the following equation:

$$p = \left(1 - e^{-kn/m}\right)^k$$

Given $m$ and $n$, an optimum value for $k$ exists minimising $p$, which is:

$$k = \ln 2 \cdot (m/n) \quad \Rightarrow \quad p = \left(\frac{1}{2}\right)^k$$

For instance, given a set of 512 elements ($n = 512$), using 6 bits per element ($m = 3072$), the optimal $k$ results in $[4.158] = 4$ and the probability of false positives becomes $p = 0.0560$.

Our idea is to use a Bloom filter to represent the set messages to be remotely sent: this compact representation would be much more bandwidth efficient compared with the UUID alternative, in the realistic assumption that the number of bits used to represent one element will be much more smaller then the size of its full identifier (i.e. $m/n \ll 128$). As before, the receiver of a Bloom filter would compose a response message by checking beforehand that the elements in the answer are not in the filter: since no false negatives are possible, none of the announcements sent on the network will be a duplicate. Since false positives are possible, it is likely to happen that a peer misses some significant announcement as a consequence of a mistake of the filter. However, the cost of missing something with very low and still tunable chances is worth to be payed compared to the possible bandwidth savings due to the high compression of the set representation: returning to the example above, a set message built using Bloom filters containing 512 elements would use only 396 bytes to be sent, while the UUID based variant would need 8 kilobytes.
6.6.3 Comparison and evaluation

To evaluate the proposed set-based protocols, we developed an ad-hoc simulator using the SimPy simulation framework [54]. SimPy is a discrete-event simulation language, based on and integrated with standard Python: it can be used to build models of complex systems and gives the modeller tools to represent active components (such as peers in our case), resources and messages, and to monitor meaningful parameters of the simulation.

Our simulator uses, as input, traces recorded from Tribler overlays which include peers connection and disconnection events and BuddyCast exchange events: from these data, the simulator recreates the overlay and simulates the exchange of metadata announcement messages between peers. The parameters we wanted to assess through our experiments are:

- **Bandwidth waste**, measured as the average waste of network traffic for duplicate messages.
- **Protocol efficiency**, measured as the network traffic used to spread set and announcement messages.
- **Reconciliation quality**, measured as the average intersection size between the announcements’ sets of each pair of simulated peers.

**Simulations Setup**

The simulations were designed as follows: we used the output of an existing BuddyCast simulator as input trace, which in turn was fed with the same real usage traces we used for our emulations (recall Section 6.2). From these, we extracted events regarding one week of activity for a set of 108 peers (100 peers + 8 super-peers used for their bootstrap): these overlay size has been chosen to grant faster simulations while still allowing to measure our goal parameters with sufficient confidence.

Each peer is given a cache of 5 megabytes in order to hold metadata announcements whose average size has been esteemed to be 10 kilobytes each; thus, each peer is able to store 512 announcements in its local cache. When a cache becomes full, a simple replacement policy is adopted which privileges announcements with bigger metadata timestamp (i.e. the newest ones). At simulation initialisation time, a universe of $4 \cdot 2^{10}$ metadata elements is generated and the cache of each peer in the simulation is filled at about its half capacity (i.e. 256 elements, plus or minus a delta of 30%) with a random sample of items from the metadata universe.

We simulated 5 different dissemination variants:

**direct.** It replicates the direct announcement approach used by the protocol implemented in the Metis prototype. At each gossip encounter two peers exchange the list of the most recent 50 announcements they have.
def _hashElement(self, element):
    h = element.md5
    hashes = list()
    for i in range(4):
        c = h[i]
        chunk_val = struct.unpack('!I', c)[0]
        hashes.append(chunk_val % self._m)
    return hashes

Listing 6.1: The 4 hash values for an element to be inserted in a Bloom filter are computed from its md5 hash

**UUID.** It implements the set based protocol using UUIDs for the set messages. At each encounter two peers perform the exchanges explained in the previous Section, with the set messages consisting of a list of 16 bytes identifiers, and the announcement messages being the most recent 50 announcement not in the received set.

**bloom 4.** It implements the set based protocol using Bloom filters to represent set messages. To size the filter, the parameter $n$ is set to the maximum cache capacity (i.e. 512) and $m/n$ is set to 4 (thus, $m = 2048$).

**bloom 6.** Same as Bloom 4, except that $m/n$ is set to 6, implying a filter size ($m$) of 3072 bits.

**bloom 8.** Same as Bloom 4, except that $m/n$ is set to 8, implying a filter size ($m$) of 4096 bits.

In all the three Bloom filter based variants, a fixed value of $k$ has been chosen ($k = 4$): this means that a suboptimal value has been used in the case of bloom 4 and bloom 8. However, this choice allowed us to develop a much faster set of hash functions which is very critical since in a single simulation something around 20 millions hashes are to be computed: Listing 6.1 shows how they are performed: the 128 bit MD5 hash of the element is taken and split in 4 chunks, each of 4 bytes; each chunk is then interpreted as a long unsigned integer represented in network byte order, and the remainder of the integer division by $m$ is used as hash value.

We decided not to support dynamic resizing of the filter since it would require its reconstruction from scratch every time, which is a very expensive operation: on the contrary, we build the filter already at its maximum capacity, i.e., ready to store all the elements in a full cache.

Table 6.3 summarises the characteristics of the different simulation setups.

121
Table 6.3: Summary of the parameters used in the different simulation settings.
mSS and MSS are respectively the minimum and the maximum size of set messages expressed in kilobytes; Mfp is the theoretical maximum false positive chance.

<table>
<thead>
<tr>
<th>Method</th>
<th>mSS</th>
<th>MSS</th>
<th>m</th>
<th>n</th>
<th>k</th>
<th>Mfp</th>
</tr>
</thead>
<tbody>
<tr>
<td>Direct</td>
<td>NA</td>
<td>NA</td>
<td>NA</td>
<td>NA</td>
<td>NA</td>
<td>NA</td>
</tr>
<tr>
<td>UUID</td>
<td>0</td>
<td>8</td>
<td>NA</td>
<td>NA</td>
<td>NA</td>
<td>0.0</td>
</tr>
<tr>
<td>Bloom 4</td>
<td>0.26</td>
<td>0.26</td>
<td>2048</td>
<td>512</td>
<td>4</td>
<td>0.159</td>
</tr>
<tr>
<td>Bloom 6</td>
<td>0.39</td>
<td>0.39</td>
<td>3072</td>
<td>512</td>
<td>4</td>
<td>0.062</td>
</tr>
<tr>
<td>Bloom 8</td>
<td>0.5</td>
<td>0.5</td>
<td>4096</td>
<td>512</td>
<td></td>
<td>0.024</td>
</tr>
</tbody>
</table>

Results analysis

In the analysis of the set based protocol, we remarked that no duplicates announcements should ever be sent since they are checked against the destination’s set before being forwarded; we said that this is also true for the Bloom based alternative, since no false negatives are allowed. This is, of course, only a first-level, coarse-grained approximation. In fact, there are chances in which duplicates are still received; consider, for instance, the following scenario:

1. Peer A sends its set message to Peer B, which — say — is \( \{x_1, x_2, x_3\} \).
2. Before receiving the answer from Peer B, Peer A initiates a gossip with another peer, Peer C, and it sends to it its set that is still \( \{x_1, x_2, x_3\} \).
3. Peer B (whose set is \( \{x_0, x_2, x_3, x_7\} \)) sends to Peer A the announcements corresponding to \( x_0 \) and \( x_7 \).
4. Peer C (whose set is \( \{x_7, x_8\} \)) sends to Peer A the announcements corresponding to \( x_7 \) and \( x_8 \).

It can be seen that at the end of the interaction Peer A receives a duplicate announcement, precisely the one corresponding to \( x_7 \).

The results from the simulations confirm this phenomenon, but they also show that the traffic it causes is very limited. Figure 6.20 compares the duplicate traffic generated in the simulation of one week using the direct approach and the set based approach. We report in the graph an average of the uuid and bloom methods, since we saw that there is no correlation between the particular set method and the number of duplicate messages, and the difference only depends on the particular simulation run. It can be seen that the gain of the set based protocol is significantly relevant, being of the order of \( 10^5 \) kilobytes per peer in a week.

To see whether the saved bandwidth is worth the overhead caused by the additional set messages per exchange, we measured the cumulative usage of bandwidth consumed, in average, by a peer to send the latter message type. This is what we
Figure 6.20: Comparison between the average traffic (in kilobytes) wasted in duplicates on a single node with the direct protocol and the set based protocol. It is important to notice that the two graphs are in different scale.
Figure 6.21: Cumulative network traffic used in average by a peer to send *send messages* during the simulation of a week period.
previously called protocol efficiency. Figure 6.21 shows how, even in the worst case, the overhead for sending set messages is way smaller than the one cause by duplicate announcements. Moreover, it’s possible to notice how the Bloom based technique largely outperforms the UUID based. In Figure 6.22 we show a zoom of the same results, which only compares the three variants of Bloom filters: as expected the differences are proportional to the number of bits used to represent the filters themselves.

Finally, we tried to compare the quality of the four different set approaches: we decided to measure that quality as the average size of the intersection of each pair of peers’ announcements: in an ideal situation, in fact, we would like to have all the peers have the same set of announcements. To register this measure we introduced an additional active component in the simulator, which, at fixed intervals, considers all the possible couple of peers and measures their set intersection. The results are shown in Figure 6.23. The values in the graph are normalised with respect to the intersection size at the beginning of the simulation, which is only dependent on the random sampling of metadata elements initially put in the peers’ caches. Unsurprisingly, the UUID based method results to be the best: in fact, using it, the
Figure 6.23: Variation of the average size of peers’ cache intersection, expressed in number of elements.
most recent announcement not possessed by a peer are always sent, since no false positives are possible. With Bloom filters, instead, the chance of making mistakes makes it possible recent announcement not owned by the destination not to be sent, since they may falsely result to be possessed by it: this is confirmed by the graph which shows that the intersection quality is strictly related to the false positive rate of the different filters (recall Table 6.3). Nevertheless, from this analysis, the differences in quality between the four approaches to set representation are very limited, spanning from a minimum average set intersection of 26.7 to a maximum of 32.07: this would suggest, at first glance, that the Bloom filter with four bits per element might be the most convenient solution.

However it has to be remarked that the possible combinations of simulation parameters are very ample, and we only explored a small subset of them: different Bloom filter sizes should be tested, and it should be checked how the variation of cache size and of initial cache usage impacts on the results. Moreover, in a real scenario, a non suboptimal number of hash functions would be used, and this may influence the performance, especially for what concerns larger Bloom filters. Finally, counting Bloom filters [11] may be considered to account for item’s removal from caches and fixed size filters may be substituted by resizable Bloom filters which are more likely to be used in a real scenario (scalable Bloom filters [3] may represent an interesting solution in that direction).

### Summary

In this Chapter we presented an analysis of the rich metadata dissemination system we designed in this Thesis and we implemented as a set of Python modules integrated with the Tribler P2P platform.

The dimension of Overlay networks, in terms of number of peers, create several difficulties to properly test a peer-to-peer application: among the possible approaches to conduct evaluation experiments there are the simulation-based approach which makes use of a model to abstract from complexities, the emulation-based which tries to recreate the characteristic of the network in a controlled environment, or the crawling-based method which consists in the analysis of usage data collected from real deployments.

We conducted our analysis through emulations based on real peer-to-peer usage traces; this method was able to give us the best confidence about the reliability of our software, and also acted as a System Testing phase in order to allow a faster integration of the code into the P2P-Next NextShare platform. To control the experiments we wrote and used an ad-hoc evaluation framework, consisting of two parts: Emulation Control is a set of bash and Python scripts allowing to control the life-cycle of an experiment and to collect the resulting logs; Logs Analyser is an extensible system able to process the collected logs and visualise them in synthetic graphs.

The announcement phase of our dissemination protocol exhibits a very little net-
work bandwidth overhead with respect to the non-extended ChannelCast protocol. Nonetheless, the analysis revealed an important problem in the amount of duplicate gossip messages sent through the Overlay: we showed how this problem is not imputable to our system, but it depends on some choices made in the design of the BuddyCast Gossip Engine in Tribler.

A good behaviour, instead, was found when studying the dissemination coverage (about the 80% of the emulated peers were covered by announcement messages within 25 minutes from the start of the experiment) and the dissemination speed (more then the 90% of the emulated peers receive the announcement within 20 minutes from their connection to the overlay).

Finally, we explored a possible solution to the duplicates problem by designing a protocol based on a three-way gossip exchange between peers at each BuddyCast round. These exchanges include set messages, which are a compact representation of the announcements already owned by the sending peer: they are used to avoid sending duplicates. We designed an ad-hoc simulator using the SimPy simulation language, and we evaluated this new approach implemented using two different types of representation for sets: the first is based on UUID lists, while the second on Bloom filters. Both the techniques are able to solve the duplicates problem, but the second shows to have a much smaller bandwidth cost compared to the first at the expense of a very limited performance penalty.
Conclusions and Future Work

In this Master’s Thesis we analysed the problem of metadata distribution in the context of peer-to-peer content-oriented systems, and we designed a possible solution to it. We implemented our ideas in Metis, a lightweight rich metadata dissemination system which uses an epidemic protocol to propagate announcements about the availability of user generated metadata, and a simple P2P request-response protocol to transfer the actual content. Our design efforts have been focused on the optimisation of goals such as full decentralisation, minimisation of the bandwidth used for the dissemination process, maximisation of the speed of information distribution and of the coverage of its spread among peers, and robustness against malicious behaviours. The system we realised has been integrated in Tribler, a widely deployed, production quality peer-to-peer platform created and distributed by the Parallel and Distributed Systems group at the Delft University of Technology, which kindly hosted and helped us during the preparation of this thesis. Through the use of Metis, Tribler adds the support for publishing and spreading subtitles for video content, inheriting all the benefits deriving from our dissemination approach which is able to grant a particularly lightweight bandwidth overhead and a fast coverage of large portions of the overlay with the information being spread.

We evaluated our prototype through an extensive set of emulations, recreating the dynamics of 140 peers inside an Overlay network. The results confirmed that the overhead of the gossip-based announcement protocol is particularly low, amounting to around 48 kilobytes of exchanged data per peer in one day. Metis exhibited good performance also for what concerns the speed and the coverage of the information distribution, with 80% of the emulated peers reached by announcements within 25 minutes of experiment time, and more then half of them receiving it within 10 minutes of uptime. Notwithstanding these good results, our experiments brought to light a relevant problem consisting in the waste of bandwidth for the exchange of duplicate announcements (i.e. announcement that the receiver does already have), which contributes to almost the 50% of the announcement related traffic. Even though, thanks to our lightweight approach, this corresponds to less then 24 kilobytes per peer per day, we explored and evaluated an alternative dissemination protocol based on Bloom filters, whose performance outweighed the previous approach in our simulations.
During the design and development of Metis several possible extensions and improvements have been left for future work. They consist in possible solutions to limitations we recognised in our work, or they are just ideas that we believe worth to investigate in order to improve the quality of our dissemination solution. Following we enumerate the most relevant ones:

- Elaborate on the explored direction of Bloom filters based dissemination techniques. The results we obtained from our simulations are strongly encouraging about the benefits the approach could give. However, a deeper study, including experiments in emulated and real deployments, is needed to better understand its properties and its possible drawbacks.

- Expand the support to a larger and possibly extensible set of metadata types in Metis. Currently, our prototype supports a single type, subtitles for video content: although this choice allowed us to focus on mechanism rather than on data modelling, it represents an obstacle for the expandability of the system.

- Related to the previous direction of development, we identify the need of an accurate analysis on possible models and representations for metadata: the goal would be to choose a standard and appropriate metadata description framework which, at the same time, grants flexibility, efficiency and extendability.

- Explore the possibility of adopting epidemic infection models in addition to the SI (infected forever) in our gossip protocols. An interesting starting point could be to perform a comparative analysis of SIR, SIS and SI epidemic protocols, measuring both their efficiency and their quality in terms of dissemination results in different application scenarios.

- Develop an informed peer selection function which better fits our channel based dissemination use case. A possible idea could be to use additional heuristic measures for peer selection, which take to account domain-related parameters such as the channel’s or item’s popularity or the history of previous interactions and content download actions, in order to increase the probability of a peer to be picked as the destination of a gossip round.
Bibliography


Appendix A

“On-The-Wire” Protocol Specification

The ChannelCast protocol [6] introduced the concept of *channels* in the Next-Share platform. A *channel* (also referred to as *moderator*) is a peer who submits metadata about a set of torrents. ChannelCast enables a fully distributed dissemination of this metadata that other peers can use to search and select high-quality contents before downloading them.

This proposal introduces a basic support for rich metadata in Next-Share, and consists of an extension of ChannelCast that aims to allow a scalable distribution of subtitle contents with minimum protocol overhead. The first rich metadata type we chose to introduce is *subtitles*: In fact we believe that localization of video contents can give a great improvement to user experience, providing at the same time a big benefit avoiding the creation of multiple swarms for different locale version of a single item. That would also permit a publisher to reach a larger audience.

The proposal consists of three parts: in the first part the process of subtitle dissemination will be described in terms of three distinct phases, namely “ingestion”, “announcement”, and “retrieval”; in the second part a lightweight extension of ChannelCast which enables subtitles announcement is outlined, while in the last part we will present two new Overlay messages used by peers to accomplish the retrieval phase.

For the rest of this chapter we will use the terms defined in Deliverable D4.0.3 [6] - Chapter 5, which we briefly report here for reader’s convenience:

*channel* also known as *moderator*, indicates any peer who publishes metadata about a non-empty set of torrents, and spreads it in a gossip-like fashion

*moderation* the set of metadata published by a moderator about an asset. In the previous version of ChannelCast the only type of published metadata was the torrent metadata.

*subscriber* a peer that has reported a positive vote (see VoteCast [6]) for a *channel*
A.1 Publishing a subtitle

A moderator — or the software agent that acts in his behalf — wanting to enrich some of his moderations with subtitles contents needs to go through a process consisting of three conceptually distinct phases.

In the first step - which we term content ingestion - a publisher selects a subtitle file to associate with one of his moderated torrents. The platform will check for subtitle validity and store the association locally in a database. Currently we only support subtitles encoded using the SubRip \cite{78} file format (.srt) mainly because we found them to be the most common in existing communities. Each provided subtitle file cannot be more then 1 MB big.

The second phase is content announcement: information about the moderations including subtitles is spread in the network through the updated ChannelCast protocol. More details about this phase will be given in section A.2.

The third aspect of the problem is content retrieval: it consists of the protocol steps that need to be performed in order for a peer to remotely retrieve the contents of a subtitle file (i.e. the contents of the .srt file), from an available source. A description of the protocol will be given in section A.3.

It is important to note that the three mentioned phases are not meant to be executed in a strictly sequential order. The only constraint is, not surprisingly, that content ingestion has to happen before the content itself can be announced to, or retrieved by another peer.

A.2 Extended ChannelCast Protocol

The ChannelCast protocol has been modified to allow moderators to publish subtitles in different languages and enrich their moderations with them. A published subtitle will be associated to the channel it was published in, and to the torrent the moderation refers to.

Information about available subtitles is spread along ChannelCast messages in a lightweight fashion: for each moderation that is sent a bitstring of 4 bytes - called languages mask is included. Each bit in the string corresponds to one of the supported languages (see section A.4), and represents the presence or absence of a subtitle published by the moderator in that language. While limiting the potential range of different idioms to 32 this choice guarantees a simple implementation with a constant (minimum) overhead even if subtitles in several languages are available. Along with the languages mask a list of 20 byte sha-1 checksums is sent: each entry corresponds to one of the bits set to 1 in the bitstring, and it is the checksum of the subtitle file in the language corresponding to the bit. These checksums are introduced to permit a secure identification of the contents of a subtitle during the retrieval phase, and to guarantee that no fake contents are introduced by other peers on the behalf of the moderator.

To allow authentication of the publisher and to guarantee the integrity of the an-
nouncement an ECDSA public key signature is taken over the subtitles announce-
ment fields and included in the announcement.

Finally, any peer gossiping an “enhanced” ChannelCast adds an additional field
in the announcement messages: called have mask, it also is a 4 bytes bitstring. The
have mask represents a subset of the languages mask and it informs the receiving
peer of which subtitle files are available at the sender’s node. This information can
be leveraged by a node to build a map indicating where subtitles are in the network,
and help him decide which peer(s) to query during the subtitle retrieval phase.

As a last remark it is important to notice that the ChannelCast propagation
policies are left unchanged from the previous version: a moderator exchanges
information about his metadata with peers selected by the BuddyCast algorithm,
which are in turn forwarded to other peers by his subscribers. A ChannelCast mes-
 sage may also be sent by a peer as answer of a remote search query. In the context
of subtitles dissemination our implementation leverages this forwarding mechan-
ism to speed up the replication of subtitle contents and thus increase their avail-
ability in the network: a subscriber automatically tries to retrieve all the available
subtitle contents from its subscribed channels, as he receives the announcement.

A.2.1 Updated ChannelCast Wire Format

We report here the complete CHANNELCAST message wire format, including
the new additions. A CHANNELCAST message contains a message id 225 and
a bencoded dictionary of 50 (key, value) pairs of moderations; 25 of them are a
subset of own moderations and the rest 25 are a subset of subscribed channels’
moderations. The key is the cryptographic signature of a moderation, which is a
67-byte binary string; this is created by the signing a tuple (publisher_id, infosh, torrenthash, time_stamp) with its private key and later can be verified by receiving
peers using the public key of the channel.

The value is a dictionary of the following fields:

- 'publisher_id’ PermID of channel, which is a 123-byte binary string
- 'publisher_name’ Name of channel, which is of string type
- 'infohash’ bencoded hash of the 'info’ field in the torrent, which is a 23-byte
  string
- 'torrenthash’ hash of the entire bdecoded dictionary of the torrent, which is a
  23-byte string
- 'torrentname’ Swarm name
- 'time_stamp’ time of publication
- 'rich_metadata’ rich metadata specification, as described in the following para-
  graph.
The semantics of each field, except the last one, is unchanged with respect to the previous CHANNELCAST message format. The 'rich_metadata' entry is the only new addition to the message, and is intended to be used to deliver rich metadata announcements. The value corresponding to the key 'rich_metadata' in the message dictionary, is the tuple (description, tstamp, langmask, checksum_list, signature, havemask). A description of each entry in the tuple follows. (See also figure A.1)

**description** a string of at most 500 bytes. Currently it is not used.

**tstamp** a bencoded integer representing the number of seconds elapsed since the Unix epoch (1st of January 1970) in UTC at the moment the rich metadata entries (i.e. the subtitles) were last modified.

**langmask** 6-bytes bencoded string, representing the 4 bytes languages mask, as described in section A.2

**checksum_list** a list of 23-byte bencoded strings corresponding to sha1 checksums. The number of entries in this list must be equal to the number of '1's in the languages mask, and each of them is the checksum of the .srt file published by the moderator for one language. The order of the elements must follow the order of the languages of the subtitles they refer to. See A.4.1 for reference about languages ordering.

**signature** a 67-bytes string corresponding to the ECDSA signature taken over the tuple (description, tstamp, langmask, checksum_list) with the channel's private key.

**havemask** a 6-bytes bencoded string representing the 4 bytes have mask, as described in section A.2. To be valid, an havemask must be such that the bitwise AND operation between langmask and havemask gives the havemask
itself, meaning that it is a valid subset of the languages represented by the languages mask. Notice also that this field is not used to generate the signature field.

A.3 Subtitles Retrieval

Once a peer is informed about the availability of a subtitle in a certain language for an item within a channel, he needs some mechanism to retrieve the actual subtitle contents. We have designed a very simple protocol to exchange subtitle contents through Overlay messages: this choice has been guided by the observation that subtitle contents are generally very small sized, and therefore their inclusion in overlay messages would not represent a significant overhead.

Two new asynchronous message types have been introduced, GET_SUBS and SUBS. With GET_SUBS a peer can query another peer for subtitle contents, while a SUBS messages is sent by peers in answer to GET_SUBS requests, and they deliver the actual subtitle contents as payloads.

When a peer wants to retrieve subtitles (in one or more languages) for an item published by a given moderator, he creates a GET_SUBS query and sends it to one or more peers. The number of peers simultaneously queried in our implementation is currently set to 3. The peers to query are chosen using the knowledge accumulated by the querying peer from the havemasks harvested from ChannelCast messages exchange. In the case the information provided by the havemasks is not sufficient, the moderator is always considered as a valid source for subtitles.

When a peer receives a GET_SUBS messages he may answer with a SUBS message in the case he has local copies of some of the requested subtitles. The SUBS message will include the entire contents of those that were locally available.

It is important to notice that a GET_SUBS query can receive more then one SUBS answer, if multiple peers are queried, or even none. Is up to the querying peer to discard duplicates or to handle the lack of answers.

A.3.1 Wire Format

A GET_SUBS message has message ID 224 and contains a bencoded list of exactly 3 entries:

- **publisher_id** PermID of the channel who published the desired subtitles, which is a 123-byte binary string
- **infohash** hash of the ’info’ field in the torrent the desired subtitles refer to, which is a 23-byte string.
- **request_mask** a 6-byte bencoded string, representing a 32-bit mask. Each ’1’ in the string represents a requested language. For reference about the meaning of the ’1’s in the request_mask please see the languages mask description in section A.2
A SUBS message has message ID 223 and contains a bencoded list of exactly 4 entries:

**publisher_id** A 123-byte string representing the PermID of the channel who published the subtitles included in the payload field. This must be equal to the PermID in the GET_SUBS request that originated this response message.

**infohash** A 23-byte string representing the hash of the 'info' field in the torrent the subtitles included in the payload field refer to. This must be equal to the infohash in the GET_SUBS request that originated this response message.

**answer_mask** a 6-byte bencoded string, representing a 32bit mask indicating the languages of the subtitles included in the payload field. The bitwise AND operation between the request_mask and the answer_mask must give the answer_mask itself, meaning that it represents a valid subset of the requested languages.

**payload** a list of variable length. The length of this list must be equal the number of '1' s in the answer_mask. Each entry in the list is an utf-8 encoded string of variable length, and corresponds to the contents of a subtitle file. Each entry is allowed to be up to 1 MB big, and the all the entries together are allowed to be up to 2 MB big. The order of the entries is relevant and it is used to identify the language it refers to. (See Section A.4.1)

### A.4 List of supported languages and codes

In Table A.1 we report the 32 languages for whom subtitle dissemination is supported in the current version of the system. The first column corresponds to a language code as defined in ISO 639-2[32], the second column is the common name of the
language in English, while the third column corresponds to the binary mask associated to the language.

Notice that each mask corresponds to a bit string containing exactly one ‘1’. All the binary language masks described in this chapter can be built performing a bitwise OR operation between the masks of single languages, and possibly padding the most significant bytes with ‘0’s to make up a 32-bit string.

A.4.1 Notes on language ordering

We define a total ordering between the 32 supported languages. This order is based on the lexicographical order of the ISO-639-2 language codes representing each language. Due to the way bit masks for each language have been defined it also corresponds also to the natural ordering of the integer values represented by the mask (interpreted as little endian).
<table>
<thead>
<tr>
<th>ISO 639-2</th>
<th>Language</th>
<th>bitmask</th>
</tr>
</thead>
<tbody>
<tr>
<td>ara</td>
<td>Arabic</td>
<td>0x1</td>
</tr>
<tr>
<td>ben</td>
<td>Bengali</td>
<td>0x2</td>
</tr>
<tr>
<td>ces</td>
<td>Czech</td>
<td>0x4</td>
</tr>
<tr>
<td>dan</td>
<td>Danish</td>
<td>0x8</td>
</tr>
<tr>
<td>deu</td>
<td>German</td>
<td>0x10</td>
</tr>
<tr>
<td>ell</td>
<td>Greek</td>
<td>0x20</td>
</tr>
<tr>
<td>eng</td>
<td>English</td>
<td>0x40</td>
</tr>
<tr>
<td>fas</td>
<td>Persian</td>
<td>0x80</td>
</tr>
<tr>
<td>fin</td>
<td>Finnish</td>
<td>0x100</td>
</tr>
<tr>
<td>fra</td>
<td>French</td>
<td>0x200</td>
</tr>
<tr>
<td>hin</td>
<td>Hindi</td>
<td>0x400</td>
</tr>
<tr>
<td>hrv</td>
<td>Croatian</td>
<td>0x800</td>
</tr>
<tr>
<td>hun</td>
<td>Hungarian</td>
<td>0x1000</td>
</tr>
<tr>
<td>ita</td>
<td>Italian</td>
<td>0x2000</td>
</tr>
<tr>
<td>jav</td>
<td>Javanese</td>
<td>0x4000</td>
</tr>
<tr>
<td>jpn</td>
<td>Japanese</td>
<td>0x8000</td>
</tr>
<tr>
<td>kor</td>
<td>Korean</td>
<td>0x10000</td>
</tr>
<tr>
<td>lit</td>
<td>Latvia</td>
<td>0x20000</td>
</tr>
<tr>
<td>msa</td>
<td>Malay</td>
<td>0x40000</td>
</tr>
<tr>
<td>nld</td>
<td>Dutch</td>
<td>0x80000</td>
</tr>
<tr>
<td>pan</td>
<td>Panjabi</td>
<td>0x100000</td>
</tr>
<tr>
<td>pol</td>
<td>Polish</td>
<td>0x200000</td>
</tr>
<tr>
<td>por</td>
<td>Portuguese</td>
<td>0x400000</td>
</tr>
<tr>
<td>ron</td>
<td>Romanian</td>
<td>0x800000</td>
</tr>
<tr>
<td>rus</td>
<td>Russian</td>
<td>0x1000000</td>
</tr>
<tr>
<td>spa</td>
<td>Spanish</td>
<td>0x2000000</td>
</tr>
<tr>
<td>srp</td>
<td>Serbian</td>
<td>0x4000000</td>
</tr>
<tr>
<td>swe</td>
<td>Swedish</td>
<td>0x8000000</td>
</tr>
<tr>
<td>tur</td>
<td>Turkish</td>
<td>0x10000000</td>
</tr>
<tr>
<td>ukr</td>
<td>Ukranian</td>
<td>0x20000000</td>
</tr>
<tr>
<td>vie</td>
<td>Vietnamese</td>
<td>0x40000000</td>
</tr>
<tr>
<td>zho</td>
<td>Chinese</td>
<td>0x80000000</td>
</tr>
</tbody>
</table>

Table A.1: List of Supported Languages
Appendix B

User Manual

This chapter is intended to be 10 minutes reference for readers who are only interested in how to use the Subtitles subsystem, and do not need to understand how things work in the details. Only the basic API functionalities will be introduced. For the full API please consult Appendix ??.

B.1 Introduction

The Subtitle Exchange Subsystems extends the channel concept in Triber adding the ability to publish, search and retrieve subtitles published within channels.

With it an user can associate one or more subtitles in different languages to an item (i.e. a torrent) in its channel. Other peers will be able to access information about subtitles availability in others’ channels and to retrieve them remotely. It is important to remark that every subtitle in a certain language is strictly associated to an item in a channel; since at the time of writing there is a one-to-one correspondence between items and torrents, a subtitle can be uniquely identified by the triple \((\text{channel}_\text{id}, \text{infohash}, \text{language})\), where \text{channel}_\text{id} is the PermID of the channel owner, \text{infohash} is the 20 bytes infohash of the torrent corresponding to the item, and \text{language} is the language of the subtitle.

Before explaining how to work with subtitles, it might be a good idea to list the functionalities that are currently implemented and the ones that are missing.

What the Subtitle Exchange Subsystem does:

- Allow a Tribler user to enrich items in its channel attaching subtitles to them.
- Disseminate information about subtitles availability throughout the Overlay Network.
- Replicate subtitle contents in the network at subscribers hosts.
- Provide a simple API trough which publish, search, and retrieve subtitles contents both locally or in a remote fashion.
from Tribler.Core.Subtitles.SubtitlesSupport \
import SubtitlesSupport

...

def method_using_subtitles():
    # retrieve the correct instance
    sub_supp = SubtitlesSupport.getInstance()

    # do things with it
    ...

...

Listing B.1: Retrieve the SubtitlesSupport instance

What the Subtitle Exchange Subsystem does not:

- Cope with any kind of GUI integration.
- Integrate and synchronise subtitles with video playback.
- Deal with different metadata other then subtitles (even tough everything is ready for an easy extension).
- Anything else that is not explicitly mentioned in this manual.

B.2 SubtitlesSupport Facade

The basic API is kept as simple as possible. So simple that is possible to perform all the basic interactions just calling methods of a single object acting as facade for the entire subsystem. This object is an instance of Tribler.Core.Subtitles.SubtitlesSupport.SubtitlesSupport from now on referred with the unqualified name SubtitlesSupport.

There is a single instance of this object in a running Tribler Session, and you should always use that instance, since it is initialized at startup with the proper arguments. To retrieve the static method SubtitlesSupport.getInstance can be called, as shown in listing B.1.

Once you have a reference to the singleton instance you can use its methods to publish, search and retrieve subtitle contents. In the next subsections we will show how to perform each of these tasks.

B.2.1 Publishing a subtitle

Let’s assume that an user wants to publish a subtitle for the torrent of a film he previously added to its channel. To do that he needs to have a copy of the file con-
Listing B.2: Publishing a subtitle

taining subtitles on his local machine. Currently only subtitles in SubRip format (.srt) [78] are supported.

A possible GUI will allow him to select the item in the channel to attach subtitle to, and of course to specify the local path where the .srt file is, and the language of the subtitle he wishes to publish.

After the user has performed the necessary associations the SubtitlesSupport instance can be used to do the actual publishing: all it is needed is to call the publishSubtitle() instance method.

- publishSubtitle(infohash, lang, pathToSrt):

  infohash is the binary string representing the infohash of the torrent to associate the subtitle to. The method assumes that such a torrent has already been added to the local channel.

  lang is a three character language code specifying the language of the subtitle; for a list of the supported language codes please see the on the wire protocol specification available at [60].

  pathToSrt is a local file system path pointing to the file containing a copy of the subtitle in srt format.

If everything works, after the method returns, the subtitle is copied in a dedicated directory located into the Tribler state dir, and all the necessary information is stored in the MegaCache. Dissemination for the added subtitle automatically starts. An exemple of the subtitle publishing procedure is shown in listing B.2.

B.2.2 Searching for available subtitles

There are two different methods in SubtitlesSupport that are intended to be used to retrieve information about available subtitles.
• getSubtitleInfos(channel, infohash)
• getSubtitleInfosForInfohash(infohash)

The first one can be used to search for subtitle for an element in a particular channel; this can be useful for example while browsing a channel. On the other hand the second method, given an infohash, searches for subtitles available in all the known channels. This can be used, for example, while showing the results from a keyword search, in order to show all the subtitles associated to an hit in different channels.

getSubtitleInfos() returns a dictionary: each value is an instance of SubtitleInfo and its key is a three characters string corresponding to the language code of the subtitle. A returned dictionary could look like:

```
{ 'eng' : subtitleInfo1,
  'ita' : subtitleInfo2,
  'deu' : subtitleInfo3 }
```

In the example there are three subtitles available, respectively in English, Italian and German.

Similarly getSubtitleInfosForInfohash() returns a dictionary: this time the keys of the dictionary are channel identifiers (PermIDs), and their associated value is itself a dictionary whose keys are language codes and whose values are SubtitleInfo instances:

```
{ channel_id1 : { 'nld' : subtitleInfo1,
               'rus' : subtitleInfo2 },
   channel_id2 : { 'kor' : subtitleInfoN } }
```

In this case subtitles were found in two channels identified by PermIDs channel_id1 and channel_id2. In the first subtitles in Dutch and Russian are available; in the second only subtitles in Korean have been published.

The SubtitleInfo class is defined in module Tribler.Core.Subtitles.MetadataDomainObjects.SubtitleInfo. It contains all the basic information about a subtitle, as shown in listing B.3, through several examples.

### B.2.3 Retrieving a subtitle

Assume that you’ve searched for some subtitles for a torrent, and now you have a bunch of SubtitleInfo instances, as it has been explained in Section B.2.2.

For some of them the property path will be set to a local path in the file system pointing to the actual .srt file. You can double check that this file actually exists.
# sub_supp is the SubtitleSupport singleton instance
# retrieved before
# channel and infohash are byte strings identifying
# a channel and the infohash of a torrent in it
results = sub_supp.getSubtitleInfos(channel, infohash)

for lang, subInfo in results.items():
    print "Hit for a subtitle in %s" % lang

    # the lang property of a SubtitleInfo
    # reflects the key in the dictionary
    assert lang == subInfo.lang

    # SHA-1 checksum of the subtitle. It is always
    # set for a valid subtitle
    sha1Checksum = subInfo.checksum
    print "Subtitle’s sha1 is \%s" % sha1Checksum

    localPath = subInfo.path

    # if localPath is None it means that the subtitle
    # is not locally available
    if localPath is None:
        # the subtitleExists method does the same
        # thing
        assert not subInfo.subtitleExists()
        print "Subtitle not available locally"
    else:
        print "Subtitle is available at \%s" % localPath

        # if the subtitle is available the
        # checksum can be verified
        print "Checking checksum..."
        if subInfo.verifyChecksum():
            print "OK"
        else
            print "FAIL"

...
sub_supp = SubtitleSupport.getInstance()

# we have the subInfo SubtitleInfo instance from previous
# interactions

def call_me_back(retrieved_sub_info):
    # notice that retrieved_sub_info might not be the same
    # instance as subInfo
    print "Subtitle now available at %s" % 
        retrieved_sub_info.path

sub_supp.retrieveSubtitleContent(channel, infohash, 
                                  subInfo, call_me_back)

print "Subtitle request scheduled to be sent"

...
B.3 Final remarks

That’s it: now you should know how to interoperate with the Subtitles Subsystem for what concerns the basic functionalities. If you’re interest in understanding how things work please consult the Full API documentation and the other documents listed in the references.
Appendix C

Glossary

.torrent  Also referred to as torrent metadata, it must not be confused with our use of the term Metadata in our Thesis. A .torrent is a file describing a BitTorrent swarm, and it is used by a peer to join that swarm.

Announcement messages  They are exchanged between peers during the dissemination announcement phase, and they contain information about the existing metadata they know about.

Announcement phase  In our system, it is the process through which users are informed about the existence of metadata. It enables them to perform discovery of metadata.

Announcement waiting time  It is the amount of time elapsed from the moment a peer connects to the overlay to the moment it receives the announcement message for a given item.

BuddyCast  The core Tribler epidemic protocol. It can be logically divided in two parts; A “Gossip Engine” provides the basic mechanisms for all the epidemic protocols in Tribler, while the “Data Exchange” acts both as a peer sampling service for the Gossip Engine, and collects data for distributed content discovery.

Channel  A container of Items published (but not necessarily created) by a single Publisher, which is said to own the Channel. One peer can have only a single channel, therefore they can be interchangeably identified.

ChannelCast  Gossip-based epidemic protocol used by the Tribler platform to disseminate information about Channels throughout the overlay.

Churn event  Any event which modifies the topology or the membership set of an Overlay Network.

Churn rate  The frequency of topology changes in an Overlay network, i.e., the rate at which peers connect and disconnect to/from the overlay.
**Crawling** Evaluation method, which consists in collecting and analysing usage data from deployments of the system in its real environment.

**Emulation Control** Component of our evaluation framework. It has the role to coordinate the actions of the emulated Tribler instances, and to collect the activities logs they produce.

**Emulation** Evaluation approach, aiming to reproduce the exact behaviour of the system being studied in a controlled environment.

**Epidemic Protocols** See Gossip Protocols.

**Gossip Interval** In a gossip protocol, it identifies the time interval between a gossip round and the next one.

**Gossip Protocols** Family of distributed communication protocols, used to probabilistically disseminate information across a network. They are characterised by the periodic exchange of messages between nodes, which solely act basing on local knowledge.

**Gossip Round** In a gossip protocol, it identifies a message exchange between one node and another.

**Infohash** A 20 bytes string, uniquely identifying a BitTorrent swarm: it is generated by performing a SHA-1 digest on the info section of the .torrent file associated to a swarm.

**Ingestion phase** In our system, it identifies the process allowing a publisher to enrich items in his channel with metadata, and to make them available to other users in the network.

**Item** An audiovisual content being distributed to peers in a peer-to-peer platform.

**Leecher** Within a BitTorrent swarm, a peer trying to get a complete copy of the file being shared.

**Logs Analyser** Part of our evaluation framework. Given the logs of an emulation run, it produces 2-D plots summarising the parameters observed during the experiment.

**MegaCaches** The set of components in Tribler which provide a peer local storage facilities.

**Meta-content** The actual binary representation of a Rich Metadata instance.

**Metadata** See Rich Metadata.

**Metis** The prototype of the Rich Metadata Dissemination system designed in this Thesis, built as an extension of the Tribler P2P platform.
**Network Thread**  One of Tribler’s thread of control, executing all the operations on the network sockets.

**Node**  A physical host running one or more instances of a peer-to-peer application.

**Overlay Application**  A service running in Tribler which uses the overlay swarm to interact with other peers and implement some desired functionalities. Examples of currently implemented applications are BuddyCast, ChannelCast and Metis.

**Overlay Swarm**  A BitTorrent swarm including all the Tribler compatible peers.

**Overlay Thread**  One of Tribler’s thread of control, executing all the Overlay Applications’ code.

**Overlay**  In Peer-to-Peer, an it is an abstraction of network built on top of the IP network, offering peers a wide range of advanced services such as membership management, message routing and delivery, data location, failure tolerance, authentication and anonymity. Often the term is used in the Thesis referring to the Tribler Overlay in particular: it is a particular overlay implementation, using the BitTorrent protocol for membership management and message routing. See also Structured Overlay and Unstructured Overlay.

**Peer Agent**  Part of the Emulation control component. An instance of peer agent is executed for each emulated peer in one experiment, and it has the role to control and issue commands to it.

**Peer Selection Function**  In a gossip protocol, it has the role to periodically select a node to exchange a message with.

**Peer**  An instance of a peer-to-peer application running on a node.

**PermId**  An Elliptic Curve Cryptography (ECC) public key, used by Tribler peers as their permanent identifier and allowing them to authenticate with each other, and to guarantee the integrity of their messages. In our design the PermId is also used to identify a peer’s Channel.

**Publisher**  The role associated to a peer-to-peer user which shares and disseminates content in the overlay network. Since we assume that the platform does not support multiple users, there can be at most one publisher for each peer.

**Retrieval phase**  In our system, is the process through which an user retrieves the actual content of some metadata it is interested in.

**Rich Metadata**  Digitally representable description of an Item. Examples of Rich Metadata are subtitles, thumbnails, video previews, timed captions. In our domain model a Rich Metadata instance always refers to a single Item in one Channel.
**Seeder** Within a BitTorrent swarm, a peer having a complete copy of the file being shared and uploading it to other peers in the swarm.

**Session** Referred to a peer, it identifies the time window during which the peer is online and connected to the Overlay.

**Set message** In the set-based announcement dissemination protocol, it is a message which contains the representation of all the announcements available at the sender’s host.

**Simulation** Evaluation approach, consisting in the creation and execution of a simplified model of the system being tested. Simulations have the benefit of being easy and cheap to set up and run, but the level of abstraction of the system model has to be carefully chosen not to loose important characteristics of the real system.

**Structured Overlay** A type of Overlay Network arranging peers in a well defined network topology which is exploited in the membership management and routing algorithms. Structured overlays are able to give formal guarantees about the complexity of their operations, but they show robustness problems in networks characterised by high churn.

**Subscriber (of a Channel)** A peer in the Overlay having explicitly marked a preference for that Channel. In ChannelCast, a subscriber marks its preference by assigning a positive vote for the Channel.

**Subtitles** A particular Rich Metadata type, adding timed text to a video Item (i.e. subtitles). In Metis a subtitle instance is associated to a single Item in one Channel, and it is written in one of 32 supported languages.

**Super-peer** In P2P Overlays, it is a peer generally having better computational or bandwidth resource then the average, and it is elected to absolve special roles depending on the specific overlay. In Tribler a set of well-known super-peers are used to allow new peer to join the network.

**Swarm** In BitTorrent, the set of peers participating in the replication of a single file shared through the protocol.

**Taste Buddy** For a peer in Tribler, it is one among the top N similar peers. The similarity of two peers is computed by a recommendation algorithm based on peers’ download histories.

**Tribler** Open-source peer-to-peer content distribution platform developed by the Delft University of Technology. Tribler builds an unstructured overlay network around the BitTorrent protocol, and allows inter-swarm message routing, permanent peer identification, peer-to-peer video streaming, and decentralised search and discovery of content.
**Unstructured Overlay**  A type of Overlay Network which does not arrange peers in a precise structure. Topologies vary from one implementation to the other, and range from random graphs to hierarchical structures. Unstructured overlays are generally more robust to unstable network dynamics, but they are not able to give strong guarantees about their performance.
Acknowledgements

I would like to thank my supervisor, Prof. Paolo Bellavista, for his constant support throughout this project: more than one time he helped me staying focused on my work even when motivations were not particularly strong. Thanks also to Prof. Johan Pauwelse who accepted me as a member of the Tribler team and gave me the possibility to contribute, and to Ir. Nitin Chiulka who followed me during my work. I would also like to thank the Delft University of Technology which kindly hosted me for six months, and gave me everything I needed to comfortably accomplish my tasks in a pleasant environment.

I am indebted to all the people around the globe that made my experience in The Netherlands really great, making me feel at home: a special mention goes to Andrea, Andreas, Anna, Filippo, Flutra, Giuseppe and Tonina and to my flatmates Jelmer, Jarno, Laura, Rick, Roos, Esti, Erik and Daniele.

I would also like to thank all the people with whom I shared these last years of my life, which provided me a fun environment in which to grow and learn. Thanks to everyone from Via Martini, which I will always consider my second home: Mauro, Marco, Luca, Cristian, Fabio and also our always welcome guests Rosicò, Danielino, Gigi, Claudia, Beppe and the new come Antonio. Thanks to my colleagues and friends from the University which kept me focused and entertained me during these years of study: Andrea for constantly pushing me into challenging projects, Salvo for its party and organisational spirit, Savio for teaching me that people can be rasta or emo but still being themselves, Francoise for his beers in S. Francesco and S. Stefano, Nicola for his cheerful and particularly colourful presence and Daniele for his friend Klaudyna.

Thanks to all my friends from Campobasso with whom I grew up and who I will always feel strongly tied to me despite the fact we have seen each other almost only during holidays in the recent years: Valerio, Valeria, Jack, Demetrio, Rossana, Nidia and Ianna.

A special thought goes to Manola which has lovely been by my side during the last year, supporting all my choices and giving me a lot of confidence about myself.

Finally, the most important acknowledgement is reserved to my parents, Francesco and Anna, who raised me, taught me, and supported me spiritually and economically during the most important twenty-four years of my life.